Industrial and
Systems Engineering

SYMPHONY 5.0 User's Manual

Ted K. Ralphs[1]

[1]Department of Industrial and Systems Engineering, Lehigh University

ISE Technical Report 04T-020

LEHIGH UNIVERSITY

SYMPHONY **5.0 User's Manual** [1]

T.K. Ralphs[2]

November 11, 2004

[1]This research was partially supported by NSF Grant DMS-9527124 and Texas ATP Grant 97-3604-010. A revised version of Chapters 2 and 4 of this manual now appears in the Springer-Verlag book *Computational Combinatorial Optimization* edited by M. Jünger and D. Naddef, see http://link.springer.de/link/service/series/0558/tocs/t2241.htm

[2]Department of Industrial and Systems Engineering, Lehigh University, Bethlehem, PA 18017, `tkralphs@lehigh.edu`, `http://www.lehigh.edu/~tkr2`

Acknowledgments

First, many thanks are due to Laci Ladányi who worked with me on the development of an early precursor of SYMPHONY called COMPSys and who taught me much of what I know about programming. Many thanks are due also to Marta Esö, who wrote a draft of this manual for what was then COMPSys. Thanks are also due to Menal Guzelsoy, who was been instrumental in the development of SYMPHONY since version 4.0 and who helped update this manual. Thanks to Matthew Galati and Ondrej Medek, who also helped with the development of SYMPHONY. Finally, thanks to Leslie Trotter, William Cook, Cornell University, Rice University, Lehigh University, the National Science Foundation, and the state of Texas, all of whom have supported the development of SYMPHONY.

Contents

1 Introduction **1**

1.1 Introducing SYMPHONY 5.0 1
1.2 How to Use This Manual 2

2 Technical Background **3**

2.1 A Brief History 3
2.2 Related Work 4
2.3 Introduction to Branch, Cut, and Price 4
2.3.1 Branch and Bound 4
2.3.2 Branch, Cut, and Price 5

3 API Overview **9**

3.1 The Callable Library 9
3.2 The OSI Interface 11
3.3 User Callback Functions 13

4 Design Overview **15**

4.1 Design Approach 15
4.1.1 An Object-oriented Approach 15
4.1.2 Data Structures and Storage 16
4.1.3 Modular Implementation 18
4.1.4 Algorithm Summary 21
4.2 Details of the Implementation 22
4.2.1 The Master Module 22
4.2.2 The Linear Programming Module 25
4.2.3 The Tree Manager Module 28
4.2.4 The Cut Generator Module 29
4.2.5 The Cut Pool Module 29
4.3 Parallelizing BCP 30
4.3.1 Parallel Configurations 31
4.3.2 Inter-process Communication 31
4.3.3 Fault Tolerance 32

5 Installation **33**

5.1 Compiling the Library and Executable in Unix 34
5.1.1 Preparing for Sample Compilation. 34
5.1.2 Compiling the Sequential Version. 34

5.1.3 Compiling the Shared Memory Version. 35
5.1.4 Compiling the Distributed Version. 36
5.2 Compiling the Library and Executable in Windows 36
5.2.1 Using the NMAKE Utility 37
5.2.2 Using the MSVC++ Workspace 37
5.3 Compiling a Custom Application Using Callbacks 39
5.3.1 Unix 39
5.3.2 Microsoft Windows 39
5.3.3 Using the NMAKE Utility 39
5.3.4 Using the MSVC++ Workspace 40
5.4 Sample Applications 40

6 Development 41
6.1 Orienting Yourself 41
6.2 Writing the Callbacks 42
6.3 Data Structures 43
6.3.1 Internal Data Structures 43
6.3.2 User-defined Data Structures 43
6.4 Inter-process Communication for Distributed Computing 43
6.5 The LP Engine 44
6.6 Cut Generation 44
6.7 Advanced Compilation 44
6.7.1 Unix Operating Systems 44
6.7.2 Microsoft Windows 46
6.8 Debugging Your Application 46
6.8.1 The First Rule 46
6.8.2 Debugging with PVM 46
6.8.3 Using `Purify` and `Quantify` 46
6.8.4 Checking the Validity of Cuts and Tracing the Optimal Path 47
6.8.5 Using the `Interactive Graph Drawing` Software 47
6.8.6 Other Debugging Techniques 48
6.9 Controlling Execution and Output 48
6.10 Other Resources 48

7 Reference 49
7.1 Callable Library C API 49
7.1.1 Primary Interface Functions 50
7.1.2 Parameter Query and Modification 62
7.1.3 Solver Status Query Functions 69
7.1.4 Data Query Functions 75
7.1.5 Data Modification Functions 98
7.1.6 Warm Starting Functions 115
7.1.7 Sensitivity Analysis Functions 121
7.2 Callable Library C++ API 125
7.3 User Callback API 128
7.3.1 Master module callbacks 128
7.3.2 LP module callbacks 144

7.3.3 Cut generator module callbacks 176
7.3.4 Cut pool module callbacks 182
7.3.5 Draw graph module callbacks 188
7.4 Run-time Parameters 192
7.4.1 Global parameters 192
7.4.2 Master module parameters 192
7.4.3 Draw Graph parameters 193
7.4.4 Tree Manager parameters 194
7.4.5 LP parameters 197
7.4.6 Cut Generator Parameters 202
7.4.7 Cut Pool Parameters 202
7.4.8 C++ Interface/OSI Parameters 203

Bibliography **205**

Chapter 1

Introduction

1.1 Introducing SYMPHONY 5.0

Welcome to the SYMPHONY user's manual. Whether you are a new user or simply upgrading to version 5.0, this manual will help you get started with what we hope you will find to be a very useful framework for solving mixed-integer linear programs either using the generic tools provided or by developing a custom branch, cut, and price algorithm. There have been some very significant developments since the last version of SYMPHONY was released. IN particular, SYMPHONY is now a callable library with an interface whose look and feel is similar to other popular solvers. This change allows SYMPHONY to be used in a variety of new and powerful ways that were not possible before. For existing users, there have been a few minor changes to the API needed to make SYMPHONY thread-safe. Code written for previous versions of SYMPHONY will have to be ported. Instructions for porting from previous version are contained in the file `SYMPHONY-5.0/README-5.0`. As always, these changes have undoubtedly introduced bugs. There are now an even larger number of configurations in which SYMPHONY can be used and we have tested many of them, but it is simply not possible to test them all. Please keep this in mind and report all bugs that you find. Among the new enhancements and features are:

- SYMPHONY is now a C callable library with an interface whose look and feel is similar to other popular solvers. This interface works for SYMPHONY's built-in generic MILP solver, as well as any customized algorithm developed by implementing one or more of SYMPHONY's user callback functions. The interface is exactly the same for both sequential and parallel versions of the code.

- The callable library also has a C++ interface conforming to COIN-OR's Open Solver Interface standard for accessing LP and MILP solvers.

- SYMPHONY has been made thread-safe in order to allow multiple environments to be opened within a single executable.

- It is now possible to stop SYMPHONY during the solution process and then restart the computation later, even after modifying the problem data. The user can also save warm start information outside the solver environment and then reload it later into a different environment, in much the same way as can be done with a simplex-based linear programming solver. This allows the user to efficiently implement procedures, such as those for multi-criteria optimization, in which a series of similar MILPs must be solved.

- Along with the ability to perform warm starts, the user call also define permanent cut pools that persist between solver calls. This is useful for situations in which a series of MILPs needs to be solved and the cuts generated during one solution call are still valid during later calls.
- SYMPHONY now has the ability to enumerate the efficient solutions of a bicriteria MILP if the user specifies a second objective function. This is done using a new algorithm described in [33] and takes advantage of the warm starting capabilities of SYMPHONY.
- SYMPHONY has a very rudimentary to perform sensitivity analysis for MILP. This capability is till very much in the development stages, but is present in version 5.0.

1.2 How to Use This Manual

The manual is divided into seven chapters. The first is the introduction, which you are reading now. Chapter 2 contains background information. Those not familiar with the basic methodology of branch, cut, and price may want to read these sections, especially Section 2.3, where we briefly describe the techniques involved. Chapter 3 contains an overview of the API, both for the callable library and for the user callback functions. Chapter 4 contains further depth and a more complete description of the design and implementation of SYMPHONY. In Section 4.1, we describe the overall design of SYMPHONY without reference to the implementational details and with only passing reference to parallelism. In Section 4.2, we discuss the details of the implementation. In Section 4.3, we briefly discuss issues involved in parallel execution of SYMPHONY. It is not necessary to read Chapters 2 and 4 before undertaking development of a SYMPHONY application. Chapter 5 describes how to install and compile SYMPHONY. Many users will want to go straight to this section of the manual to get started quickly. Chapter 6 describes in detail how to develop a custom application using SYMPHONY. Chapter 7 contains reference material. Section 7.1 contains a description of the native C interface for the callable library. Section 7.2 contains a description of the interface for C++ environments. Section 7.3 contains a description of the user callback functions. SYMPHONY's parameters are described in Section 7.4. Please note that for reference use, the HTML version of this manual may be more practical, as the embedded hyperlinks make it easier to navigate.

Chapter 2

Technical Background

2.1 A Brief History

Since the inception of optimization as a recognized field of study in mathematics, researchers have been both intrigued and stymied by the difficulty of solving many of the most interesting classes of discrete optimization problems. Even combinatorial problems, though conceptually easy to model as integer programs, have long remained challenging to solve in practice. The last two decades have seen tremendous progress in our ability to solve large-scale discrete optimization problems. These advances have culminated in the approach that we now call ***branch and cut***, a technique (see [20, 31, 21]) which brings the computational tools of branch and bound algorithms together with the theoretical tools of polyhedral combinatorics. Indeed, in 1998, Applegate, Bixby, Chvátal, and Cook used this technique to solve a ***Traveling Salesman Problem*** instance with 13,509 cities, a full order of magnitude larger than what had been possible just a decade earlier [2] and two orders of magnitude larger than the largest problem that had been solved up until 1978. This feat becomes even more impressive when one realizes that the number of variables in the standard formulation for this problem is approximately the ***square*** of the number of cities. Hence, we are talking about solving a problem with roughly ***100 million variables***.

There are several reasons for this impressive progress. Perhaps the most important is the dramatic increase in available computing power over the last decade, both in terms of processor speed and memory. This increase in the power of hardware has subsequently facilitated the development of increasingly sophisticated software for optimization, built on a wealth of theoretical results. As software development has become a central theme of optimization research efforts, many theoretical results have been "re-discovered" in light of their new-found computational importance. Finally, the use of parallel computing has allowed researchers to further leverage their gains.

Because of the rapidly increasing sophistication of computational techniques, one of the main difficulties faced by researchers who wish to apply these techniques is the level of effort required to develop an efficient implementation. The inherent need for incorporating problem-dependent methods (most notably for dynamic generation of variables and cutting planes) has typically required the time-consuming development of custom implementations. Around 1993, this led to the development by two independent research groups of software libraries aimed at providing a generic framework that users could easily customize for use in a particular problem setting. One of these groups, headed by Jünger and Thienel, eventually produced ABACUS (A Branch And CUt System) [23], while the other, headed by the authors, produced what was then known as COMPSys (Combinatorial Optimization Multi-processing System). After several revisions to enable more broad functionality, COMPSys became SYMPHONY (Single- or Multi-Process Optimization over

Networks). A version of SYMPHONY written in C++, which we call COIN/BCP has also been produced at IBM under the COIN-OR project [26]. The COIN/BCP package takes substantially the same approach and has the same functionality as SYMPHONY, but has extended SYMPHONY's capabilities in some areas.

2.2 Related Work

The 1990's witnessed a broad development of software for discrete optimization. Almost without exception, these new software packages were based on the techniques of branch, cut, and price. The packages fell into two main categories—those based on general-purpose algorithms for solving mixed-integer linear programs (MILPs) (without the use of special structure) and those facilitating the use of special structure by interfacing with user-supplied, problem-specific subroutines. We will call packages in this second category *frameworks*. There have also been numerous special-purpose codes developed for use in particular problem settings.

Of the two categories, MILP solvers are the most common. Among the dozens of offerings in this category are MINTO [29], MIPO [3], bc-opt [9], and SIP [28]. Generic frameworks, on the other hand, are far less numerous. The three frameworks we have already mentioned (SYMPHONY, ABACUS, and COIN/BCP) are the most full-featured packages available. Several others, such as MINTO, originated as MILP solvers but have the capability of utilizing problem-specific subroutines. CONCORDE [2, 1], a package for solving the *Traveling Salesman Problem* (TSP), also deserves mention as the most sophisticated special-purpose code developed to date.

Other related software includes several frameworks for implementing parallel branch and bound. Frameworks for general parallel branch and bound include PUBB [35], BoB [5], PPBB-Lib [37], and PICO [11]. PARINO [25] and FATCOP [7] are parallel MILP solvers.

2.3 Introduction to Branch, Cut, and Price

2.3.1 Branch and Bound

Branch and bound is the broad class of algorithms from which branch, cut, and price is descended. A branch and bound algorithm uses a divide and conquer strategy to partition the solution space into *subproblems* and then optimizes individually over each subproblem. For instance, let S be the set of solutions to a given problem, and let $c \in \mathbf{R}^S$ be a vector of costs associated with members of S. Suppose we wish to determine a least cost member of S and we are given $\hat{s} \in S$, a "good" solution determined heuristically. Using branch and bound, we initially examine the entire solution space S. In the *processing* or *bounding* phase, we relax the problem. In so doing, we admit solutions that are not in the feasible set S. Solving this relaxation yields a lower bound on the value of an optimal solution. If the solution to this relaxation is a member of S or has cost equal to $\hat{s}$, then we are done—either the new solution or $\hat{s}$, respectively, is optimal. Otherwise, we identify n subsets of S, $S_1, \ldots, S_n$, such that $\cup_{i=1}^n S_i = S$. Each of these subsets is called a *subproblem*; $S_1, \ldots, S_n$ are sometimes called the *children* of S. We add the children of S to the list of *candidate subproblems* (those which need processing). This is called *branching*.

To continue the algorithm, we select one of the candidate subproblems and process it. There are four possible results. If we find a feasible solution better than $\hat{s}$, then we replace $\hat{s}$ with the new solution and continue. We may also find that the subproblem has no solutions, in which case we discard, or *prune* it. Otherwise, we compare the lower bound to our global upper bound. If it is greater than or equal to our current upper bound, then we may again prune the subproblem.

Bounding Operation
Input: A subproblem S, described in terms of a "small" set of inequalities L such that $S = \{x^S : s \quad F$ and $ax^S \leq \beta \quad (a, \beta) \quad L\}$ and α, an upper bound on the global optimal value.
Output: Either (1) an optimal solution $s \quad S$ to the subproblem, (2) a lower bound on the optimal value of the subproblem, or (3) a message `pruned` indicating that the subproblem should not be considered further.
Step 1. Set $C \leftarrow L$.
Step 2. Solve the LP $\min\{cx : ax \leq \beta \quad (a, \beta) \quad C\}$.
Step 3. If the LP has a feasible solution $\hat{x}$, then go to Step 4. Otherwise, STOP and output `pruned`. This subproblem has no feasible solutions.
Step 4. If $c\hat{x} < \alpha$, then go to Step 5. Otherwise, STOP and output `pruned`. This subproblem cannot produce a solution of value better than α.
Step 5. If $\hat{x}$ is the incidence vector of some $\hat{s} \quad S$, then $\hat{s}$ is the optimal solution to this subproblem. STOP and output $\hat{s}$ as s. Otherwise, apply separation algorithms and heuristics to $\hat{x}$ to get a set of violated inequalities C. If $C =$, then $c\hat{x}$ is a lower bound on the value of an optimal element of S. STOP and return $\hat{x}$ and the lower bound $c\hat{x}$. Otherwise, set $C \leftarrow C \quad C$ and go to Step 2.

Figure 2.1: Bounding in the branch and cut algorithm

Finally, if we cannot prune the subproblem, we are forced to branch and add the children of this subproblem to the list of active candidates. We continue in this way until the list of active subproblems is empty, at which point our current best solution is the optimal one.

2.3.2 Branch, Cut, and Price

In many applications, the bounding operation is accomplished using the tools of linear programming (LP), a technique first described in full generality by Hoffman and Padberg [21]. This general class of algorithms is known as *LP-based branch and bound*. Typically, the integrality constraints of an integer programming formulation of the problem are relaxed to obtain a *LP relaxation*, which is then solved to obtain a lower bound for the problem. In [31], Padberg and Rinaldi improved on this basic idea by describing a method of using globally valid inequalities (i.e., inequalities valid for the convex hull of integer solutions) to strengthen the LP relaxation. They called this technique *branch and cut*. Since then, many implementations (including ours) have been fashioned around the framework they described for solving the Traveling Salesman Problem.

As an example, let a combinatorial optimization problem $CP = (E, F)$ with *ground set* E and *feasible set* $F \quad 2^E$ be given along with a cost function $c \quad \mathbf{R}^E$. The incidence vectors corresponding to the members of F are sometimes specified as the the set of all incidence vectors obeying a (relatively) small set of inequalities. These inequalities are typically the ones used in the initial LP relaxation. Now let P be the convex hull of incidence vectors of members of F. Then we know by Weyl's Theorem (see [30]) that there exists a finite set L of inequalities valid for P such that

$$P = \{x \quad \mathbf{R}^n : ax \leq \beta \quad (a, \beta) \quad L\}. \tag{2.1}$$

The inequalities in L are the potential cutting planes to be added to the relaxation as needed. Unfortunately, it is usually difficult, if not impossible, to enumerate all of inequalities in L or we

Branching Operation
Input: A subproblem S and $\hat{x}$, the LP solution yielding the lower bound.
Output: $S_1, \ldots, S_p$ such that $S = \cup_{i=1}^{p} S_i$.
Step 1. Determine sets $L_1, \ldots, L_p$ of inequalities such that $S = \cup_{i=1}^{n} \{x \in S : ax \leq \beta \ \forall (a, \beta) \in L_i\}$ and $\hat{x} \notin \cup_{i=1}^{n} S_i$.
Step 2. Set $S_i = \{x \in S : ax \leq \beta \ \forall (a, \beta) \in L_i \cup L'\}$ where L' is the set of inequalities used to describe S.

Figure 2.2: Branching in the branch and cut algorithm

Generic Branch and Cut Algorithm
Input: A data array specifying the problem instance.
Output: The global optimal solution s^* to the problem instance.
Step 1. Generate a "good" feasible solution $\hat{s}$ using heuristics. Set $\alpha \leftarrow c(\hat{s})$.
Step 2. Generate the first subproblem S^I by constructing a small set L' of inequalities valid for P. Set $A \leftarrow \{S^I\}$.
Step 3. If $A = \emptyset$, STOP and output $\hat{s}$ as the global optimum s^*. Otherwise, choose some $S \in A$. Set $A \leftarrow A \setminus \{S\}$. Process S.
Step 4. If the result of Step 3 is a feasible solution $\overline{s}$, then $c\overline{s} < c\hat{s}$. Set $\hat{s} \leftarrow \overline{s}$ and $\alpha \leftarrow c(\overline{s})$ and go to Step 3. If the subproblem was pruned, go to Step 3. Otherwise, go to Step 5.
Step 5. Perform the branching operation. Add the set of subproblems generated to A and go to Step 3.

Figure 2.3: Description of the generic branch and cut algorithm

could simply solve the problem using linear programming. Instead, they are defined implicitly and we use separation algorithms and heuristics to generate these inequalities when they are violated. In Figure 2.1, we describe more precisely how the bounding operation is carried out in branch and cut.

Once we have failed to either prune the current subproblem or separate the current fractional solution from P, we are forced to branch. The branching operation is accomplished by specifying a set of hyperplanes which divide the current subproblem in such a way that the current solution is not feasible for the LP relaxation of any of the new subproblems. For example, in a combinatorial optimization problem, branching could be accomplished simply by fixing a variable whose current value is fractional to 0 in one branch and 1 in the other. The procedure is described more formally in Figure 2.2. Figure 2.3 gives a high level description of the generic branch and cut algorithm.

As with cutting planes, the columns of A can also be defined implicitly if n is large. If column i is not present in the current matrix, then variable x_i is implicitly taken to have value zero. The process of dynamically generating variables is called *pricing* in the jargon of linear programming, but can also be viewed as that of generating cutting planes for the dual of the current LP relaxation. Hence, LP-based branch and bound algorithms in which the variables are generated dynamically when needed are known as *branch and price* algorithms. In [4], Barnhart, et al. provide a thorough review of these methods.

When both variables and cutting planes are generated dynamically during LP-based branch and bound, the technique becomes known as *branch, cut, and price* (BCP). In such a scheme, there is a pleasing symmetry between the treatment of cuts and that of variables. We further examine this symmetry later in the manual. For now, however, it is important to note that while branch, cut, and price does combine ideas from both branch and cut and branch and price (which are very similar to each other anyway), combining the two techniques requires much more sophisticated methods than either one requires on its own. This is an important idea that is at the core of our design.

In the remainder of the manual, we often use the term ***search tree***. This term derives from the common representation of the list of subproblems as the nodes of a graph in which each subproblem is connected only to its parent and its children. Storing the subproblems in such a form is an important aspect of our global data structures. Since the subproblems correspond to the nodes of this graph, they are sometimes be referred to as ***nodes in the search tree*** or simply as ***nodes***. The ***root node*** or ***root*** of the tree is the node representing the initial subproblem.

Chapter 3

API Overview

SYMPHONY 5.0 is the first version of SYMPHONY to be implemented as a callable library with a new interface derived from the COIN-OR Open Solver Interface. This change markedly improves SYMPHONY's usability and flexibility. Below, we briefly describe the new API, the C++ interface, and the use of the user callback functions.

3.1 The Callable Library

SYMPHONY's callable library consists of a complete set of subroutines for loading and modifying problem data, setting parameters, and invoking solution algorithms. The user invokes these subroutines through the API specified in the header file `sym_api.h`. Some of the basic commands are described below. For the sake of brevity, the arguments have been left out.

`sym_open_environment()` Opens a new environment, and returns a pointer to it. This pointer then has to be passed as an argument to all other API subroutines (in the C++ interface, this pointer is maintained for the user).

`sym_parse_command_line()` Invokes the built-in parser for setting commonly used parameters, such as the file name which to read the problem data, via command-line switches. A call to this subroutine instructs SYMPHONY to parse the command line and set the appropriate parameters. This subroutine also sets all other parameter values to their defaults, so it should only called when this is desired.

`sym_load_problem()` Reads the problem data and sets up the root subproblem. This includes specifying which cuts and variables are in the *core* (those that are initially present in every subproblem during the search process) and the additional cuts and variables to be initially active in the root subproblem. By default, SYMPHONY reads an MPS or GMPL file specified by the user, but the user can override this default by implementing a user callback that reads the data from a file in a customized format (see Section 3.3).

`sym_find_initial_bounds()` Invokes the user callback to find initial bounds using a custom heuristic.

```
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);
}
```

Figure 3.1: Implementation of a generic MILP solver with the SYMPHONY C callable library.

`sym_solve()` Solves the currently loaded problem from scratch. This method is described in more detail in Section 4.2.1.

`sym_warm_solve()` Solves the currently loaded problem from a warm start. This method is described in more detail in Section 4.2.1.

`sym_mc_solve()` Solves the currently loaded problem as a multicriteria problem. This method is described in more detail in Section 4.2.1.

`sym_close_environment()` Frees all problem data and deletes the environment.

As an example of the use of the library functions, Figure 3.1 shows the code for implementing a generic MILP solver with default parameter settings. To read in an MPS file called `sample.mps` and solve it using this program, the following command would be issued:

```
symphony -F sample.mps
```

The user does not have to invoke a command to read the MPS file. During the call to `sym_parse_command_line()`, SYMPHONY determines that the user wants to read in an MPS file. During the subsequent call to `sym_load_problem()`, the file is read and the problem data stored. To read an GMPL file, the user would issue the command

```
symphony -F sample.mod -D sample.dat
```

Although the same command-line switch is used to specify the model file, the additional presence of the `-D` option indicates to SYMPHONY that the model file is in GMPL format and GLPK's GMPL parser is invoked [27]. Note that the interface and the code of Figure 3.1 is the same for both sequential and parallel computations. The choice between sequential and parallel execution modes is made at compile-time through modification of the makefile or the project settings, depending on the operating system.

To start the solution process from a warm start, the `sym_warm_solve()` command is used. SYMPHONY automatically records the warm start information resulting from the last solve call and restarts from that checkpoint if a call to `sym_warm_solve()` is made. Alternatively, external warm start information can be loaded manually. Figure 3.2 illustrates the use of the re-solve

```
int main(int argc, char **argv)
{
   sym_environment *env = sym_open_environment();
   sym_parse_command_line(env, argc, argv);
   sym_load_problem(env);
   sym_set_int_param(env, "find_first_feasible", TRUE);
   sym_set_int_param(env, "node_selection_strategy", DEPTH_FIRST_SEARCH);
   sym_solve(env);
   sym_set_int_param(env, "find_first_feasible", FALSE);
   sym_set_int_param(env, "node_selection_strategy", BEST_FIRST_SEARCH);
   sym_warm_solve(env);
}
```

Figure 3.2: Implementation of a dynamic MILP solver with SYMPHONY.

capability by showing the code for implementing a solver that changes from depth first search to best first search after the first feasible solution is found. The user can also modify problem data in between calls to the solver. Code for doing so is shown in Figure 3.3. In this example, the solver is allowed to process 100 nodes and then save the warm start information. Afterward, the original problem is solved to optimality, then is modified and re-solved from the saved checkpoint.

Finally, SYMPHONY now also has a bicriteria solve call. The applications of such a solver are numerous. Besides yielding the ability to closely examine the tradeoffs between competing objectives, the method can be used to perform detailed sensitivity analysis in a manner analogous to that which can be done with simplex based solvers for linear programs. As an example, suppose we would like to know exactly how the optimal objective function value for a given pure integer program depends on the value of a given objective function coefficient. Consider increasing the objective function coefficient of variable i from its current value. Taking the first objective function to be the original one and taking the second objective function to be the i^{th} unit vector, we can derive the desired sensitivity function by using the bicriteria solution algorithm to enumerate all supported solutions and breakpoints. This information can easily be used to obtain the desired function. Figure 3.4 shows the code for performing this analysis on variable 0.

In addition to the parts of the API we have just described, there are a number of standard subroutines for accessing and modifying problem data and parameters. These can be used between calls to the solver to change the behavior of the algorithm or to modify the instance being solved. These modifications are discussed in more detail in Section 4.2.1.

3.2 The OSI Interface

The Open Solver Interface (OSI) is a C++ class that provides a standard API for accessing a variety of solvers for mathematical programs. It is provided as part of the COIN-OR repository [26], along with a collection of solver-specific derived classes that translate OSI call into calls to the underlying libraries of the solvers. A code implemented using calls to the methods in the OSI base class can easily be linked with any solver for which there is an OSI interface. This allows development of solver-independent codes and eliminates many portability issues. The current incarnation of OSI supports only solvers for linear and mixed-integer linear programs, although a new version

```
int main(int argc, char **argv)
{
   warm_start_desc *ws;
   sym_environment *env = sym_open_environment();
   sym_parse_command_line(env, argc, argv);
   sym_load_problem(env);
   sym_set_int_param(env, "node_limit", 100);
   sym_set_int_param(env, "keep_warm_start", TRUE);
   sym_solve(env);
   ws = sym_get_warm_start(env);
   sym_set_int_param(env, "node_limit", -1);
   sym_warm_solve(env);
   sym_set_obj_coeff(env, 0, 100);
   sym_set_obj_coeff(env, 200, 150);
   sym_set_warm_start(ws);
   sym_warm_solve(env);
}
```

Figure 3.3: Use of SYMPHONY's warm start capability.

```
int main(int argc, char **argv)
{
   sym_environment *env = sym_open_environment();
   sym_parse_command_line(env, argc, argv);
   sym_load_problem(env);
   syn_set_obj2_coeff(env, 0, 1);
   sym_mc_solve(env);
}
```

Figure 3.4: Performing sensitivity analysis with SYMPHONY's bicriteria solver.

supporting a wider variety of solvers is currently under development.

We have implemented an OSI interface for SYMPHONY 5.0 that allows any solver built with SYMPHONY to be accessed through the OSI, including customized solvers and those configured to run on parallel architectures. To ease code maintenance, for each method in the OSI base class, there is a corresponding method in the callable library. The OSI methods are implemented simply as wrapped calls to the SYMPHONY callable library. When an instance of the OSI interface class is constructed, a call is made to `sym_open_environment()` and a pointer to the environment is stored in the class. Most subsequent calls within the class can then be made without any arguments. When the OSI object is destroyed, `sym_close_environment` is called and the environment is destroyed.

To fully support SYMPHONY's capabilities, we have extended the OSI interface to include some methods not in the base class. For example, we added calls equivalent to our `sym_parse_command_line()` and `sym_find_initial_bounds()`. Figure 3.5 shows the program of Figure 3.1 implemented using the OSI interface. Note that the code would be exactly the same for accessing any customized SYMPHONY solver, sequential or parallel.

```
int main(int argc, char **argv)
{
   OsiSymSolverInterface si;
   si.parseCommandLine(argc, argv);
   si.loadProblem();
   si.branchAndBound();
}
```

Figure 3.5: Implementation of a generic MILP solver with the SYMPHONY OSI interface.

Although we are using the OSI to access a MILP solver, the current version of the OSI is geared primarily toward support of solvers for linear programming (LP) problems. This is because LP solvers employing some version of the simplex algorithm support much richer functionality and a wider range of interface functions, due to their support of warm starting from previously saved checkpoints. This functionality is difficult to provide for MILP solvers. In SYMPHONY 5.0, we have implemented for MILPs some of the same functionality that has long been available for LP solvers. As such, our OSI interface supports warm starting and sensitivity analysis. The implementations of this functionality is straightforward at the moment, but will be improved in future versions.

3.3 User Callback Functions

The user's main avenues for customization of SYMPHONY are the tuning of parameters and the implementation of one or more of over 50 user callback functions. The callback functions allow the user to override SYMPHONY's default behavior for many of the functions performed as part of its algorithm. The user has complete control over branching, cutting plane generation, management of the cut pool and the LP relaxation, search and diving strategies, and limited column generation. The callback functions are grouped by module according to their functionality. The names of the callback functions begin with the prefix `user_`. For instance, the `user_find_cuts()` subroutine is used to implement subroutines for finding problem-specific cutting planes and is part of the cut generation module. A full list of callbacks is contained in Chapter 7.3.

Callbacks in SYMPHONY are implemented slightly differently than in other popular libraries. Each user function is called from a SYMPHONY *wrapper function* that interprets the user's return value and determines what action should be taken. If the user performs the required function, the wrapper function normally exits without further action. If the user requests that SYMPHONY perform a certain default action, then this is done. Files containing default function stubs for all callbacks are provided along with the SYMPHONY source code and must be compiled and linked with SYMPHONY's internal library functions to obtain an executable. Makefiles and Microsoft Visual C++ project files are provided for automatic compilation.

Chapter 4

Design Overview

4.1 Design Approach

SYMPHONY was designed with two major goals in mind—portability and ease of use. With respect to ease of use, we aimed for a "black box" design, whereby the user would not be required to know anything about the implementation of the library, but only about the user interface. With respect to portability, we aimed not only for it to be *possible* to use the framework in a wide variety of settings and on a wide variety of hardware, but also for it to perform *e ectively* in all these settings. Our primary measure of effectiveness was how well the framework would perform in comparison to a problem-specific (or hardware-specific) implementation written "from scratch."

It is important to point out that achieving such design goals involves a number of very difficult tradeoffs. For instance, ease of use is quite often at odds with efficiency. In several instances, we had to give up some efficiency to make the code easy to work with and to maintain a true black box implementation. Maintaining portability across a wide variety of hardware, both sequential and parallel, also required some difficult choices. For example, solving large-scale problems on sequential platforms requires extremely memory-efficient data structures in order to maintain the very large search trees that can be generated. These storage schemes, however, are highly centralized and do not scale well to large numbers of processors.

4.1.1 An Object-oriented Approach

As we have already alluded to, applying BCP to large-scale problems presents several difficult challenges. First and foremost is designing methods and data structures capable of handling the potentially huge numbers of cuts and variables that need to be accounted for during the solution process. The dynamic nature of the algorithm requires that we must also be able to efficiently move cuts and variables in and out of the *active set* of each search node at any time. A second, closely-related challenge is that of effectively dealing with the very large search trees that can be generated for difficult problem instances. This involves not only the important question of how to store the data, but also how to move it between modules during parallel execution. A final challenge in developing a generic framework, such as SYMPHONY, is to deal with these issues using a problem-independent approach.

Describing a node in the search tree consists of, among other things, specifying which cuts and variables are initially *active* in the subproblem. In fact, the vast majority of the methods in BCP that depend on the model are related to generating, manipulating, and storing the cuts and variables. Hence, SYMPHONY can be considered an object-oriented framework with the central "objects" being the cuts and variables. From the user's perspective, implementing a BCP algorithm using SYMPHONY consists primarily of specifying various properties of objects, such as how they are generated, how they are represented, and how they should be realized within the context of a particular subproblem.

With this approach, we achieved the "black box" structure by separating these problem-specific functions from the rest of the implementation. The internal library interfaces with the user's subroutines through a well-defined Application Program Interface (API) (see Section 7.3) and independently performs all the normal functions of BCP—tree management, LP solution, and cut pool management, as well as inter-process communication (when parallelism is employed). Although there are default options for many of the operations, the user can also assert control over the behavior of the algorithm by overriding the default methods or by parameter setting.

Although we have described our approach as being "object-oriented," we would like to point out that SYMPHONY is implemented in C, not C++. To avoid inefficiencies and enhance the modularity of the code (allowing for easy parallelization), we used a more "function-oriented" approach for the implementation of certain aspects of the framework. For instance, methods used for communicating data between modules are not naturally "object-oriented" because the type of data being communicated is usually not known by the message-passing interface. It is also common that efficiency considerations require that a particular method be performed on a whole set of objects at once rather than on just a single object. Simply invoking the same method sequentially on each of the members of the set can be extremely inefficient. In these cases, it is far better to define a method which operates on the whole set at once. In order to overcome these problems, we have also defined a set of *interface functions*, which are associated with the computational modules. These function is described in detail in Section 7.3.

4.1.2 Data Structures and Storage

Both the memory required to store the search tree and the time required to process a node are largely dependent on the number of objects (cuts and variables) that are active in each subproblem. Keeping this active set as small as possible is one of the keys to efficiently implementing BCP. For this reason, we chose data structures that enhance our ability to efficiently move objects in and out of the active set. Allowing sets of cuts and variables to move in and out of the linear programs simultaneously is one of the most significant challenges of BCP. We do this by maintaining an abstract *representation* of each global object that contains information about how to add it to a particular LP relaxation.

In the literature on linear and integer programming, the terms *cut* and *row* are typically used interchangeably. Similarly, *variable* and *column* are often used with similar meanings. In many situations, this is appropriate and does not cause confusion. However, in object-oriented BCP frameworks, such as SYMPHONY or ABACUS [23], a *cut* and a *row* are *fundamentally di erent objects*. A *cut* (also referred to as a *constraint*) is a user-defined representation of an abstract object which can only be realized as a row in an LP matrix *with respect to a particular set of active variables*. Similarly, a *variable* is a representation which can only be realized as a column of an LP matrix with respect to a *particular set of cuts*. This distinction between the *representation* and the *realization* of objects is a crucial design element and is what allows us to effectively address some of the challenges inherent in BCP. In the remainder of this section, we further discuss this distinction and its implications.

Variables

In SYMPHONY, problem variables are *represented* by a unique global index assigned to each variable by the user. This index represents each variable's position in a "virtual" global list known only to the user. The main requirement of this indexing scheme is that, given an index and a list of active cuts, the user must be able to generate the corresponding column to be added to the matrix. As an example, in problems where the variables correspond to the edges of an underlying graph, the index could be derived from a lexicographic ordering of the edges (when viewed as ordered pairs of nodes).

This indexing scheme provides a very compact representation, as well as a simple and effective means of moving variables in and out of the active set. However, it means that the user must have a priori knowledge of all problem variables and a method for indexing them. For combinatorial models such as the *Traveling Salesman Problem*, this does not present a problem. However, for some set partitioning models, for instance, the number of columns may not be known in advance. Even if the number of columns is known in advance, a viable indexing scheme may not be evident. Eliminating the indexing requirement by allowing variables to have abstract, user-defined representations (such as we do for cuts), would allow for more generality, but would also sacrifice some efficiency. A hybrid scheme, allowing the user to have both indexed and *algorithmic* variables (variables with user-defined representations) is planned for a future version of SYMPHONY.

For efficiency, the problem variables can be divided into two sets, the *base variables* and the *extra variables*. The base variables are active in all subproblems, whereas the extra variables can be added and removed. There is no theoretical difference between base variables and extra variables; however, designating a well-chosen set of base variables can significantly increase efficiency. Because they can move in and out of the problem, maintaining extra variables requires additional bookkeeping and computation. If the user has reason to believe a priori that a variable is "good" or has a high probability of having a non-zero value in some optimal solution to the problem, then that variable should be designated as a base variable. It is up to the user to designate which variables should be active in the root subproblem. Typically, when column generation is used, only base variables are active. Otherwise, all variables must be active in the root node.

Constraints

Because the global list of potential constraints (also called cuts) is not usually known a priori or is extremely large, constraints cannot generally be represented simply by a user-assigned index. Instead, each constraint is assigned a global index only after it becomes active in some subproblem. It is up to the user, if desired, to designate a compact *representation* for each class of constraints that is to be generated and to implement subroutines for converting from this compact representation to a matrix row, given the list of active variables. For instance, suppose that the set of nonzero variables in a particular class of constraints corresponds to the set of edges across a cut in a graph. Instead of storing the indices of each variable explicitly, one could simply store the set of nodes on one side ("shore") of the cut as a bit array. The constraint could then be constructed easily for any particular set of active variables (edges).

Just as with variables, the constraints are divided into *core constraints* and *extra constraints*. The core constraints are those that are active in every subproblem, whereas the extra constraints can be generated dynamically and are free to enter and leave as appropriate. Obviously, the set of core constraints must be known and constructed explicitly by the user. Extra constraints, on the other hand, are generated dynamically by the cut generator as they are violated. As with variables, a good set of core constraints can have a significant effect on efficiency.

Note that the user is not *required* to designate a compact representation scheme. Constraints can simply be represented explicitly as matrix rows with respect to the global set of variables. However, designating a compact form can result in large reductions in memory use if the number of variables in the problem is large.

Search Tree

Having described the basics of how objects are represented, we now describe the representation of search tree nodes. Since the base constraints and variables are present in every subproblem, only the indices of the extra constraints and variables are stored in each node's description. A complete description of the current basis is maintained to allow a warm start to the computation in each search node. This basis is either inherited from the parent, computed during strong branching (see Section 4.2.2), or comes from earlier partial processing of the node itself (see Section 4.2.3). Along with the set of active objects, we must also store the identity of the object(s) which were branched upon to generate the node. The branching operation is described in Section 4.2.2.

Because the set of active objects and the status of the basis do not tend to change much from parent to child, all of these data are stored as differences with respect to the parent when that description is smaller than the explicit one. This method of storing the entire tree is highly memory-efficient. The list of nodes that are candidates for processing is stored in a heap ordered by a comparison function defined by the search strategy (see 4.2.3). This allows efficient generation of the next node to be processed.

4.1.3 Modular Implementation

SYMPHONY's functions are grouped into five independent computational modules. This modular implementation not only facilitates code maintenance, but also allows easy and highly configurable parallelization. Depending on the computational setting, the modules can be compiled as either (1) a single sequential code, (2) a multi-threaded shared-memory parallel code, or (3) separate processes running in distributed fashion over a network. The modules pass data to each other either through shared memory (in the case of sequential computation or shared-memory parallelism) or through a message-passing protocol defined in a separate communications API (in the case of distributed execution). an schematic overview of the modules is presented in Figure 4.1. In the remainder of the section, we describe the modularization scheme and the implementation of each module in a sequential environment.

The Master Module

The *master module* includes functions that perform problem initialization and I/O. This module is the only persistent module and stores all static problem data. The other modules are created only during a solve call and destroyed afterward. All calls to the API are processed through the master module. These functions of the master module implement the following tasks:

- Initialize the environment.
- Set and maintain parameter values.
- Read and store static problem data for instance to be solved.
- Compute an initial upper bound using heuristics.

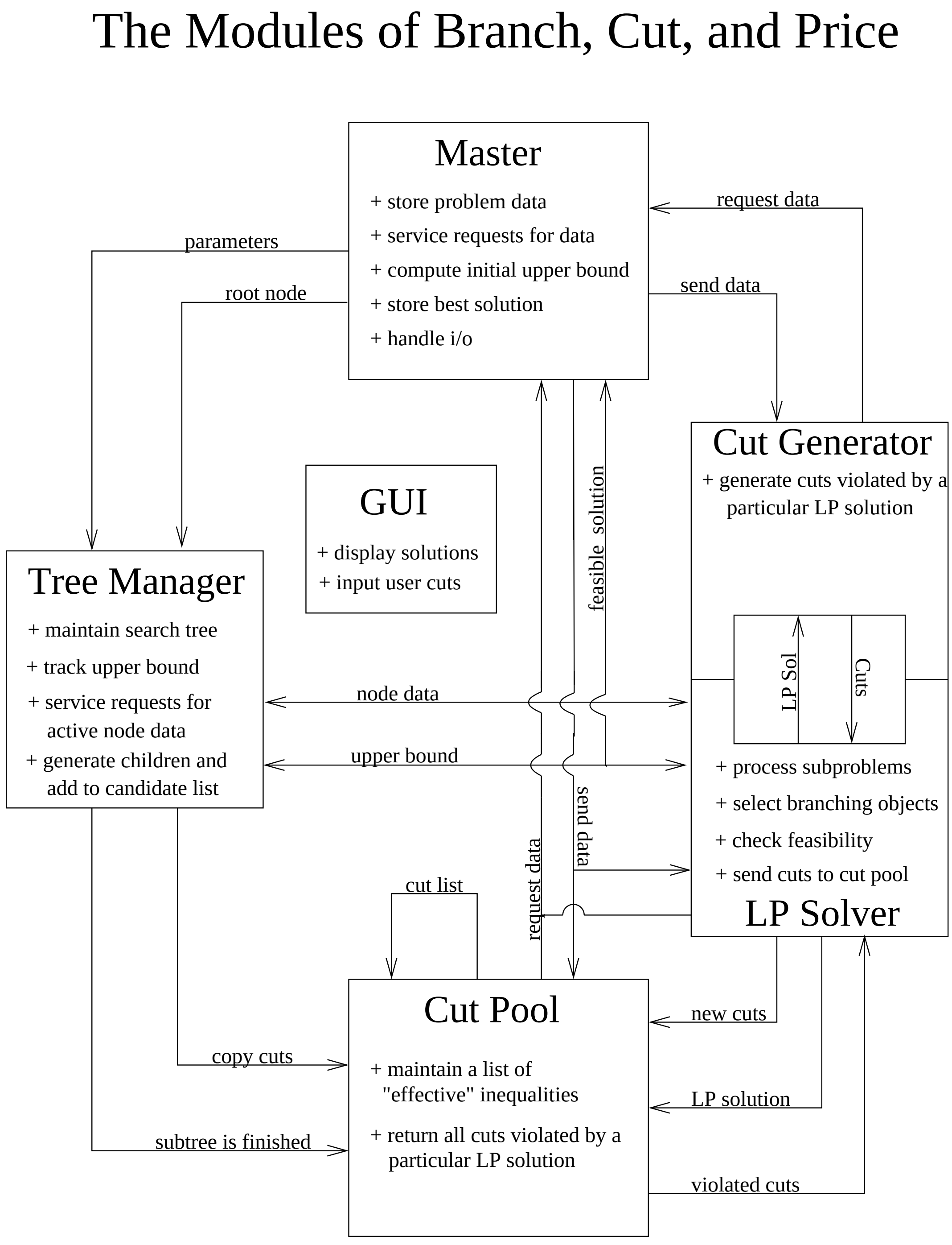

Figure 4.1: Schematic overview of the branch, cut, and price algorithm

- Perform problem preprocessing.
- Initialize the solution process, pass problem information to the solver modules and store the results after completion of the solve call.
- Track the status of associated processes during parallel solution calls.
- Act as a clearing house for output during the solution process.
- Store warm start information between solver calls.
- Service requests from the user through the API for problem data, problem modification, and parameter modification.

The Tree Manager Module

The *tree manager* controls the overall execution of the algorithm. It tracks the status of its worker modules, as well as that of the search tree, and distributes the subproblems to be processed to the LP module(s). Functions performed by the tree manager module are:

- Receive data for the root node and place it on the list of candidates for processing.
- Receive data for subproblems to be held for later processing.
- Handle requests from linear programming modules to release a subproblem for processing.
- Receive branching object information, set up data structures for the children, and add them to the list of candidate subproblems.
- Keep track of the global upper bound and notify all LP modules when it changes.
- Write current state information out to disk periodically to allow a restart in the event of a system crash.
- Keep track of run data and send it to the master program at termination.

The Linear Programming Module

The *linear programming* (LP) module is the most complex and computationally intensive of the five processes. Its job is to perform the bounding and branching operations. These operations are, of course, central to the performance of the algorithm. Functions performed by the LP module are:

- Inform the tree manager when a new subproblem is needed.
- Receive a subproblem and process it in conjunction with the cut generator and the cut pool.
- Decide which cuts should be sent to the global pool to be made available to other LP modules.
- If necessary, choose a branching object and send its description back to the tree manager.
- Perform the fathoming operation, including generating variables.

The Cut Generator Module

The ***cut generator*** performs only one function—generating valid inequalities violated by the current fractional solution and sending them back to the requesting LP process. Here are the functions performed by the cut generator module:

- Receive an LP solution and attempt to separate it from the convex hull of all solutions.
- Send generated valid inequalities back to the LP solver.
- When finished processing a solution vector, inform the LP not to expect any more cuts in case it is still waiting.

The Cut Pool Module

The concept of a ***cut pool*** was first suggested by Padberg and Rinaldi [31], and is based on the observation that in BCP, the inequalities which are generated while processing a particular node in the search tree are also generally valid and potentially useful at other nodes. Since generating these cuts is usually a relatively expensive operation, the cut pool maintains a list of the "best" or "strongest" cuts found in the tree so far for use in processing future subproblems. Hence, the cut pool functions as an auxiliary cut generator. More explicitly, here are the functions of the cut pool module:

- Receive cuts generated by other modules and store them.
- Receive an LP solution and return a set of cuts which this solution violates.
- Periodically purge "ineffective" and duplicate cuts to control its size.

4.1.4 Algorithm Summary

Currently, SYMPHONY is what is known as a single-pool BCP algorithm. The term ***single-pool*** refers to the fact that there is a single central list of candidate subproblems to be processed, which is maintained by the tree manager. Most sequential implementations use such a single-pool scheme. However, other schemes may be used in parallel implementations. For a description of various types of parallel branch and bound, see [17].

The user begins by initializing the SYMPHONY environment and can then invoke subroutines for reading in parameters and problem data, finding an initial upper bound, and designating the initial set of active cuts and variables in the root node. Once the user invokes a solve routine, a tree manager is created to manage the solution process. The tree manager module in turn sets up the cut pool module(s), the linear programming module(s), and the cut generator module(s). Currently, there are three solve calls supported by the API. The first call is the ***initial solve*** (see Section 4.2.1), which solves the problem from scratch without using warm start information. The second type of solve call is a ***warm solve***, which solves the problem using previously computed warm start information (see Section 4.2.1). Finally, there is a ***multicriteria solve*** call which is used to enumerate efficient solutions to a given multicriteria MILP (see Section 4.2.1).

During the solution process, the tree manager functions control the execution by maintaining the list of candidate subproblems and sending them to the LP modules as they become idle. The LP modules receive nodes from the tree manager, process them, branch (if required), and send back the identity of the chosen branching object to the tree manager, which in turn generates the children and places them on the list of candidates to be processed (see Section 4.2.2 for a description of the branching operation). A schematic summary of the algorithm is shown in Figure 4.1.

Currently, SYMPHONY is what is known as a single-pool BCP algorithm. The term *single-pool* refers to the fact that there is a single central list of candidate subproblems to be processed, which is maintained by the tree manager. Most sequential implementations use such a single-pool scheme. However, other schemes may be used in parallel implementations. For a description of various types of parallel branch and bound, see [17].

The preference ordering for processing nodes is a run-time parameter. Typically, the node with the smallest lower bound is chosen to be processed next since this strategy minimizes the overall size of the search tree. However, at times, it is advantageous to *dive* down in the tree. The concepts of *diving* and *search chains*, introduced in Section 4.2.3, extend the basic "best-first" approach.

We mentioned earlier that cuts and variables can be treated in a somewhat symmetric fashion. However, it should be clear by now that our current implementation favors the implementation of branch and cut algorithms, where the computational effort spent generating cuts dominates that of generating variables. Our methods of representation also clearly favor such problems. In a future version of the software, we plan to erase this bias by adding additional functionality for handling variable generation and storage. This is the approach already taken by of COIN/BCP [26]. For more discussion of the reasons for this bias and the differences between the treatment of cuts and variables, see Section 4.2.2.

4.2 Details of the Implementation

4.2.1 The Master Module

The primary functions performed by the master module were listed in Section 4.1.3. Here, we describe the implementational details of the various solve calls.

Initial Solve

Calling the initial solve method solves a given MILP from scratch, as described above. The first action taken is to create an instance of the tree manager module that will control execution of the algorithm. If the algorithm is to be executed in parallel on a distributed architecture, the master module spawns a separate tree manager process that will autonomously control the solution process. The tree manager in turn creates the modules for processing the nodes of the search tree, generating cuts, and maintaining cut pools. These modules work in concert to execute the solution process. When it makes sense, sets of two or more modules, such as a node processing module and a cut generation module may be combined to yield a single process in which the combined modules work in concert and communicate with each other through shared memory instead of across the network. When running as separate process, the modules communicate with each other using a standard communications protocol. Currently, the only option supported is PVM, but it would be relatively easy to add an MPI implementation.

The overall flow of the algorithm is similar to other branch and bound implementations and is detailed below. A priority queue of candidate subproblems available for processing is maintained at all times and the candidates are processed in an order determined by the search strategy. The algorithm terminates when the queue is empty or when another specified condition is satisfied. A new feature in SYMPHONY 5.0 is the ability to stop the computation based on exceeding a given time limit, exceeding a given limit on the number of processed nodes, achieving a target percentage gap between the upper and lower bounds, or finding the first feasible solution. After halting prematurely, the computation can be restarted after modifying parameters or problem data. This enables the implementation of a wide range of dynamic and on-line solution algorithms, as we describe next.

Solve from Warm Start

Among the utility classes in the COIN-OR repository is a base class for describing the data needed to warm start the solution process for a particular solver or class of solvers. To support this option for SYMPHONY, we have implemented such a warm start class for MILPs. The main content of the class is a compact description of the search tree at the time the computation was halted. This description contains complete information about the subproblem corresponding to each node in the search tree, including the branching decisions that lead to the creation of the node, the list of active variables and constraints, and warm start information for the subproblem itself (which is a linear program). All information is stored compactly using SYMPHONY's native data structures, which store only the differences between a child and its parent, rather than an explicit description of every node. This approach reduces the tree's description to a fraction of the size it would otherwise be. In addition to the tree itself, other relevant information regarding the status of the computation is recorded, such as the current bounds and best feasible solution found so far. Using the warm start class, the user can save a warm start to disk, read one from disk, or restart the computation at any point after modifying parameters or the problem data itself. This allows the user to easily implement periodic checkpointing, to design dynamic algorithms in which the parameters are modified after the gap reaches a certain threshold, or to modify problem data during the solution process if needed.

Modifying Parameters. The most straightforward use of the warm start class is to restart the solver after modifying problem parameters. To start the computation from a given warm start when the problem data has not been modified, the tree manager simply traverses the tree and adds those nodes marked as candidates for processing to the node queue. Once the queue has been reformed, the algorithm is then able to pick up exactly where it left off. Code for using the resolve command was shown in Figure 3.2. The situation is more challenging if the user modifies problem data in between calls to the solver. We address this situation next.

Modifying Problem Data. If the user modifies problem data in between calls to the solver, SYMPHONY must make corresponding modifications to the leaf nodes of the current search tree to allow execution of the algorithm to continue. In principle, any change to the original data that does not invalidate the subproblem warm start data, i.e., the basis information for the LP relaxation, can be accommodated. Currently, SYMPHONY can only handle modifications to the rim vectors of the original MILP. Methods for handling other modifications, such as the addition of columns or the modification of the constraint matrix itself, will be added in the future. To initialize the algorithm, each leaf node, regardless of its status after termination of the previous solve call, must be inserted into the queue of candidate nodes and reprocessed with the changed rim vectors. After this reprocessing, the computation can continue as usual. Optionally, the user can "trim the tree" before resolving. This consists of locating nodes whose descendants are all likely to be pruned in the resolve and eliminating those descendants in favor of processing the parent node itself. This ability could be extended to allow changes that invalidate the warm start data of some leaf nodes.

The ability to resolve after modifying problem data has a wide range of applications in practice. One obvious use is to allow dynamic modification of problem data during the solve procedure, or even after the procedure has been completed. Implementing such a solver is simply a matter of periodically stopping to check for user input describing a change to the problem. Another obvious application is in situations where it is known a priori that the user will be solving a sequence of very similar MILPs. This occurs, for instance, when implementing algorithms for multicriteria optimization, as we describe in Section 4.2.1. One approach to this is to solve a given "base problem" (possibly limiting the size of the warm start tree), save the warm start information from the base problem and then start each subsequent call from this same checkpoint. Code for implementing this was shown in Figure 3.3.

Bicriteria Solve

For those readers not familiar with bicriteria integer programming, we briefly review the basic notions here. For clarity, we restrict the discussion here to pure integer programs (ILPs), but the principles are easily generalized. A bicriteria ILP is a generalization of a standard ILP presented earlier that includes a second objective function, yielding an optimization problem of the form

$$\begin{aligned} \text{vmin } & [cx, dx], \\ \text{s.t. } & Ax \leq b, \\ & x \in \mathbb{Z}^n. \end{aligned} \tag{4.2}$$

The operator ***vmin*** is understood to mean that solving this program is the problem of generating ***efficient*** solutions, which are these feasible solutions p to (4.2) for which there does not exist a second distinct feasible solution q such that $cq \leq cp$ and $dq \leq dp$ and at least one inequality is strict. Note that (4.2) does not have a unique optimal solution value, but a set of pairs of solution values called ***outcomes***. The pairs of solution values corresponding to efficient solutions are called ***Pareto outcomes***. Surveys of methodology for for enumerating the Pareto outcomes of multicriteria integer programs are provided by Climaco et al. [8] and more recently by Ehrgott and Gandibleux [12, 13] and Ehrgott and Wiecek [14].

The bicriteria ILP (4.2) can be converted to a standard ILP by taking a nonnegative linear combination of the objective functions [18]. Without loss of generality, the weights can be scaled so they sum to one, resulting in a family of ILPs parameterized by a scalar $0 \leq \alpha \leq 1$, with the bicriteria objective function replaced by the ***weighted sum objective***

$$(\alpha c + (1 - \alpha)d)x. \tag{4.3}$$

Each selection of weight α produces a different single-objective problem. Solving the resulting ILP produces a Pareto outcome called a ***supported outcome***, since it is an extreme point on the convex lower envelope of the set of Pareto outcomes. Unfortunately, not all efficient outcomes are supported, so it is not possible to enumerate the set of Pareto outcomes by solving a sequence of ILPs from this parameterized family. To obtain all Pareto outcomes, one must replace the weighted sum objective (4.3) with an objective based on the ***weighted Chebyshev norm*** studied by Eswaran et al. [15] and Solanki [36]. If x^c is a solution to a weighted sum problem with $\alpha = 1$ and x^d is the solution with $\alpha = 0$, then the weighted Chebyshev norm of a feasible solution p is

$$\max\{\alpha(cp - cx^c), (1 - \alpha)(dp - dx^d)\}. \tag{4.4}$$

Although this objective function is not linear, it can easily be linearized by adding an artificial variable, resulting in a second parameterized family of ILPs. Under the assumption of ***uniform*** *dominance*, Bowman showed that an outcome is Pareto if and only if it can be obtained by solving some ILP in this family [6]. In [33], the authors presented a method for enumerating all Pareto outcomes by solving a sequence of ILPs in this parameterized family. By slightly perturbing the objective function, they also showed how to relax the uniform dominance assumption. Note that the set of all supported outcomes, which can be thought of as an approximation of the set of Pareto outcomes, can be similarly obtained by solving a sequence of ILPs with weighted sum objectives.

SYMPHONY 5.0 contains a generic implementation of the algorithm described in [33], along with a number of methods for approximating the set of Pareto outcomes. To support these capabilities, we have extended the OSI interface so that it allows the user to define a second objective function. Of course, we have also added a method for invoking this bicriteria solver called `multiCriteriaBranchAndBound()`. Relaxing the uniform dominance requirement requires the underlying ILP solver to have the ability to generate, among all optimal solutions to a ILP with a primary objective, a solution minimizing a given secondary objective. We added this capability to SYMPHONY through the use of optimality cuts, as described in [33].

Because implementing the algorithm requires the solution of a sequence of ILPs that vary only in their objective functions, it is possible to use warm starting to our advantage. Although the linearization of (4.4) requires modifying the constraint matrix from iteration to iteration, it is easy to show that these modifications cannot invalidate the basis. In the case of enumerating all supported outcomes, only the objective function is modified from one iteration to the next. In both cases, we save warm start information from the solution of the first ILP in the sequence and use it for each subsequent computation.

4.2.2 The Linear Programming Module

The LP module is at the core of the algorithm, as it performs the processing and bounding operations for each subproblem. A schematic diagram of the LP solver loop is presented in Fig. 4.2. The details of the implementation are discussed in the following sections.

The LP Engine

SYMPHONY requires the use of a third-party callable library (referred to as the ***LP engine*** or ***LP library***) to solve the LP relaxations once they are formulated. As with the user functions, SYMPHONY communicates with the LP engine through an API that converts SYMPHONY's internal data structures into those of the LP engine. Currently, the framework will only work with advanced, simplex-based LP engines, such as CPLEX [10], since the LP engine must be able to accept an advanced basis, and provide a variety of data to the framework during the solution process. The internal data structures used for maintaining the LP relaxations are similar to those of CPLEX and matrices are stored in the standard column-ordered format.

Managing the LP Relaxation

The majority of the computational effort of BCP is spent solving LPs and hence a major emphasis in the development was to make this process as efficient as possible. Besides using a good LP engine, the primary way in which this is done is by controlling the size of each relaxation, both in terms of number of active variables and number of active constraints.

The number of constraints is controlled through use of a local pool and through purging of ineffective constraints. When a cut is generated by the cut generator, it is first sent to the local cut pool. In each iteration, up to a specified number of the strongest cuts (measured by degree

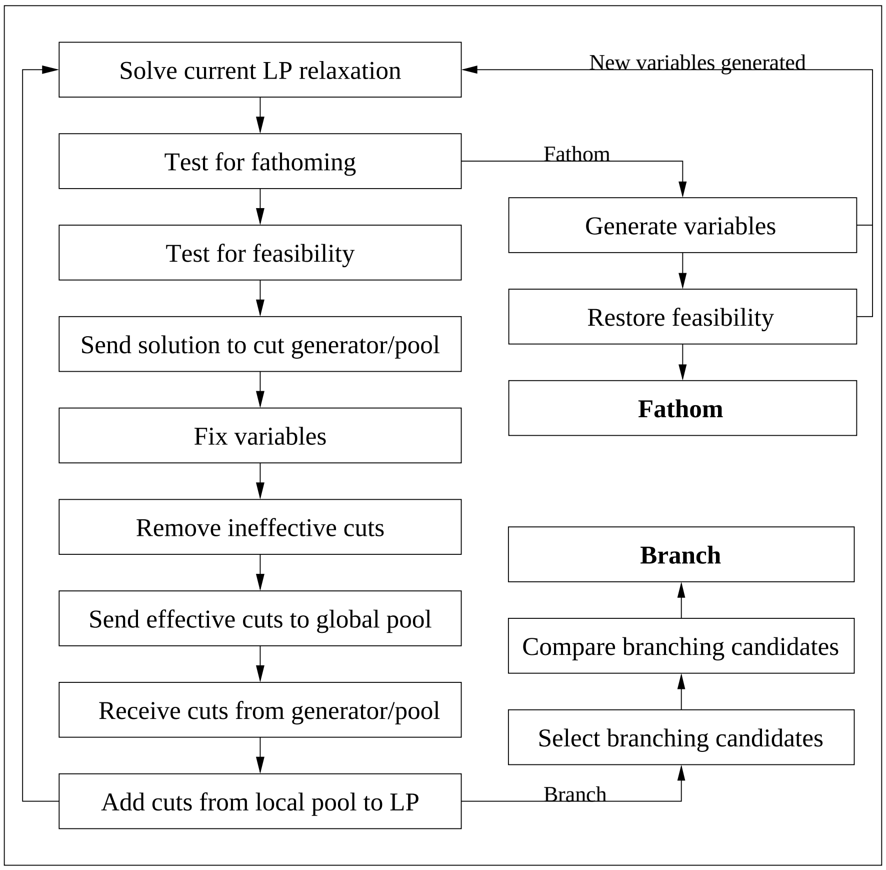

Figure 4.2: Overview of the LP solver loop

of violation) from the local pool are added to the problem. Cuts that are not strong enough to be added to the relaxation are eventually purged from the list. In addition, cuts are purged from the LP itself when they have been deemed ineffective for more than a specified number of iterations, where ineffective is defined as either (1) the corresponding slack variable is positive, (2) the corresponding slack variable is basic, or (3) the dual value corresponding to the row is zero (or very small). Cuts that have remained effective in the LP for a specified number of iterations are sent to the global pool where they can be used in later search nodes. Cuts that have been purged from the LP can be made active again if they later become violated.

The number of variables (columns) in the relaxation is controlled through ***reduced cost fixing*** and ***dynamic column generation***. Periodically, each active variable is ***priced*** to see if it can be fixed by reduced cost. That is, the LP reduced cost is examined in an effort to determine whether fixing that variable at one of its bounds would remove improving solutions; if not, the variable is fixed and removed from consideration. If the matrix is ***full*** at the time of the fixing, meaning that all unfixed variables are active, then the fixing is permanent for that subtree. Otherwise, it is temporary and only remains in force until the next time that columns are dynamically generated.

Because SYMPHONY was originally designed for combinatorial problems with relatively small numbers of variables, techniques for performing dynamic column generation are somewhat unrefined. Currently, variables are priced out sequentially by index, which can be costly. To improve the process of pricing variables, we plan to increase the symmetry between our methods for handling variables and those for handling cuts. This includes (1) allowing user-defined, abstract representations for variables, (2) allowing the use of "variable generators" analogous to cut generators, (3) implementing both global and local pools for variables, (4) implementing heuristics that help determine the order in which the indexed variables should be priced, and (5) allowing for methods of simultaneously pricing out large groups of variables. Much of this is already implemented in COIN/BCP.

Because pricing is computationally burdensome, it currently takes place only either (1) before branching (optional), or (2) when a node is about to be pruned (depending on the phase—see the description of the two-phase algorithm in Sect. 4.2.3). To use dynamic column generation, the user must supply a subroutine which generates the column corresponding to a particular user index, given the list of active constraints in the current relaxation. When column generation occurs, each column not currently active that has not been previously fixed by reduced cost is either priced out immediately, or becomes active in the current relaxation. Only a specified number of columns may enter the problem at a time, so when that limit is reached, column generation ceases. For further discussion of column generation, see Sect. 4.2.3, where the two-phase algorithm is described.

Since the matrix is stored in compressed form, considerable computation may be needed to add and remove rows and columns. Hence, rows and columns are only physically removed from the problem when there are sufficiently many to make it "worthwhile." Otherwise, deleted rows and columns remain in the matrix but are simply ignored by the computation. Note that because ineffective rows left in the matrix increase the size of the basis unnecessarily, it is usually advisable to adopt an aggressive strategy for row removal.

Branching

Branching takes place whenever either (1) both cut generation and column generation (if performed) have failed; (2) "tailing off" in the objective function value has been detected; or (3) the user chooses to force branching. Branching can take place on cuts or variables and can be fully automated or fully controlled by the user, as desired. Branching can result in as many children as the user desires, though two is typical. Once it is decided that branching will occur, the user must either select the list of candidates for *strong branching* (see below for the procedure) or allow SYMPHONY to do so automatically by using one of several built-in strategies, such as branching on the variable whose value is farthest from being integral. The number of candidates may depend on the level of the current node in the tree—it is usually best to expend more effort on branching near the top of the tree.

After the list of candidates is selected, each candidate is *pre-solved*, by performing a specified number of iterations of the dual simplex algorithm in each of the resulting subproblems. Based on the objective function values obtained in each of the potential children, the final branching object is selected, again either by the user or by built-in rule. This procedure of using exploratory LP information in this manner to select a branching candidate is commonly referred to as *strong branching*. When the branching object has been selected, the LP module sends a description of that object to the tree manager, which then creates the children and adds them to the list of candidate nodes. It is then up to the tree manager to specify which node the now-idle LP module should process next. This issue is further discussed below.

4.2.3 The Tree Manager Module

Managing the Search Tree

The tree manager's primary job is to control the execution of the algorithm by deciding which candidate node should be chosen as the next to be processed. This is done using either one of several built-in rules or a user-defined rule. Usually, the goal of the search strategy is to minimize overall running time, but it is sometimes also important to find good feasible solutions early in the search process. In general, there are two ways to decrease running time—either by decreasing the size of the search tree or by decreasing the time needed to process each search tree node.

To minimize the size of the search tree, the strategy is to select consistently that candidate node with the smallest associated lower bound. In theory, this strategy, sometimes called *best-first*, will lead the smallest possible search tree. However, we need to consider the time required to process each search tree node as well. This is affected by both the quality of the current upper bound and by such factors as communication overhead and node set-up costs. When considering these additional factors, it is sometimes be more effective to deviate from the best-first search order. We discuss the importance of such strategies below.

Search Chains and Diving

One reason for not strictly enforcing the search order is because it is somewhat expensive to construct a search node, send it to the LP solver, and set it up for processing. If, after branching, we choose to continue processing one of the children of the current subproblem, we avoid the set-up cost, as well as the cost of communicating the node description of the retained child subproblem back to the tree manager. This is called *diving* and the resulting chain of nodes is called a *search chain*. There are a number of rules for deciding when an LP module should be allowed to dive. One such rule is to look at the number of variables in the current LP solution that have fractional values. When this number is low, there may be a good chance of finding a feasible integer solution quickly by diving. This rule has the advantage of not requiring any global information. We also dive if one of the children is "close" to being the best node, where "close" is defined by a chosen parameter.

In addition to the time saved by avoiding reconstruction of the LP in the child, diving has the advantage of often leading quickly to the discovery of feasible solutions, as discussed above. Good upper bounds not only allow earlier pruning of unpromising search chains, but also should decrease the time needed to process each search tree node by allowing variables to be fixed by reduced cost.

The Two-Phase Algorithm

If no heuristic subroutine is available for generating feasible solutions quickly, then a unique two-phase algorithm can also be invoked. In the two-phase method, the algorithm is first run to completion on a specified set of core variables. Any node that would have been pruned in the first phase is instead sent to a pool of candidates for the second phase. If the set of core variables is small, but well-chosen, this first phase should be finished quickly and should result in a near-optimal solution. In addition, the first phase will produce a list of useful cuts. Using the upper bound and the list of cuts from the first phase, the root node is *repriced*—that is, it is reprocessed with the full set of variables and cuts. The hope is that most or all of the variables not included in the first phase will be priced out of the problem in the new root node. Any variable thus priced out can be eliminated from the problem globally. If we are successful at pricing out all of the inactive variables, we have shown that the solution from the first phase was, in fact, optimal. If not, we must go back and price out the (reduced) set of extra variables in each leaf of the search tree produced during the first phase. We then continue processing any node in which we fail to price out all the variables.

In order to avoid pricing variables in every leaf of the tree, we can *trim the tree* before the start of the second phase. Trimming the tree consists of eliminating the children of any node for which each child has lower bound above the current upper bound. We then reprocess the parent node itself. This is typically more efficient, since there is a high probability that, given the new upper bound and cuts, we will be able to prune the parent node and avoid the task of processing each child individually.

4.2.4 The Cut Generator Module

To implement the cut generator process, the user must provide a function that accepts an LP solution and returns cuts violated by that solution to the LP module. In parallel configurations, each cut is returned immediately to the LP module, rather than being passed back as a group once the function exits. This allows the LP to begin adding cuts and solving the current relaxation before the cut generator is finished if desired. Parameters controlling if and when the LP should begin solving the relaxation before the cut generator is finished can be set by the user.

4.2.5 The Cut Pool Module

Maintaining and Scanning the Pool

The cut pool's primary job is to receive a solution from an LP module and return cuts from the pool that are violated by it. The cuts are stored along with two pieces of information—the level of the tree on which the cut was generated, known simply as the *level* of the cut, and the number of times it has been checked for violation since the last time it was actually found to be violated, known as the number of *touches*. The number of touches can be used as a simplistic measure of its effectiveness. Since the pool can get quite large, the user can choose to scan only cuts whose number of touches is below a specified threshold and/or cuts that were generated on a level at or above the current one in the tree. The idea behind this second criterion is to try to avoid checking cuts that were not generated "nearby" in the tree, as they are less likely to be effective. Any cut generated at a level in the tree below the level of the current node must have been generated in a different part of the tree. Although this is admittedly a naive method, it does seem to work reasonably well.

On the other hand, the user may define a specific measure of quality for each cut to be used instead. For example, the degree of violation is an obvious candidate. This measure of quality must be computed by the user, since the cut pool module has no knowledge of the cut data structures. The quality is recomputed every time the user checks the cut for violation and a running average is used as the global quality measure. The cuts in the pool are periodically sorted by this measure and only the highest quality cuts are checked each time. All duplicate cuts, as well as all cuts whose number of touches exceeds or whose quality falls below specified thresholds, are periodically purged from the pool to keep it as small as possible.

Using Multiple Pools

For several reasons, it may be desirable to have multiple cut pools. When there are multiple cut pools, each pool is initially assigned to a particular node in the search tree. After being assigned to that node, the pool services requests for cuts from that node and all of its descendants until such time as one of its descendants gets assigned to another cut pool. After that, it continues to serve all the descendants of its assigned node that are not assigned to other cut pools.

Initially, the first cut pool is assigned to the root node. All other cut pools are unassigned. During execution, when a new node is sent to be processed, the tree manager must determine which cut pool the node should be serviced by. The default is to use the same cut pool as its parent. However, if there is currently an idle cut pool process (either it has never been assigned to any node or all the descendants of its assigned node have been processed or reassigned), then that cut pool is assigned to this new node. All the cuts currently in the cut pool of its parent node are copied to the new pool to initialize it, after which the two pools operate independently on their respective subtrees. When generating cuts, the LP process sends the new cuts to the cut pool assigned to service the node during whose processing the cuts were generated.

The primary motivation behind the idea of multiple cut pools is two-fold. First, we want simply to limit the size of each pool as much as possible. By limiting the number of nodes that a cut pool has to service, the number of cuts in the pool will be similarly limited. This not only allows cut storage to spread over multiple processors, and hence increases the available memory, but at the same time, the efficiency with which the cut pool can be scanned for violated cuts is also increased. A secondary reason for maintaining multiple cut pools is that it allows us to limit the scanning of cuts to only those that were generated in the same subtree as the current search node. As described above, this helps focus the search and should increase the efficiency and effectiveness of the search. This idea also allows us to generate locally valid cuts, such as the classical Gomory cuts (see [30]).

4.3 Parallelizing BCP

Because of the clear partitioning of work that occurs when the branching operation generates new subproblems, branch and bound algorithms lend themselves well to parallelization. As a result, there is already a significant body of research on performing branch and bound in parallel environments. We again point the reader to the survey of parallel branch and bound algorithms by Gendron and Crainic [17], as well as other references such as [11, 19, 34, 24].

In parallel BCP, as in general branch and bound, there are two major sources of parallelism. First, it is clear that any number of subproblems on the current candidate list can be processed simultaneously. Once a subproblem has been added to the list, it can be properly processed before, during, or after the processing of any other subproblem. This is not to say that processing a particular node at a different point in the algorithm won't produce different results—it most certainly will—but the algorithm will terminate correctly in any case. The second major source of parallelism is to parallelize the processing of individual subproblems. By allowing separation to be performed in parallel with the solution of the linear programs, we can theoretically process a node in little more than the amount of time it takes to solve the sequence of LP relaxations. Both of these sources of parallelism can be easily exploited using the SYMPHONY framework.

The most straightforward parallel implementation, which is the one we currently employ, is a master-slave model, in which there is a central manager responsible for partitioning the work and parceling it out to the various slave processes that perform the actual computation. The reason we chose this approach is because it allows memory-efficient data structures for sequential computation and yet is conceptually easy to parallelize. Unfortunately, this approach does have limited scalability. For further discussions on the scalability of BCP algorithms and approaches to improving it, see [32] and [38].

4.3.1 Parallel Configurations

SYMPHONY supports numerous configurations, ranging from completely sequential to fully parallel, allowing efficient execution in many different computational settings. As described in the previous section, there are five modules in the standard distributed configuration. Various subsets of these modules can be combined to form separate executables capable of communicating with each other across a network. When two or more modules are combined, they simply communicate through shared-memory instead of through message-passing. However, they are also forced to run in sequential fashion in this case, unless the user chooses to enable threading using an OpenMP compliant compiler (see next section).

As an example, the default distributed configuration includes a separate executable for each module type, allowing full parallelism. However, if cut generation is fast and not memory-intensive, it may not be worthwhile to have the LP solver and its associated cut generator work independently, as this increases communication overhead without much potential benefit. In this case, the cut generator functions can be called directly from the LP solver, creating a single, more efficient executable.

4.3.2 Inter-process Communication

SYMPHONY can utilize any third-party communication protocol supporting basic message-passing functions. All communication subroutines interface with SYMPHONY through a separate communications API. Currently, PVM [16] is the only message-passing protocol supported, but interfacing with another protocol is a straightforward exercise.

Additionally, it is possible to configure the code to run in parallel using threading to process multiple search tree nodes simultaneously. Currently, this is implemented using OpenMP compiler directives to specify the parallel regions of the code and perform memory locking functions. Compiling the code with an OpenMP compliant compiler will result in a shared-memory parallel executable. For a list of OpenMP compliant compilers and other resources, visit `http://www.openmp.org`.

4.3.3 Fault Tolerance

Fault tolerance is an important consideration for solving large problems on computing networks whose nodes may fail unpredictably. The tree manager tracks the status of all processes and can restart them as necessary. Since the state of the entire tree is known at all times, the most that will be lost if an LP process or cut generator process is killed is the work that had been completed on that particular search node. To protect against the tree manager itself or a cut pool being killed, full logging capabilities have been implemented. If desired, the tree manager can write out the entire state of the tree to disk periodically, allowing a warm restart if a fault occurs. Similarly, the cut pool process can be warm-started from a log file. This not only allows for fault tolerance but also for full reconfiguration in the middle of solving a long-running problem. Such reconfiguration could consist of anything from adding more processors to moving the entire solution process to another network.

Chapter 5

Installation

SYMPHONY Version 5.0 is a powerful environment for implementing custom branch, cut, and price algorithms. The subroutines in the SYMPHONY library comprise a state-of-the-art MILP solver designed to be modular and easy to customize for various problem settings. All internal library subroutines are generic—their implementation does not depend on the the problem-setting. As from Version 4.0, SYMPHONY works out of the box as a generic MILP solver, with the capability to read both MPS files and GMPL (a subset of AMPL) files and solve the described mixed integer programs. To customize SYMPHONY, various user subroutines can be written and parameters set that modify the default behavior of the algorithm. The API for these subroutines is described in this manual and files containing function stubs are provided. As an example, by replacing the default I/O subroutine, one can easily modify the solver so that it reads in problem instances in a custom format (such as the TSPLIB format for specifying traveling salesman problem instances).

The vast majority of the computation takes place within a "black box," of which the user need have no knowledge. SYMPHONY performs all the normal functions of branch, cut, and price—tree management, LP solution, cut pool management, as well as inter-process or inter-thread communication. Solvers can be built in a wide variety of configurations, ranging from fully parallel to completely sequential, depending on the user's needs. The library runs serially on almost any platform, and can also run in parallel in either a fully distributed environment (network of workstations) or shared-memory environment simply by changing a few options in the makefile. To run in a distributed environment, the user must have installed the ***Parallel Virtual Machine*** (PVM), available for free from Oak Ridge National Laboratories. To run in a shared memory environment, the user must have installed an OpenMP compliant compiler. A cross-platform compiler called ***Omni***, which uses `cc` or `gcc` as a back end, is available for free download at `http://phase.etl.go.jp/Omni/` . For other options, visit `http://www.openmp.org`.

SYMPHONY-5.0 is now a C callable library with an interface whose look and feel is similar to other popular solvers, see Sections 7.1 and 7.2 for the library routines. This interface works for SYMPHONY's built-in generic MILP solver, as well as any customized algorithm developed by implementing one or more of SYMPHONY's user callback functions. For a summary of what else is new, see Section 1.1.Code written for previous versions of SYMPHONY will be broken, but not too badly. Instructions for porting from previous version are contained in the file `SYMPHONY-5.0/README-5.0`.

This section of the manual is concerned with the detailed specifications needed to compile the SYMPHONYlibrary, to create the generic MILP solver and to develop an application using SYMPHONY. It is assumed that the user has already read the first part of the manual, which provides a high-level introduction to parallel branch, cut, and price and the overall design and use of SYMPHONY.

5.1 Compiling the Library and Executable in Unix

Here is a sketch outline of how to get started with SYMPHONY in Unix. This is basically the same information contained in the README file that comes with the distribution and will lead you through the steps required to compile SYMPHONY as a generic MILP solver that can then be customized by filling out the functions provided in the user interface files. For more information, see Section 6.7.

Because SYMPHONY is intended to run over nonhomogeneous networks of workstations, installation is not fully automated, but requires the user to make minor edits to the makefile. With this setup, compilation for multiple architectures and configurations can be performed in a single directory without reconfiguring or "cleaning." This is convenient on nonhomogeneous networks, but it means that you might need to edit the makefiles to get SYMPHONY to compile. For the casual user, this editing is limited to providing some path names.

5.1.1 Preparing for Sample Compilation.

- Download the file `SYMPHONY-5.0.tgz`.
- Unpack the distribution with `tar -xzf SYMPHONY-5.0.tgz`. This will create a subdirectory called `SYMPHONY-5.0` containing the distribution.
- Edit the makefile (`SYMPHONY-5.0/Makefile`) to reflect your environment. This involves specifying the LP solver to be used, assigning some variables and setting the paths to various libraries and include files.Only minor edits should be required. An explanation of what has to be set is contained in the comments in the makefile.
- To use many of the new capabilities of SYMPHONY, you must have installed the COIN optimization libraries COIN optimization libraries, available from `http://www.coin-or.org` . By default, SYMPHONY is set up to use COIN LP solver, CLP, COIN Open Solver Interface, OSI, and COIN Cut Generation Library, CGL. To keep this configuration, you should install OSI, CGL, CLP and the Coin utilities (under `COIN/Coin`). The path to the COIN libraries must be specified in `SYMPHONY-5.0/Makefile`. If you want to use the new OSI interface to SYMPHONY, you should be sure to compile it when you are installing the rest of the COIN packages.
- If you wish to read `GMPL/AMPL` files, you will have to install the Gnu Linear Programming Kit (GLPK), which contains a parser for GMPL/AMPL files. The path to the GLPK libraries must also be specified in `SYMPHONY-5.0/Makefile`.

5.1.2 Compiling the Sequential Version.

- Unlike previous version of SYMPHONY, to compile `SYMPHONY 5.0` as a generic solver, the user simply has to type `make` in the `SYMPHONY-5.0` subdirectory. This will first make the SYMPHONY library (sequential version): `SYMPHONY-5.0/lib.$(ARCH)/$(LP_SOLVER)/libsym.so` (or `'libsym.a'` if library type is set to be static) where `ARCH` is the current architecture and `LP_SOLVER` is the current LP solver, as specified in the makefile. In addition, in order to have the flexibility in using different LP solvers, a symbolic link to the latest created callable library with the same name (libsym.so or libsym.a) will be created in `SYMPHONY-5.0/lib` subdirectory. This library together with the header files in the subdirectory `SYMPHONY-5.0/include` can then be used to call SYMPHONY from any C code. The `API` for this is described in section 7.3. After compiling the SYMPHONY library, the default `main` function will be compiled and linked with the the callable library to form an executable called `symphony` to be used for solving generic MILP problems in `MPS` or `GMPL` format. `FlopC++` [22] can also be used to obtain a capability similar to `ILOG`'s Concert technology for building math programming models. The executable is installed in `SYMPHONY-5.0/bin.$(ARCH)/$(LP_SOLVER)` subdirectory. The makefile can also be modified to enable parallel execution of the code (see below).

- After the SYMPHONY library is compiled, you are free to type `make clean` if you want to save disk space. You should only have to remake the library if you change something in SYMPHONY's internal files.

- To test SYMPHONY, a sample MPS file called `sample.mps` is included with the distribution. To specify the file name, use the `-F` command-line option, i.e., type

  ```
  bin.$(ARCH)/$(LP_SOLVER)/symphony -F sample.mps
  ```

  in the `SYMPHONY-5.0` subdirectory. To obtain more MPS data files for further testing, download the `MIPLIB` library.

- That's it! Now you are ready to use SYMPHONY callable library or solve generic MILP problems through the executable.

5.1.3 Compiling the Shared Memory Version.

Please note that the shared-memory parallel version has not been tested in Version 5.0 and may be broken. Please let me know if you want to use it and I will get it working.

- To compile a shared memory version, obtain an OpenMP compliant compiler, such as Omni (free from `http://phase.etl.go.jp/Omni`) . Other options are listed at the OpenMP Web site (`http://www.openmp.org`) .

- Follow the instructions above for configuring the makefile. Set the variable `CC` to the compiler name in the makefile and compile as above. Note that if you have previously compiled the sequential version, then you should first type `make clean_all`, as this version uses the same directories. With one thread allowed, it should run exactly the same as the sequential version so there is no need to compile both versions.

- Voila, you have a shared memory parallel solver. As above, to test SYMPHONY, a sample MPS file called `sample.mps` is included with the distribution. To specify the file name, use the `-F` command-line option, i.e., type `bin.$(ARCH)/$(LP_SOLVER)/symphony -F sample.mps` in the `SYMPHONY-5.0` subdirectory. To obtain more MPS data files for further testing, download the `MIPLIB` library.

- That's it! Now, you are ready to develop your own application using SYMPHONY callable library or solve MILP problems using the executable. See the user manual for help.

5.1.4 Compiling the Distributed Version.

Please note that the distributed-memory parallel version has not been tested in Version 5.0 and may be broken. Please let me know if you want to use it and I will get it working.

- If you wish to compile a distributed version of the code, obtain and install the *Parallel Virtual Machine* (PVM) software, available for free from Oak Ridge National Laboratories at `http://www.ccs.ornl.gov/pvm/` . See Section 6.7.1 for more notes on using PVM.

- In `SYMPHONY-5.0/Makefile`, be sure to set the `COMM_PROTOCOL` to `PVM`. Also, in the same makefile, you need to change one or more of `SYM_COMPILE_IN_TM`, `SYM_COMPILE_IN_LP`, `SYM_COMPILE_IN_CG`, and `SYM_COMPILE_IN_CP` to `FALSE` or you will end up with the sequential version. Various combinations of these variables will give you different configurations and different executables. See Section 6.7.1 for more info on setting them. Also, be sure to set the path variables in the makefile appropriately so that `make` can find the PVM library.

- As above, type `make` in the `SYMPHPONY-5.0` subdirectory to make the distributed libraries. As in Step 1 of the sequential version, you may type `make clean` after making the library. It should not have to remade again unless you modify SYMPHONY's internal files.

- After the SYMPHONY libraries, main function will be compiled and required executables linked.

- Make sure there are links from your `$(PVM_ROOT)/bin/$(PVM_ARCH)` subdirectory to each of the executables in the `SYMPHONY-5.0/bin.$(ARCH)/$(LP_SOLVER)` subdirectory. This is required by PVM.

- Start the PVM daemon by typing `pvm` on the command line and then typing `quit`.

- As above, test SYMPHONY using the sample MPS file called `sample.mps` included with the distribution. To specify the file name, use the `-F` command-line option, i.e., type `bin.$(ARCH)/$(LP_SOLVER)/symphony -F sample.mps` in the `SYMPHONY-5.0` subdirectory. To obtain more MPS data files for further testing, download the MIPLIB library.

- That's it! Now, you are ready to develop your own application using SYMPHONY callable library or solve MILP problems using the executable.

5.2 Compiling the Library and Executable in Windows

Here is a sketch outline of how to compile SYMPHONY in MS Windows. Direct support is provided for compilation with MS Visual Studio 6.0. Compilation for other compilers should also be possible. Note that the Windows version has some limitations. Detailed timing information is not currently provided. Support is only provided for running in sequential mode at this time.

First, download `SYMPHONY-5.0.zip` and unzip the archive. This will create a subdirectory called `SYMPHONY-5.0` containing all the source files. You now have two options. You can either compile on the command-line, using the MSVC++ makefile called `sym.mak` in the `SYMPHONY-5.0\WIN32` subdirectory or you can use the provided projects and workspaces. Compiling on the command-line is somewhat easier since it requires only editing the makefile and typing a single command.

5.2.1 Using the NMAKE Utility

- Edit the file `SYMPHONY-5.0\WIN32\sym.mak` makefile to reflect your environment. This involves specifying the LP solver to be used and various paths. Only minor edits should be required. An explanation of what has to be set is contained in the comments in the makefile.

- To use many of the new capabilities of SYMPHONY, you must have installed the *COIN* optimization libraries COIN optimization libraries, available from `http://www.coin-or.org` . By default SYMPHONY is set to use COIN LP solver, `CLP`, COIN Open Solver Interface, `OSI`, and COIN Cut Generation Library, `CGL`. To keep this configuration, you should install `OSI`, `CGL`, `CLP` and additionally, the Coin utilities (under COIN\Coin).The path to the COIN libraries must be specified in `SYMPHONY-5.0\WIN32\sym.mak`.

- If you wish to read `GMPL/AMPL` files, you will have to install the Gnu Linear Programming Kit (GLPK), which contains a parser for GMPL/AMPL files. The path to the GLPK libraries must be specified in the makefile.

- Once configuration is done, type `nmake /f sym.mak` at the command prompt in the `SYMPHONY-5.0\WIN32` subdirectory. This will first make the SYMPHONY library (sequential version): `SYMPHONY-5.0\WIN32\Debug\symphonyLib.lib`. This library together with the header files in the subdirectory (`SYMPHONY-5.0\include`) can then be be used to call SYMPHONY from any C code. The `API` for this is described in section 7.3. After compiling the SYMPHONY library, the default `main` function will be compiled and linked with the the callable library to form an executable called `symphony.exe` to be used for solving generic MILP problems in `MPS` or `GMPL` format. The executable will be created in the `SYMPHONY-5.0\WIN32\Debug` subdirectory.

- To test the executable, type `symphony.exe -F ..\..\sample.mps` at a command prompt in the `SYMPHONY-5.0\WIN32\Debug` subdirectory.

5.2.2 Using the MSVC++ Workspace

- In MS Visual C++ 6.0, open the workspace `SYMPHONY-5.0\WIN32\symphony.dsw`. Note that there are two projects, one called `symphony` and the other called `symphonyLib`. The `symphonyLib` project compiles the source code to create the callable library: `symphonyLib.lib`. The `symphony` project compiles the main function and links that with the c allable library to create the executable: `symphony.exe`.

- To use many of the new capabilities of SYMPHONY, you must have installed the COIN optimization libraries COIN optimization libraries, available from `http://www.coin-or.org` . By default SYMPHONY is set up to use COIN LP solver, `CLP`, COIN Open Solver Interface, `OSI`, and COIN Cut Generation Library, `CGL`. To keep this configuration, you should install `OSI`, `CGL`, `CLP` and additionally, the Coin utilities (under COIN\Coin). The default location for COIN is C:\COIN\.

- By default, SYMPHONY is set up to use the OSI CLP interface. To see this check the following settings:

  - `__OSI_CLP__` is defined in the preprocessor definitions of both `symphony` and `symphonyLib` projects (right-click on one of the projects, and then choose `Settings --> C/C++ --> Preprocessor` in the category drop-down menu).

- Paths to the include files of COIN utilities (Coin), OSI, OSI_CLP, and CLP are specified in the same settings window as for the preprocessor definitions. Note that the Coin, OSI, and OSI_CLP and CLP include directories are assumed to be in `C:\COIN\Coin`, `C:\COIN\Osi`, `C:\COIN\Osi\OsiClp` and `C:\COIN\Clp`, directories, respectively. If they are not, make sure that you have set the correct paths in both projects before compiling.
- The `symphony` project is dependent on the `symphonyLib` project (see the dependencies in `Project --> Dependencies`) and it includes the necessary libraries: `symphonyLib`, `coinLib`, `osiLib`, `osiClpLib` and `clpLib` (solver library).

If you want to use the native CPLEX or OSL interface (without downloading COIN) or a solver other than CLP:

- If another OSI interface, change the preprocessor definition in both projects from `__OSI_CLP__` to `__OSI_XXX__` , where `XXX` is replaced by the desired solver's acronym (e.g., `__OSI_CPLEX__`, `__OSI_GLPK__`, `__OSI_OSL__`, etc.). Otherwise, change it to either `__CPLEX__` or `__OSL__` in both projects.
- Change the path definitions of the include files: for instance, if you want to use `__OSI_CPLEX__`, define `C:\COIN\Osi\OsiCpx` and `C:\ILOG\cplex81\include\ilcplex` (assuming it is installed there) instead of the OSI CLP and CLP path definitions. Or, if you want to use `__OSI_OSL__`, define `C:\COIN\Osi\OsiOsl` and `C:\ProgramFiles\IbmOslV3Lib\osllib` (assuming OSL is installed there) instead of the OSI CLP and CLP path definitions. If you want to use the native CPLEX or OSL interface, delete all the path definitions (you are not required to have COIN or OSI), and just add the path definitions for the CPLEX or OSL include files.
- Add the appropriate libraries to the `symphony` project. For instance, if you want to use `__OSI_CPLEX__`, then add the `osiCpxLib` and `cplex81` library files after deleting `osiClpLib` and `clpLib` libraries from the `symphony` project. If you want to use the native CPLEX interface, then delete all the libraries (except the `symphonyLib`) and just add the `cplex81` library file for it is the unique solver library file we need now.

- By default, SYMPHONY is also set up to use the `COIN CGL` library for generating cuts. To use CGL, the `symphonyLib` project has the `ADD_CGL_CUTS` preprocessor definition, the path to `C:\COIN\Cgl\` (be sure that this path directs SYMPHONY to the include subdirectory of CGL). If you don't want to use the CGL library, simply delete the `ADD_CGL_CUTS` preprocessor definition, the CGL path definitions and the `cgllib` library from the `symphony` project.

- DO NOT CHANGE COMPILER DEFINES NOT RELATED TO THE LP SOLVER. Important note for OSL users: when using OSL in Windows, you must also add `OSLMSDLL` to the list of definitions.

- Note that there are a number of additional preprocessor definitions that control the functionality of SYMPHONY. These definitions are described in `SYMPHONY-5.0/Makefile`, a Unix-style makefile included with the distribution. To enable the functionality associated with a particular definition, simply add it to the list of definitions, as above.

- You must also be sure to have any `.dll` files required for your LP solver to be in your search path. Either move the required `.dll` to the subdirectory containing `symphony.exe` or add the path to the `PATH` Windows environment variable.

- Once you have the proper settings for your LP solver, choose `Build symphony.exe` from the `Build menu`. This should successfully build the SYMPHONY library and the executable.
- To test the executable, right click on the `symphony` project, go to the `Debug` tab and set the program arguments to `-F ..\sample.mps` Note that command-line switches are Unix-style.
- Now choose `Execute` from the build menu and the solver should solve the sample problem.

Note that there is some functionality missing from the Windows version. Most prominently, the timing functions do not work. This functionality should be easy to add—let me know if you are interested in doing it and I will give you all the help I can. In addition, the Windows version will only run in sequential mode for a variety of reasons. However, it should be relatively easy to get it running in parallel if you can get PVM working under Windows. Let me know if you are interested.

5.3 Compiling a Custom Application Using Callbacks

5.3.1 Unix

First, configure and compile SYMPHONY 5.0 as described in SYMPHONY-5.0/README-5.0. Modify the variables in the USER/Makefile appropriately. Typing "make" in the USER subdirectory should successfully make the USER executable. It will be installed in the directory SYMPHONY-5.0/USER/bin.($ARCH$)/(LP_SOLVER). After you've successfully compiled the code, you can develop our custom application by following the instructions for filling in the user callback functions as described in Section 6.

5.3.2 Microsoft Windows

First, download `SYMPHONY-5.0.zip` and unzip the archive. This will create a subdirectory called `SYMPHONY-5.0` containing all the source files together with the `USER` subdirectory. There are two options to get the executable. You can either compile on the command-line, using the MSVC++ makefile, `USER\WIN32\user.mak`, or you can use the provided projects and workspaces. However for the second option, it is important the `USER` archive be kept in the `SYMPHONY-5.0` subdirectory or the project files will not work. Compiling on the command-line is somewhat easier since it requires only editing the makefile and typing a single command.

5.3.3 Using the NMAKE Utility

- Edit the `USER\WIN32\user.mak` makefile to reflect your environment. Only minor edits should be required. An explanation of what has to be set is contained in the comments in the makefile. This basically requires the same routines that one needs to walk through in `SYMPHONY`'s makefile. See the related parts of 5.2.1 section of `SYMPHONY` above.
- Once configuration is done, type `nmake /f user.mak` in the `USER\WIN32` subdirectory. The executable `user.exe` will be created under the `USER\WIN32\Debug` directory.
- To test the executable, type `user.exe -F ..\..\sample.mps` at a command prompt from the `USER\WIN32\Debug` directory. After this point, you will be ready to develop your own application by modifying the other files in the `USER` subdirectory.

5.3.4 Using the MSVC++ Workspace

- In MS Visual C++ 6.0, open the workspace `SYMPHONY-5.0\USER\WIN32\user.dsw`. Note that there are two projects, one called `symphonyLib` and the other called `user`.The `symphonyLib` project compiles the source code, with the calls to the user-defined callbacks used to customize the solver, to create the callable library: `symphonyLib.lib`. The `user` project compiles those user callbacks together with the main function, links them with the callable library and creates the executable: `user.exe`.

- The configuration steps are exactly the same with the MSVC++ section of `SYMPHONY`. The only difference is that, you have the `user` project instead of the `symphony` project. Go through the related steps of section 5.2 to see how to configure to use `COIN`, `OSI`, `CGL`, `COIN` utilities, `GMPL` input, and to change the lp solver which is by default `CLP`.

- Once you have the proper settings for your LP solver, choose `Build user.exe` from the `Build` menu. This should successfully build the executable.

- To test the executable, right click on the `user` project, go to the `Debug` tab and set the program arguments to `-F ..\sample.mps`. Note that command-line switches are Unix-style.

- Now choose `Execute` from the build menu and you have a working branch and bound solver! After successful compilation, you can fill in the user callback functions as describe in SectionSYMPHONY-development.

5.4 Sample Applications

There are a number of sample applications available as examples of how to do development with SYMPHONY. These include solvers for the matching problem, the set partitioning problem (simple and advanced versions), the vehicle routing and traveling salesman problems, the mixed postman problem and, capacitated network routing problem. These applications are distributed as separate packages and can be downloaded from `http://www.branchandcut.org/SYMPHONY`. There is also a white paper that guides the user through the development of the matching solver.

Chapter 6

Development

6.1 Orienting Yourself

The easiest way to get oriented is to examine the organization of the source files (note that file names will be given Unix-style). When you unpack the SYMPHONY distribution, you will notice that the source files are organized along the lines of the modules. There is a separate directory for each module—master (`Master`), tree manager (`TreeManager`), cut generator (`CutGen`), cut pool (`CutPool`), and LP solver (`LP`). In addition, there is a directory called `DrawGraph` and a directory called `Common` that also contain source files. The `DrawGraph` directory provides an interface from SYMPHONY to the *Interactive Graph Drawing* software package developed by Marta Esö. This is an excellent utility for graphical display and debugging. The `Common` directory contains source code for functions used by multiple modules.

Within each module's directory, there is a primary source file containing the function `main()` (named `*.c` where * is the module name), a source file containing functions related to inter-process communication (named `*_proccomm.c`) and a file containing general subroutines used by the module (named `*_func.c`). The master is the exception and is organized slightly differently. The LP process source code is further subdivided due to the sheer number of functions.

The `include` directory contains the header files. Corresponding to each module, there are three header files, one containing internal data structures and function prototypes associated with the module (named `*.h` where * is the module name), one containing the data structures for storing the parameters (these are also used by the master process), and the third containing the function prototypes for the user callbacks (name `*_u.h`). By looking at the header files, you should get a general idea of how things are laid out.

In addition to the subdirectories corresponding to each module, there is a subdirectory called `SYMPHONY-5.0/USER`, which contains the files needed for implementing the callbacks. Before beginning customization, it is recommended to make a copy of the directory `SYMPHONY-5.0/USER` that will be used as a template for creating your customized solver. In this directory and its subdirectories, which mirror the subdirectories of SYMPHONY itself, each file contains function stubs that can be filled in to create a new custom application. There is one file for each module, initially called `SYMPHONY-5.0/USER/*/user_*.c`, where `*` is the name of the module. The primary thing that you, as the user, need to understand to build a custom application is how to fill in these stubs. That is what the second section of this manual is about.

6.2 Writing the Callbacks

For each module, all callback functions are invoked from so-called *wrapper functions* that provide the interface and also performs a default action if the user chooses not to override it. Although SYMPHONY is written in C, the wrapper functions provide a C++-style interface in which the user can either accept the default action or override it. Each wrapper function is named `*_u()` , where `*` is the name of the corresponding callback function, and is defined in a file called `*_wrapper.c`. The wrapper function first collects the necessary data and hands it to the user by calling the user function. Based on the return value from the user, the wrapper then performs any necessary post-processing. All callback functions have default options, so that SYMPHONY now acts as a generic MILP solver out of the box.

In Section 7.3, the callback functions are described in detail. The name of every callback function starts with `user_`. There are three kinds of arguments:

IN: An argument containing information that the user might need to perform the function.

OUT: A pointer to an argument in which the user should return a result (requested data, decision, etc.) of the function.

INOUT: An argument which contains information the user might need, but also for which the user can change the value.

The return values for most function are as follows:

Return values:

`USER_ERROR`	Error in the user function. Printing an error message is the user's responsibility. Depending on the work the user function was supposed to do, the error might be ignored (and some default option used), or the process aborts.
`USER_SUCCESS`	The user function was implemented and executed correctly.
`USER_DEFAULT`	This option means that the user function was not implemented and that SYMPHONY should either execute a default subroutine (the default is one of the built-in options, SYMPHONY decides which one to use based on initial parameter settings and the execution of the algorithm) or else do nothing, if execution of the subroutine is optional.
`built_in_option1`	
`built_in_option2` ...	The specified built-in option will be used.

Notes:

- Sometimes an output is optional. This is always noted in the function descriptions.
- If an array has to be returned (i.e., the argument is `type **array`) then (unless otherwise noted) the user has to allocate space for the array itself and set `*array` to be the array allocated. If an output array is optional and the user is not returning any values in that array, then the user *must not* set `*array` because this is how SYMPHONY decides which optional arrays are filled up.
- Some built-in options are implemented so that the user can invoke them directly from the callback function. This might be useful if, for example, the user wants to use different built-in options at different stages of the algorithm.

6.3 Data Structures

6.3.1 Internal Data Structures

With few exceptions, the data structures used internally by SYMPHONY are undocumented and most users will not need to access them directly. However, if such access is desired, a pointer to the main data structure used by each of the modules can be obtained simply by calling the function `get_*_ptr()` where * is the appropriate module (see the header files). This function will return a pointer to the data structure for the appropriate module. Casual users are advised against modifying SYMPHONY's internal data structures directly.

6.3.2 User-defined Data Structures

The user can define her own data structure for each module to maintain problem data and any other information the user needs access to in order to implement functions to customize the solver. A pointer to this data structure is maintained by SYMPHONY and is passed to the user as an argument to each user function. Since SYMPHONY knows nothing about this data structure, it is up to the user to allocate it and maintain it. The user must also implement a function to free it. The functions for freeing the user data structures in each module are called `user_free_*`, where `*` is the module. These functions are called by SYMPHONY at the time when other data structures for the modules are being freed and the module is being closed. By default, for sequential computation, there is one common user data structure for all modules and the pointer to that data structure is passed to all user functions, regardless of the module. This setup should work fine for most sequential applications. In parallel, however, pointers cannot be shared between modules and data must be explicitly passed. IN this case, it is sometimes more efficient to maintain in each module only the data necessary to perform the functions of that module.

6.4 Inter-process Communication for Distributed Computing

While the implementation of SYMPHONY strives to shield the user from having to know anything about communications protocols or the specifics of inter-process communication, it may be necessary for the user to pass information from one module to another in order to implement a parallel application. For instance, the user may want to pass data describing the problem instance to the LP process after reading them in from a file in the master process. For the purpose of passing user data from the master process to other processes, a customization function called `user_send_*_data()` is provided in the master module, along with a corresponding function called `user_receive_*_data()` in the module `*`. These two functions work in tandem to transport the user's data from the maser, where it can be read in from a file, to the proper module for processing. There are also a number of other tandem pairs of *send* and *receive* functions that are used to transport user data from place to place.

All data are sent in the form of arrays of either type `char`, `int`, or `double`, or as strings. To send an array, the user has simply to invoke the function `send_XXX_array(XXX *array, int length)` where `XXX` is one of the previously listed types. To receive that array, there is a corresponding function called `receive_?_array(?  *array, int length)`. When receiving an array, the user must first allocate the appropriate amount of memory. In cases where variable length arrays need to be passed, the user must first pass the length of the array (as a separate array of length one) and then the array itself. In the receive function, this allows the length to be received first so that the proper amount of space can be allocated before receiving the array itself. Note that data must be received in exactly the same order as it was passed, as data is read linearly into and out of the message buffer. The easiest way to ensure this is done properly is to simply copy the send statements into the receive function and change the function names. It may then be necessary to add some allocation statements in between the receive function calls.

6.5 The LP Engine

SYMPHONY requires the use of a third-party callable library to solve the LP relaxations once they are formulated. Native interfaces to ILOG's CPLEX c and IBM's OSL are available. As of Version 4.0, the Open Solver Interface, available from COIN (`http://www.coin-or.org`) . can be used to interface with most commonly available LP solvers. The list of solvers with OSI interfaces currently numbers eight and includes both commercial and open source alternatives. If the COIN libraries are used, make sure to set the proper paths in the SYMPHONY makefile.

6.6 Cut Generation

SYMPHONY now generates generic cutting planes using the Cut Generator Library, also available from COIN COIN (`http://www.coin-or.org`) . The CGL can be used to generate cuts in cases where problem-specific cutting planes are not available or not implemented yet.

6.7 Advanced Compilation

6.7.1 Unix Operating Systems

Once the callback functions are filled in, all that remains is to compile the application. The distribution comes with two makefiles that facilitate this process. The primary makefile resides in the `SYMPHONY-5.0/` directory. The user makefile resides in the user's subdirectory, initially called `SYMPHONY-5.0/USER/`. This subdirectory can be moved, as well as renamed. There are a number of variables that must be set in the primary make file. To modify the makefiles appropriately, see the instructions in Section 5.1.

Working with PVM. To compile a distributed application, it is necessary to install PVM. The current version of PVM can be obtained at `http://www.csm.ornl.gov/pvm/`. It should compile and install without any problem. You will have to make a few modifications to your `.cshrc` file, such as defining the `PVM_ROOT` environment variable, but this is all explained clearly in the PVM documentation. Note that all executables (or at least a link to them) must reside in the `$PVM_ROOT/bin/$PVM_ARCH` directory in order for parallel processes to be spawned correctly. The environment variable `PVM_ARCH` is set in your `.cshrc` file and contains a string representing the current architecture type. To run a parallel application, you must first start up the daemon on each of the machines you plan to use in the computation. How to do this is also explained in the PVM documentation.

Communication with Shared Memory. In the shared memory configuration, it is not necessary to use message passing to move information from one module to another since memory is globally accessible. In the few cases where the user would ordinarily have to pass information using message passing, it is easiest and most efficient to simply copy the information to the new location.

This copying gets done in the *send* function and hence the *receive* function is never actually called. This means that the user must perform all necessary initialization, etc. in the send function. This makes it a little confusing to write source code which will work for all configurations. However, the confusion should be minimized by looking at the sample applications, especially the VRP solver, which works in all configurations, sequential, distributed parallel, and shared parallel.

Configuring the Modules. In the application makefile, e.g., `SYMPHONY-5.0/USER/Makefile`,there are four variables that control which modules run as separate executables and which are called directly in serial fashion. The variables are as follows:

COMPILE_IN_CG: If set to `TRUE`, then the cut generator function will be called directly from the LP in serial fashion, instead of running as a separate executable. This is desirable if cut generation is quick and running it in parallel is not worth the price of the communication overhead.

COMPILE_IN_CP: If set to `TRUE`, then the cut pool(s) will be maintained as a data structure auxiliary to the tree manager.

COMPILE_IN_LP: If set to `TRUE`, then the LP functions will be called directly from the tree manager. When running the distributed version, this necessarily implies that there will only be one active subproblem at a time, and hence the code will essentially be running serially. IN the shared-memory version, however, the tree manager will be threaded in order to execute subproblems in parallel.

COMPILE_IN_TM: If set to `TRUE`, then the tree will be managed directly from the master process. This is only recommended if a single executable is desired (i.e. the three other variables are also set to true). A single executable is extremely useful for debugging purposes.

These variables can be set in virtually any combination, though some don't really make much sense. Note that in a few user functions that involve process communication, there will be different versions for serial and parallel computation. This is accomplished through the use of `#ifdef` statements in the source code. This is well documented in the function descriptions and the in the source files containing the function stubs. See also Section 6.7.1.

Executable Names. In order to keep track of the various possible configurations, executable and their corresponding libraries are named as follows. The name of the master module, along with all other modules compiled in with the master, is set in the makefile. For the other modules, default names are typically used, since these names have to be hard-coded in order for PVM to correctly spawn the corresponding processes. In the fully distributed version, the default names are `tm`, `lp`, `cg`, and `cp`. For other configurations, the executable name is a combination of all the modules that were compiled together joined by underscores. In other words, if the LP and the cut generator modules were compiled together (i.e. `COMPILE_IN_CG` set to `TRUE`), then the executable name would be "`lp_cg`" and the corresponding library file would be called "`liblp_cg.a`." You can rename the executables as you like. However, if you are using PVM to spawn the modules, as in the fully distributed version, you must set the parameters `*_exe` in the parameter file to the new executable names. See Section 7.4.4 for information on setting parameters in the parameter file.

6.7.2 Microsoft Windows

First, follow the instructions for compiling SYMPHONY in Section 5.2 to ensure you have the proper settings. Once the stub files in the `SYMPHONY-5.0\USER` hierarchy are filled in, you should be able to compile the new application and run it successfully.

6.8 Debugging Your Application

Much of this section applies to Unix operating systems. However, it may also be useful for Windows users.

6.8.1 The First Rule

SYMPHONY has many built-in options to make debugging easier. The most important one, however, is the following rule. **It is easier to debug the fully sequential version than the fully distributed version**. Debugging parallel code is not terrible, but it is more difficult to understand what is going on when you have to look at the interaction of several different modules running as separate processes. This means multiple debugging windows which have to be closed and restarted each time the application is re-run. For this reason, it is highly recommended to develop code that can be compiled serially even if you eventually intend to run in a fully distributed environment. This does make the coding marginally more complex, but believe me, it's worth the effort. The vast majority of your code will be the same for either case. Make sure to set the compile flag to "`-g`" in the makefile.

6.8.2 Debugging with PVM

If you wish to venture into debugging your distributed application, then you simply need to set the parameter `*_debug`, where * is the name of the module you wish to debug, to the value "4" in the parameter file (the number "4" is chosen by PVM). This will tell PVM to spawn the particular process or processes in question under a debugger. What PVM actually does in this case is to launch the script `$PVM_ROOT/lib/debugger`. You will undoubtedly want to modify this script to launch your preferred debugger in the manner you deem fit. If you have trouble with this, please send e-mail to the list serve (see Section 6.10).

It's a little tricky to debug interacting parallel processes, but you will quickly get the idea. The main difficulty is in that the order of operations is difficult to control. Random interactions can occur when processes run in parallel due to varying system loads, process priorities, etc. Therefore, it may not always be possible to duplicate errors. To force runs that you should be able to reproduce, make sure the parameter `no_cut_timeout` appears in the parameter file or start SYMPHONY with the "`-a`" option. This will keep the cut generator from timing out, a major source of randomness. Furthermore, run with only one active node allowed at a time (set `max_active_nodes` to "1"). This will keep the tree search from becoming random. These two steps should allow runs to be reproduced. You still have to be careful, but this should make things easier.

6.8.3 Using `Purify` and `Quantify`

The makefile is already set up for compiling applications using `purify` and `quantify`. Simply set the paths to the executables and type "`make pall`" or "`p*`" where * is the module you want to purify. The executable name is the same as described in Section 6.7.1, but with a "p" in front of it.

To tell PVM to launch the purified version of the executables, you must set the parameters `*_exe` in the parameter file to the purified executable names. See Section 7.4.4 for information on setting parameters in the parameter file.

6.8.4 Checking the Validity of Cuts and Tracing the Optimal Path

Sometimes the only evidence of a bug is the fact that the optimal solution to a particular problem is never found. This is usually caused by either (1) adding an invalid cut, or (2) performing an invalid branching. There are two options available for discovering such errors. The first is for checking the validity of added cuts. This checking must, of course, be done by the user, but SYMPHONY can facilitate such checking. To do this, the user must fill in the function `user_check_validity_of_cut()` (see Section 7.3.3). THIS function is called every time a cut is passed from the cut generator to the LP and can function as an independent verifier. To do this, the user must pass (through her own data structures) a known feasible solution. Then for each cut passed into the function, the user can check whether the cut is satisfied by the feasible solution. If not, then there is a problem! Of course, the problem could also be with the checking routine. After filling in this function, the user must recompile everything (including the libraries) after uncommenting the line in the makefile that contains "`BB_DEFINES += -DCHECK_CUT_VALIDITY`." Type "`make clean_all`" and then "`make`."

Tracing the optimal path can alert the user when the subproblem which admits a particular known feasible solution (at least according to the branching restrictions that have been imposed so far) is pruned. This could be due to an invalid branching. Note that this option currently only works for branching on binary variables. To use this facility, the user must fill in the function `user_send_feas_sol()` (see Section 7.3.1). All that is required is to pass out an array of user indices that are in the feasible solution that you want to trace. Each time the subproblem which admits this feasible solution is branched on, the branch that continues to admit the solution is marked. When one of these marked subproblems is pruned, the user is notified.

6.8.5 Using the `Interactive Graph Drawing` Software

The Interactive Graph Drawing (IGD) software package is included with SYMPHONY and SYMPHONY facilitates its use through interfaces with the package. The package, which is a Tcl/Tk application, is extremely useful for developing and debugging applications involving graph-based problems. Given display coordinates for each node in the graph, IGD can display support graphs corresponding to fractional solutions with or without edge weights and node labels and weights, as well as other information. Furthermore, the user can interactively modify the graph by, for instance, moving the nodes apart to "disentangle" the edges. The user can also interactively enter violated cuts through the IGD interface.

To use IGD, you must have installed PVM since the drawing window runs as a separate application and communicates with the user's routines through message passing. To compile the graph drawing application, type "`make dg`" in the SYMPHONY root directory. The user routines in the file `user_dg.c` can be filled in, but it is not necessary to fill anything in for basic applications.

After compiling `dg`, the user must write some subroutines that communicate with `dg` and cause the graph to be drawn. Regrettably, this is currently a little more complicated than it needs to be and is not well documented. However, by looking at the sample application, it should be possible to see how it is done. To enable graph drawing, put the line `do_draw_graph 1` into the parameter file or use the `-d` command line option. It can be difficult to get IGD to work. If you are interested in using it and cannot get it to work, feel free to contact me.

6.8.6 Other Debugging Techniques

Another useful built-in function is `write_mps()`, which will write the current LP relaxation to a file in MPS format. This file can then be read into the LP solver interactively or examined by hand for errors. Many times, CPLEX gives much more explicit error messages interactively than through the callable library. The form of the function is

```
void write_mps(LPdata *lp_data, char *fname)
```

where `fname` is the name of the file to be written. If SYMPHONY is forced to abandon solution of an LP because the LP solver returns an error code, the current LP relaxation is automatically written to the file "`matrix.[bc_index].[iter_num].mps`" where *bc_index* is the index of the current subproblem and *iter_num* is the current iteration number. The `write_mps()` function can be called using breakpoint code to examine the status of the matrix at any point during execution.

Logging is another useful feature. Logging the state of the search tree can help isolate some problems more easily. See Section 7.4.4 for the appropriate parameter settings to use logging.

6.9 Controlling Execution and Output

Calling SYMPHONY with no arguments simply lists all command-line options. Most of the common parameters can be set on the command line. Sometimes, however, it may be easier to use a parameter file. To invoke SYMPHONY with a parameter file type "`master -f filename ...`" where filename is the name of the parameter file. The format of the file is explained in Section 7.4.

The output level can be controlled through the use of the verbosity parameter. Setting this parameter at different levels will cause different progress messages to be printed out. Level 0 only prints out the introductory and solution summary messages, along with status messages every 10 minutes. Level 1 prints out a message every time a new node is created. Level 3 prints out messages describing each iteration of the solution process. Levels beyond 3 print out even more detailed information.

There are also two possible graphical interfaces. For graph-based problems, the Interactive Graph Drawing Software allows visual display of fractional solutions, as well as feasible and optimal solutions discovered during the solution process. For all types of problems, VBCTOOL creates a visual picture of the branch and cut tree, either in real time as the solution process evolves or as an emulation from a file created by SYMPHONY. See Section 7.4.4 for information on how to use VBCTOOL with SYMPHONY. Binaries for VBCTOOL can be obtained at
`http://www.informatik.uni-koeln.de/ls_juenger/projects/vbctool.html`.

6.10 Other Resources

There is a SYMPHONY user's list serve for posting questions/comments. To subscribe, send "`subscribe symphony-users`" to `majordomo@branchandcut.org`. There is also a Web site for SYMPHONY at `http://branchandcut.org/SYMPHONY` . Bug reports can be sent to
`symphony-bugs@branchandcut.org`.

Chapter 7

Reference

7.1 Callable Library C API

This chapter specifies the interface for using SYMPHONY's callable library. These function calls can be used to build custom applications that call SYMPHONY as a subroutine, as described in Section 3.1. All callable library function begin with the prefix `sym_`. To call these function from an application, include the header file `symphony_api.h` and then link with the SYMPHONY library as described in Section5. In general, if an array is requested, such as the array of lower bounds on the variables, for instance, the user is responsible for allocating an array of appropriate size and passing it to SYMPHONY. SYMPHONY will then fill up the array.

7.1.1 Primary Interface Functions

sym_open_environment

```
sym_environment *sym_open_environment()
```

Description:

This routine is used to get a new SYMPHONY environment to be passed as an argument to all other API subroutines. This routine also invokes the callback function `user_initialize()` (see Section 7.3.1).

Return values:

`NULL`	Error. Environment could not be initialized. None of the other API subroutines can be called after this point.
`sym_environment *`	Pointer to a successfully opened environment

sym_create_copy_environment

```
sym\_environment *sym_create_copy_environment(sym_environment *env)
```

Description:

This routine is used to copy the given environment.

Arguments:

`sym_environment *env`	IN	Pointer to the SYMPHONY environment.

Return values:

tt NULL	An empty environment is passed in.
`SYM_ENVIRONMENT *`	Pointer to the copy of the environment.

sym_parse_command_line

```
int sym_parse_command_line(sym_environment *env, int argc, char **argv)
```

Description:

This routine parses the command line arguments. It must be called whenever the user specifies any of SYMPHONY's built-in command-line switches. For instance, this is the case when the user specifies the location of an MPS or GMPL file using the `-F` switch or when the user specifies the location of a parameter file with the `-f` switch. This command also invokes the user callback function `user_readparams()` (see Section 7.3.1).

Arguments:

`sym_environment *env`	INOUT	Pointer to the SYMPHONY environment.
`int argc`	IN	The number of command line arguments.
`char **argv`	IN	Array of pointers to these arguments.

Return values:

`ERROR__USER`	Error. User error detected in `user_readparams()` function.
`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.
`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.

sym_find_initial_bounds

```
int sym_find_initial_bounds(sym_environment *env)
```

Description:

This routine invokes the user callback `user_start_heurs()` (see Section 7.3.1) to set the priori bound for the problem.

Arguments:

`sym_environment *env`	INOUT	Pointer to the SYMPHONY environment.

Return values:

`ERROR__USER`	Error. User error detected in `user_start_heurs()` function.
`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.
`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.

sym_load_problem

```
int sym_load_problem(sym_environment *env)
```

Description:

This routine loads the description of the problem given in MPS or GMPL/AMPL format or in a file read by a custom file parser implemented in the `user_io()` (see Section 7.3.1) callback. If the problem is to be loaded from an MPS or a GMPL/AMPL file whose location is specified on the command line, then the `sym_parse_command_line()` function has to be invoked beforehand. This function also invokes the user callback `user_initialize_root_node()` (see Section 7.3.1). Note that if the user wishes to load the problem manually without implementing a callback or using one of SYMPHONY's built-in parsers (as is typically done in other callable libraries), then the `sym_explicit_load_problem()` routine should be used.

Arguments:

`sym_environment *env`	INOUT	Pointer to the SYMPHONY environment.

Return values:

`ERROR__USER`	Error. User error detected in one of `user_io()`, `user_init_draw()` functions.
`ERROR__READING_GMPL_FILE`	Error. Error detected in the given GMPL/AMPL file.
`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.
`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.

sym_explicit_load_problem

```
int sym_explicit_load_problem_user(sym_environment * env, int numcols,
                          int numrows, int *start, int *index, double *value,
                          double *collb, double *colub, char *is_int,
                          double *obj, double *obj2, char *rowsen,
                          double *rowrhs, double *rowrng, char make_copy)
```

Description:

This routine is used to load a problem description into SYMPHONY manually. The constraint matrix is passed in a standard column-ordered format. The arguments here are the same as the fields in the `MIPdesc` data structure discussed in Section 7.3.2. Please see the discussion there for a more detailed description of the arguments here.

Arguments:

`sym_environment *env`	INOUT	Pointer to the SYMPHONY environment.
`int numcols`	IN	Number of the columns.
`int numrows`	IN	Number of the rows.
`int *start`	IN	Array of the starting positions of each of column.
`int *index`	IN	Array of the row indices corresponding to each entry of `value`.
`int *value`	IN	Array of the values of nonzero entries of the constraint matrix in *column order*.
`double *collb`	IN	Array of the lower bounds of the columns.
`double *colub`	IN	Array of the upper bounds of the columns.
`double *obj`	IN	Array of the objective function coefficients.
`double *obj2`	IN	Array of the second objective function coefficients when multi criteria solver is to be used.
`char *rowsen`	IN	Array of the senses of the constraints. 'L': ≤ constraint 'E': = constraint 'G': ≥ constraint 'R': ranged constraint 'N': free constraint
`double *rowrhs`	IN	Array of the right hand side values.
`double *rowrng`	IN	Array of the row ranges. (`sym_get_row_upper`) - (`sym_get_row_lower`) if the row sense is `'R'`, 0 otherwise.
`char make_copy`	IN	SYMPHONY will create the copies of these arrays for internal usage if this flag is set to true, otherwise, will own them.

Return values:

`ERROR__USER`	Error. User error detected in `user_initialize_root_node` function.
`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.
`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.

sym_solve

```
int sym_solve(sym_environment *env)
```

Description:

This routine solves the currently loaded MILP problem from scratch even in the presence of a loaded warm start. Any warm start information loaded or kept before will be deleted from the environment!

Arguments:

`sym_environment *env`	INOUT	Pointer to the SYMPHONY environment.

Return values:

`ERROR__USER`	Error. User error detected in one of `user_send_lp_data()`, `user_send_cg_data()`, `user_send_cp_data()`, `user_receive_feasible_solution()`, `user_display_solution()`, `user_process_own_messages()` functions.
`TM_OPTIMAL_SOLUTION_FOUND`	Tree Manager (TM) found the optimal solution and stopped.
`TM_TIME_LIMIT_EXCEEDED`	TM stopped after reaching the predefined time limit.
`TM_NODE_LIMIT_EXCEEDED`	TM stopped after reaching the predefined node limit.
`TM_TARGET_GAP_ACHIEVED`	TM stopped after achieving the predefined target gap.
`TM_FOUND_FIRST_FEASIBLE`	TM stopped after finding the first feasible solution.
`TM_ERROR__NO_BRANCHING_CANDIDATE`	Error. TM stopped. User didn't select branching candidate in `user_select_candidates()` callback.
`TM_ERROR__ILLEGAL_RETURN_CODE`	Error. TM stopped after getting a non-valid return code.
`TM_ERROR__NUMERICAL_INSTABILITY`	Error. TM stopped due to some numerical difficulties.
`TM_ERROR__COMM_ERROR`	Error. TM stopped due to communication error.
`TM_ERROR__USER`	Error. TM stopped. User error detected in one of user callbacks called during TM processes.

sym_warm_solve

```
int sym_warm_solve(sym_environment *env)
```

Description:

This routine re-solves the corresponding problem after some of the parameters have been changed or problem data has been modified from a warm start. If the user plans to invoke this routine, the `keep_warm_start` parameter must be set to `TRUE` before the initial call to the `sym_solve()` routine, so that SYMPHONY will collect the necessary warm starting information during the solve procedure.

Arguments:

`sym_environment *env`	INOUT	Pointer to the SYMPHONY environment.

Return values:

`ERROR__USER`	Error. User error detected in one of `user_send_lp_data`, `user_send_cg_data`, `user_send_cp_data`, `user_receive_feasible_solution`, `user_display_solution`, `user_process_own_messages` functions.
`TM_OPTIMAL_SOLUTION_FOUND`	Tree Manager (TM) found the optimal solution and stopped.
`TM_TIME_LIMIT_EXCEEDED`	TM stopped after reaching the predefined time limit.
`TM_NODE_LIMIT_EXCEEDED`	TM stopped after reaching the predefined node limit.
`TM_TARGET_GAP_ACHIEVED`	TM stopped after achieving the predefined target gap.
`TM_FOUND_FIRST_FEASIBLE`	TM stopped after finding the first feasible solution.
`TM_ERROR__NO_BRANCHING_CANDIDATE`	Error. TM stopped. User didn't select branching candidate in `user_select_candidates` callback
`TM_ERROR__ILLEGAL_RETURN_CODE`	Error. TM stopped after getting a non-valid return code.
`TM_ERROR__NUMERICAL_INSTABILITY`	Error. TM stopped due to some numerical difficulties.
`TM_ERROR__COMM_ERROR`	Error. TM stopped due to communication error.
`TM_ERROR__USER`	Error. TM stopped. User error detected in one of user callbacks called during TM processes.
`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.

sym_mc_solve

```
int sym_mc_solve(sym_environment *env)
```

Description:

This routine is used to solve the loaded problem as a multicriteria problem. For this function, a second objective function must be set either by calling the `sym_set_obj2_coeff()` function or by passing it directly using the `sym_explict_load_problem()` function.

Arguments:

`sym_environment *env`	INOUT	Pointer to the SYMPHONY environment.

Return values:

`ERROR__USER`	Error. User error detected in one of `user_send_lp_data()`, `user_send_cg_data()`, `user_send_cp_data()`, `user_receive_feasible_solution()`, `user_display_solution()`, `user_process_own_messages()` functions.
`TM_OPTIMAL_SOLUTION_FOUND`	The set of supported or nondominated solutions have been found.
`TM_ERROR__NO_BRANCHING_CANDIDATE`	Error. TM stopped. User didn't select branching candidate in `user_select_candidates` callback
`TM_ERROR__ILLEGAL_RETURN_CODE`	Error. TM stopped after getting a non-valid return code.
`TM_ERROR__NUMERICAL_INSTABILITY`	Error. TM stopped due to some numerical difficulties.
`TM_ERROR__COMM_ERROR`	Error. TM stopped due to communication error.
`TM_ERROR__USER`	Error. TM stopped. User error detected in one of user callbacks activated by user and invoked during TM processes.
`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.

sym_create_permanent_cut_pools

```
int sym_create_permanent_cut_pools(sym_environment *env, int *cp_num)
```

Description:

This routine is used to create a global cut pool that will be saved even after the solve call exits and can be used to initialize the cut pool for later solve calls. This can be useful when solving a series of related MILPs that share classes of globally valid inequalities. For instance, if only the objective function is varied, as is the case with multicriteria integer programming, then cuts can be saved for use in later solve calls.

Arguments:

`sym_environment *env`	INOUT	Pointer to the SYMPHONY environment.
`int *cp_num`	OUT	Pointer to an integer indicating the number of cut pools stored in the environment.

Return values:

`INT`	The number of the cut pools created.

sym_set_user_data

```
int sym_set_user_data(sym_environment *env, void *user)
```

Description:

This routine is used to give SYMPHONY a pointer to the user's problem data structure. This pointer will then be handed back to the user during subsequent calls to user callbacks. This allows the user to store static problem data. Note that this pointer can also be stored by filling out the callback function `user_initialize()` (see Section 7.3.1).

Arguments:

`sym_environment *env`	INOUT	Pointer to the SYMPHONY environment.
`void *user`	IN	Pointer to the user defined problem structure.

Return values:

`ERROR__USER`	Error in the passed in `user` structure.
`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully
`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.

sym_close_environment

```
int sym_close_environment(sym_environment *env)
```

Description:

This routine closes the environment and returns the allocated memory.

Arguments:

`sym_environment *env`	INOUT	Pointer to the SYMPHONY environment.

Return values:

`ERROR__USER`	Error. User error detected in `user_free_master()` function.
`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.
`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.

7.1.2 Parameter Query and Modification

sym_set_defaults

```
int sym_set_defaults(sym_environment *env)
```

Description:

This routine sets all the environment variables and parameters to their default values.

Arguments:

`sym_environment *env`	INOUT	Pointer to the SYMPHONY environment to be modified.

Return values:

`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.
`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.

sym_set_int_param

```
void sym_set_int_param(sym_environment *env, char *key, int value)
```

Description:

This routine is used to set an integer type parameter.

Arguments:

`sym_environment *env`	INOUT	Pointer to the SYMPHONY environment.
`char *key`	IN	The name of the parameter to be set.
`int value`	OUT	New value of the corresponding parameter.

sym_set_dbl_param

```
void sym_set_int_param(sym_environment *env, char *key, double value)
```

Description:

This routine is used to set a double type parameter.

Arguments:

`sym_environment *env`	INOUT	Pointer to the SYMPHONY environment.
`char *key`	IN	The name of the parameter to be set.
`double value`	OUT	New value of the corresponding parameter.

sym_set_str_param

```
void sym_set_str_param(sym_environment *env, char *key, char *value)
```

Description:

This routine is used to set a string type parameter.

Arguments:

`sym_environment *env`	INOUT	Pointer to the SYMPHONY environment.
`char *key`	IN	The name of the parameter to be set.
`char *value`	OUT	New value of the corresponding parameter.

sym_get_int_param

```
int sym_get_int_param(sym_environment *env, char *key)
```

Description:

This routine is used to get the value of an integer type parameter.

Arguments:

`sym_environment *env`	INOUT	Pointer to the SYMPHONY environment.
`char *key`	IN	The name of the parameter.

Return values:

`INT`	An integer indicating the value of the parameter.

sym_get_dbl_param

```
double sym_get_int_param(sym_environment *env, char *key)
```

Description:

This routine is used to get the value of a double type parameter.

Arguments:

`sym_environment *env`	INOUT	Pointer to the SYMPHONY environment.
`char *key`	IN	The name of the parameter.

Return values:

`DOUBLE`	A double indicating the value of the parameter.

sym_get_str_param

```
char *sym_get_int_param(sym_environment *env, char *key)
```

Description:

This routine is used to get the value of a string type parameter.

Arguments:

`sym_environment *env`	INOUT	Pointer to the SYMPHONY environment.
`char *key`	IN	The name of the parameter.

Return values:

`CHAR*`	A character array indicating the value of the parameter.

7.1.3 Solver Status Query Functions

sym_get_status

```
int sym_get_status(sym_environment *env)
```

Description: This post-solution query routine is used to learn the termination status of the solution procedure.

Arguments: `sym_environment *env` IN Pointer to the SYMPHONY environment.

Return values:

`ERROR__USER`	Error. User error detected in one of `user_send_lp_data()`, `user_send_cg_data()`, `user_send_cp_data()`, `user_receive_feasible_solution()`, `user_display_solution()`, `user_process_own_messages()` functions.
`TM_OPTIMAL_SOLUTION_FOUND`	Tree Manager (TM) found the optimal solution and stopped.
`TM_TIME_LIMIT_EXCEEDED`	TM stopped after reaching the predefined time limit.
`TM_NODE_LIMIT_EXCEEDED`	TM stopped after reaching the predefined node limit.
`TM_TARGET_GAP_ACHIEVED`	TM stopped after achieving the predefined target gap.
`TM_FOUND_FIRST_FEASIBLE`	TM stopped after finding the first feasible solution.
`TM_ERROR__NO_BRANCHING_CANDIDATE`	Error. TM stopped. User didn't select branching candidate in `user_select_candidates()` callback
`TM_ERROR__ILLEGAL_RETURN_CODE`	Error. TM stopped after getting an invalid return code.
`TM_ERROR__NUMERICAL_INSTABILITY`	Error. TM stopped due to some numerical difficulties.
`TM_ERROR__COMM_ERROR`	Error. TM stopped due to communication error.
`TM_ERROR__USER`	Error. TM stopped. User error detected in one of user callbacks called during TM processes.

sym_is_proven_optimal

```
int sym_is_proven_optimal(sym_environment *env)
```

Description:

This post-solution query routine is used to learn whether the problem was solved to optimality.

Arguments:

`sym_environment *env`	IN	Pointer to the SYMPHONY environment.

Return values:

`TRUE`	The problem was solved to optimality.
`FALSE`	The problem was not solved to optimality.

sym_is_proven_primal_infeasible

```
int sym_is_proven_primal_infeasible(sym_environment *env)
```

Description:

This post-solution query routine is used to learn whether the problem was proven to be infeasible.

Arguments:

`sym_environment *env`	IN	Pointer to the SYMPHONY environment.

Return values:

`TRUE`	The problem was proven to be infeasible.
`FALSE`	The problem was not proven to be infeasible.

sym_is_iteration_limit_reached

```
int sym_is_iteration_limit_reached(sym_environment *env)
```

Description:

This post-solution query routine is used to learn whether the iteration (node limit) was reached. It can also be used if "find_first_feasible" parameter was set to true before solving the problem.

Arguments:

`sym_environment *env`	IN	Pointer to the SYMPHONY environment.

Return values:

`TRUE`	The iteration limit is reached.
`FALSE`	The iteration limit is not reached.

sym_is_time_limit_reached

```
int sym_is_time_limit_reached(sym_environment *env)
```

Description:

This post-solution query routine is used to learn whether the time limit was reached.

Arguments:

`sym_environment *env`	IN	Pointer to the SYMPHONY environment.

Return values:

`TRUE`	Time limit was reached.
`FALSE`	Time limit was not reached.

sym_is_target_gap_achieved

```
int sym_is_target_gap_achieved(sym_environment *env)
```

Description:

This post-solution query routine is used to learn whether the target gap was reached.

Arguments:

sym_environment *env	IN	Pointer to the SYMPHONY environment.

Return values:

TRUE	Target gap was reached.
FALSE	Target gap was not reached.

sym_is_abandoned

```
int sym_is_abandoned(sym_environment *env)
```

Description:

This post-solution query routine is used to learn whether the problem was abandoned for some reason.

Arguments:

sym_environment *env	IN	Pointer to the SYMPHONY environment.

Return values:

TRUE	The problem was abandoned.
FALSE	The problem was not abandoned.

7.1.4 Data Query Functions

sym_create_copy_mip_desc

```
MIPdesc *sym_create_copy_mip_desc(sym_environment *env)
```

Description:

This routine is used to copy the problem description loaded to the environment.

Arguments:

`sym_environment *env`	IN	Pointer to the SYMPHONY environment.

Return values:

`NULL`	An empty environment is passed in or there is no problem description loaded to the environment.
`MIPdesc *`	Pointer to the copy of the problem description.

sym_get_num_cols

```
int sym_get_num_cols(sym_environment *env, int *numcols)
```

Description:

This routine is used to get the number of the columns of the current problem.

Arguments:

`sym_environment *env`	IN	Pointer to the SYMPHONY environment.
`int *numcols`	OUT	Pointer to an integer indicating the number of columns.

Return values:

`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.
`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.

sym_get_num_rows

```
int sym_get_num_cols(sym_environment *env, int *numrows)
```

Description:

This routine is used to get the number of the rows of the current problem.

Arguments:

`sym_environment *env`	IN	Pointer to the SYMPHONY environment.
`int *numrows`	OUT	Pointer to an integer indicating the number of rows.

Return values:

`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.
`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.

sym_get_num_elements

```
int sym_get_num_elements(sym_environment *env, int *numelems)
```

Description:

This routine is used to get the number of non-zero entries of the constraint matrix of the current problem.

Arguments:

`sym_environment *env`	IN	Pointer to the SYMPHONY environment.
`int *numelems`	OUT	Pointer to an integer indicating the number of non-zero elements.

Return values:

`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.
`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.

sym_get_col_lower

```
int sym_get_col_lower(sym_environment *env, double *collb)
```

Description:

This routine is used to get the lower bounds of the variables.

Arguments:

`sym_environment *env`	IN	Pointer to the SYMPHONY environment.
`double *collb`	OUT	Pointer to a double type array to be filled by the column lower bounds. Note that, the size of this array has to be at least the number of columns.

Return values:

`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.
`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.

sym_get_col_upper

```
int sym_get_col_upper(sym_environment *env, double *colub)
```

Description:

This routine is used to get the upper bounds of the variables.

Arguments:

`sym_environment *env`	IN	Pointer to the SYMPHONY environment.
`double *colub`	OUT	Pointer to a double type array to be filled by the column upper bounds. Note that, the size of this array has to be at least the number of columns.

Return values:

`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.
`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.

sym_get_row_sense

```
int sym_get_row_sense(sym_environment *env, char *rowsen)
```

Description:

This routine is used to get the row senses.

Arguments:

`sym_environment *env`	IN	Pointer to the SYMPHONY environment.
`char *rowsen`	OUT	Pointer to a char type array to be filled by the row senses. Note that, the size of this array has to be at least the number of rows.

Return values:

`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.
`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.

sym_get_rhs

```
int sym_get_rhs(sym_environment *env, double *rowrhs)
```

Description:

This routine is used to get the right hand side vector.

Arguments:

`sym_environment *env`	IN	Pointer to the SYMPHONY environment.
`double *rowrhs`	OUT	Pointer to a double type array to be filled by the right hand side vector. Note that, the size of this array has to be at least the number of rows.

Return values:

`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.
`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.

sym_get_row_range

```
int sym_get_row_range(sym_environment *env, double *rowrng)
```

Description:

This routine is used to get the row ranges.

Arguments:

`sym_environment *env`	IN	Pointer to the SYMPHONY environment.
`double *rowrng`	OUT	Pointer to a double type array to be filled by the row range values. Note that, the size of this array has to be at least the number of rows.

Return values:

`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.
`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.

sym_get_row_lower

```
int sym_get_row_lower(sym_environment *env, double *rowlb)
```

Description:

This routine is used to get the lower bounds of the rows.

Arguments:

`sym_environment *env`	IN	Pointer to the SYMPHONY environment.
`double *rowlb`	OUT	Pointer to a double type array to be filled by the row lower bounds. Note that, the size of this array has to be at least the number of rows.

Return values:

`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.
`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.

sym_get_row_upper

```
int sym_get_row_upper(sym_environment *env, double *rowub)
```

Description:

This routine is used to get the upper bounds of the rows.

Arguments:

`sym_environment *env`	IN	Pointer to the SYMPHONY environment.
`double *rowub`	OUT	Pointer to a double type array to be filled by the row upper bounds. Note that, the size of this array has to be at least the number of rows.

Return values:

`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.
`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.

sym_get_obj_coeff

```
int sym_get_obj_coeff(sym_environment *env, double *obj)
```

Description:

This routine is used to get the objective vector.

Arguments:

`sym_environment *env`	IN	Pointer to the SYMPHONY environment.
`double *obj`	OUT	Pointer to a double type array to be filled by the objective vector. Note that, the size of this array has to be at least the number of columns.

Return values:

`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.
`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.

sym_get_obj2_coeff

```
int sym_get_obj2_coeff(sym_environment *env, double *obj2)
```

Description:

This routine is used to get the second objective vector if it exists. By default, it is set to the zero vector.

Arguments:

`sym_environment *env`	IN	Pointer to the SYMPHONY environment.
`double *obj2`	OUT	Pointer to a double type array to be filled by the second objective vector. Note that, the size of this array has to be at least the number of columns.

Return values:

`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.
`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.

sym_get_obj_sense

```
int sym_get_obj_sense(sym_environment *env, int *sense)
```

Description:

This routine is used to get the objective sense.

Arguments:

`sym_environment *env`	IN	Pointer to the SYMPHONY environment.
`int *sense`	OUT	Pointer to an integer indicating the objective sense. In return, it will be `1` in case of minimization and `-1` in case of maximization.

Return values:

`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.
`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.

sym_is_continuous

```
int sym_is_continuous(sym_environment *env, int index, int *value)
```

Description:

This routine is used to learn whether the queried variable is continuous.

Arguments:

`sym_environment *env`	IN	Pointer to the SYMPHONY environment.
`int index`	IN	The index of the queried variable. Note that, it has to be at most the number of columns.
`int *value`	OUT	Pointer to a boolean indicating the variable status.

Return values:

`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.
`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.

sym_is_binary

```
int sym_is_binary(sym_environment *env, int index, int *value)
```

Description:

This routine is used to learn whether the queried variable is binary.

Arguments:

`sym_environment *env`	IN	Pointer to the SYMPHONY environment.
`int index`	IN	The index of the queried variable. Note that, it has to be at most the number of columns.
`int *value`	OUT	Pointer to a boolean indicating the variable status.

Return values:

`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.
`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.

sym_is_integer

```
int sym_is_integer(sym_environment *env, int index, int *value)
```

Description:

This routine is used to ask whether the queried variable is integer.

Arguments:

`sym_environment *env`	IN	Pointer to the SYMPHONY environment.
`int index`	IN	Index of the queried variable. Note that, it has to be at most the number of columns.
`int *value`	OUT	Pointer to a boolean indicating the variable status.

Return values:

`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.
`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.

sym_get_infinity

```
double sym_get_infinity()
```

Description:

This routine returns the infinity value of SYMPHONY.

Arguments:

Return values:

`DOUBLE` Infinity value of SYMPHONY

sym_get_col_solution

```
int sym_get_col_solution(sym_environment *env, double *colsol)
```

Description:

This routine is used to get the post-solution column values.

Arguments:

`sym_environment *env`	IN	Pointer to the SYMPHONY environment.
`double *colsol`	OUT	Pointer to a double type array to be filled by the solution vector. Note that, the size of this array has to be at least the number of columns.

Return values:

`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.
`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.

sym_get_row_activity

```
double *sym_get_row_activity(sym_environment *env, double *rowact)
```

Description:

This routine is used to get the row activities which are defined as the left hand side values, i.e., constraint matrix times the solution.

Arguments:

`sym_environment *env`	IN	Pointer to the SYMPHONY environment.
`double *rowact`	OUT	Pointer to a double type array to be filled by the row activity values. Note that, the size of this array has to be at least the number of rows.

Return values:

`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.
`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.

sym_get_obj_val

```
double *sym_get_obj_val(sym_environment *env, double *objval)
```

Description:

This routine is used to get the objective value after solving the problem.

Arguments:

sym_environment *env	IN	Pointer to the SYMPHONY environment.
double *objval	OUT	Pointer to a double indicating the post-solution objective value.

Return values:

FUNCTION_TERMINATED_ABNORMALLY	Function invoked unsuccessfully.
FUNCTION_TERMINATED_NORMALLY	Function invoked successfully.

sym_get_primal_bound

```
double *sym_get_primal_bound(sym_environment *env, double *ub)
```

Description:

This routine is used to get the a priori upper/lower bound for the problem.

Arguments:

`sym_environment *env`	IN	Pointer to the SYMPHONY environment.
`double *ub`	OUT	Pointer to a double indicating the upper (for minimization) or lower (for maximization) bound obtained through user defined primal heuristics.

Return values:

`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.
`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.

sym_get_iteration_count

```
double *sym_get_iteration\_count(sym_environment *env, int *numnodes)
```

Description:

This routine is used to get the number of the analyzed nodes of the branching tree after solving the problem. It can also be used to query the status of a loaded warm start.

Arguments:

`sym_environment *env`	IN	Pointer to the SYMPHONY environment.
`int *numnodes`	OUT	Pointer to an integer indicating the number of nodes analyzed so far.

Return values:

`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.
`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.

7.1.5 Data Modification Functions

sym_set_obj_coeff

```
double *sym_set_obj_coeff(sym_environment *env, int index, double value)
```

Description:

This routine is used to set an objective coefficient.

Arguments:

`sym_environment *env`	INOUT	Pointer to the SYMPHONY environment.
`int index`	IN	Index of the objective coefficient to be modified. Note that, it has to be at most the number of columns.
`double value`	IN	New objective value of the corresponding column.

Return values:

`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.
`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.

sym_set_obj2_coeff

```
double *sym_set_obj2_coeff(sym_environment *env, int index, double value)
```

Description:

This routine is used to set a coefficient of the second objective function of the corresponding bicriteria problem.

Arguments:

`sym_environment *env`	INOUT	Pointer to the SYMPHONY environment.
`int index`	IN	Index of the objective coefficient to be modified. Note that, it has to be at most the number of columns.
`double value`	IN	New value of the objective coefficient to be modified.

Return values:

`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.
`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.

sym_set_col_lower

```
double *sym_set_col_lower(sym_environment *env, int index, double value)
```

Description:

This routine is used to set the lower bound of a variable.

Arguments:

`sym_environment *env`	INOUT	Pointer to the SYMPHONY environment.
`int index`	IN	Index of the variable. Note that, it has to be at most the number of columns.
`double value`	IN	New lower bound of the variable.

Return values:

`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.
`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.

sym_set_col_upper

```
double *sym_set_col_upper(sym_environment *env, int index, double value)
```

Description:

This routine is used to set the upper bound of a variable.

Arguments:

`sym_environment *env`	INOUT	Pointer to the SYMPHONY environment.
`int index`	IN	Index of the variable. Note that, it has to be at most the number of columns.
`double value`	IN	New upper bound of the variable.

Return values:

`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.
`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.

sym_set_row_lower

```
double *sym_set_row_lower(sym_environment *env, int index, double value)
```

Description:

This routine is used to set the lower bound of a row.

Arguments:

`sym_environment *env`	INOUT	Pointer to the SYMPHONY environment.
`int index`	IN	Index of the row. Note that, it has to be at most the number of rows.
`double value`	IN	New lower bound of the row.

Return values:

`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.
`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.

sym_set_row_upper

```
double *sym_set_row_upper(sym_environment *env, int index, double value)
```

Description:

This routine is used to set the upper bound of a row.

Arguments:

`sym_environment *env`	INOUT	Pointer to the SYMPHONY environment.
`int index`	IN	Index of the row. Note that, it has to be at most the number of rows.
`double value`	IN	New upper bound of the row.

Return values:

`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.
`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.

sym_set_row_type

```
int sym_set_row_type(sym_environment *env, int index, char rowsense,
                     double rowrhs, double rowrng)
```

Description:

This routine is used to set the characteristics of a row.

Arguments:

`sym_environment *env`	INOUT	Pointer to the SYMPHONY environment.
`int index`	IN	Index of the row. Note that, it has to be at most the number of rows.
`char rowsense`	IN	New sense of the row.
`double rowrhs`	IN	New value of the right hand side of the row.
`double rowrng`	IN	New value of the row range.

Return values:

`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.
`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.

sym_set_obj_sense

```
int sym_set_obj_sense(sym_environment *env, int sense)
```

Description:

This routine is used to set the objective sense. By default, SYMPHONY will solve a minimization problem.

Arguments:

`sym_environment *env`	INOUT	Pointer to the SYMPHONY environment.
`int sense`	IN	New sense of the objective function. It can be 1 and -1 for minimization and maximization. Otherwise, SYMPHONY will assume the objective sense to be minimization.

Return values:

`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.
`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully

sym_set_col_solution

```
int sym_set_col_solution(sym_environment *env, double *colsol)
```

Description:

This routine is used to set the current solution if a known one exists. Note that setting the column solution will not affect or help the treemanager's processes other than setting the best feasible solution and the corresponding upper bound.

Arguments:

`sym_environment *env`	INOUT	Pointer to the SYMPHONY environment.
`double *colsol`	IN	Pointer to a double type array of the known column values. Note that, if the given solution is not feasible or if a better solution was found/loaded before, SYMPHONY will refuse to set the column solution and will leave this function without success.

Return values:

`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.
`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.

sym_set_primal_bound

```
int sym_set_primal_bound(sym_environment *env, double bound)
```

Description:

This routine is used to set a priori upper/lower bound to the problem.

Arguments:

`sym_environment *env`	INOUT	Pointer to the SYMPHONY environment.
`double double`	IN	The value of the priori upper (for minimization) or lower (for maximization) bound.

Return values:

`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.
`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.

sym_set_continuous

```
int sym_set_continuous(sym_environment *env, int index)
```

Description:

This routine is used to set the type of a variable to be continuous.

Arguments:

`sym_environment *env`	INOUT	Pointer to the SYMPHONY environment.
`int index`	IN	The index of the variable to be modified. Note that, it has to be at most the number of columns.

Return values:

`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.
`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.

sym_set_integer

```
int sym_set_continuous(sym_environment *env, int index)
```

Description:

This routine is used to set the type of a variable to be integer.

Arguments:

`sym_environment *env`	INOUT	Pointer to the SYMPHONY environment.
`int index`	IN	The index of the variable to be modified. Note that, it has to be at most the number of columns.

Return values:

`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.
`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.

sym_set_col_names

```
int sym_set_col_names(sym_environment * env, char **colname)
```

Description:

This routine is used to set the column names.

Arguments:

`sym_environment *env`	INOUT	Pointer to the SYMPHONY environment.
`char **colname`	IN	Pointer to a string array including the column names. Note that, the size of this array has to be at least the number of columns.

Return values:

`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.
`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.

sym_add_col

```
int sym_add_col(sym_environment *env, int numelems, int *indices,
                double *elements, double collb, double colub,
                double obj, char *name)
```

Description:

This routine is used to add a new column to the original problem description.

Arguments:

`sym_environment *env`	INOUT	Pointer to the SYMPHONY environment.
`int numelems`	IN	An integer indicating the non zero elements of the column.
`int *indices`	IN	Pointer to an integer type array indicating the row indices of the non zero elements of the column and having a size of at least `numelems`.
`double *elements`	IN	Pinter to a double type array indicating the values of the non zero elements of the column and having a size of at least `numelems`.
`double collb`	IN	A double indicating the lower bound of the column.
`double colub`	IN	A double indicating the upper bound of the column.
`double obj`	IN	A double indicating the objective coefficient value of the column.
`char *name`	IN	Pointer to a string of the name of the column.

Return values:

`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.
`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.

sym_add_row

```
int sym_add_row(sym_environment *env, int numelems, int *indices,
                double *elements, char rowsen, double rowrhs,
                double rowrng)
```

Description:

This routine is used to add a new row to the original constraint matrix.

Arguments:

`sym_environment *env`	INOUT	Pointer to the SYMPHONY environment.
`int numelems`	IN	An integer indicating the non zero elements of the row.
`int *indices`	IN	Pointer to an integer type array indicating the column indices of the non zero elements of the row and having a size of at least `numelems`.
`double *elements`	IN	Pointer to a double type array indicating the values of the non zero elements of the row and having a size of at least `numelems`.
`char rowsen`	IN	A character indicating the sense of the row.
`double rowrhs`	IN	A double indicating the right hand side of the row.
`double rowrng`	IN	A double indicating the range value of the row.

Return values:

`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.
`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.

sym_delete_cols

```
int sym_delete_cols(sym_environment *env, int num, int * indices)
```

Description:

This routine is used to delete columns from the original problem description.

Arguments:

`sym_environment *env`	INOUT	Pointer to the SYMPHONY environment.
`int num`	IN	An integer indicating the number of columns to be deleted.
`int *indices`	IN	Pointer to an integer type array indicating the indices of the columns to be deleted and having a size of at least `num`.

Return values:

`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.
`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully or one of the `indices` is out of the range of [0, number of variables-1]

sym_delete_rows

```
int sym_delete_rows(sym_environment *env, int num, int * indices)
```

Description:

This routine is used to delete rows from the original constraint matrix.

Arguments:

`sym_environment *env`	INOUT	Pointer to the SYMPHONY environment.
`int num`	IN	An integer indicating the number of rows to be deleted.
`int *indices`	IN	An array indicating the indices of the rows to be deleted and having a size of at least `num`.

Return values:

`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.
`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully or one of the `indices` is out of the range of [0, number of variables-1]

7.1.6 Warm Starting Functions

sym_write_warm_start_desc

```
int sym_write_warm_start_desc(warm_start_desc *ws, char *file)
```

Description:

This routine is used to write the given warm start structure to a file.

Arguments:

`warm_start_desc *ws`	IN	Pointer to the warm start description to be written.
`char *file`	IN	The name of the file the warm start is desired to be written to.

Return values:

`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.
`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.

sym_read_warm_start

```
int sym_read_warm_start(char *file, warm_start_desc *ws)
```

Description:

This routine is used to read in a warm start structure from a file.

Arguments:

`char *file`	IN	The name of the file the warm start is desired to be read from.
`warm_start_desc *ws`	OUT	Pointer to a warm start object to be read from the file.

Return values:

`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.
`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.

sym_delete_warm_start

```
void sym_delete_warm_start(warm_start_desc *ws)
```

Description:

This routine is used to free a warm start structure and return the allocated memory.

Arguments:

`warm_start_desc *ws`	IN	Pointer to the warm start description to be deleted.

sym_get_warm_start

```
warm_start_desc *sym_get_warm_start(sym_environment *env, int copy_warm_start,
warm_start_desc **ws)
```

Description:

This routine is used to get the warm start description loaded to the environment.

Arguments:

`sym_environment *env`	IN	Pointer to the SYMPHONY environment.
`int copy_warm_start`	IN	A boolean indicating whether the warm start of the environment is desired to be copied or overtaken.
`warm_start_desc **ws`	OUT	Pointer to a pointer to be directed to a copy or the itself of the currently loaded warm start.

Return values:

`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.
`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.

sym_set_warm_start

```
int sym_set_warm_start(sym_environment *env, warm_start_desc *ws)
```

Description:

This routine is used to load a warm start structure to the environment.

Arguments:

`sym_environment *env`	INOUT	Pointer to the SYMPHONY environment.
`warm_start_desc *ws`	IN	Pointer to the warm start structure to be loaded to the environment.

Return values:

`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.
`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.

sym_create_copy_warm_start

```
warm_start_desc *sym_create_copy_warm_start(warm_start_desc *ws)
```

Description:

This routine is used to copy the given warm start structure.

Arguments:

`warm_start_desc *ws`	INOUT	Pointer to the warm start structure to be copied.

Return values:

tt NULL	An empty warm start description is passed in.
`WARM_START_DESC *`	Pointer to the copy of the warm start structure.

7.1.7 Sensitivity Analysis Functions

sym_get_lb_for_new_rhs

```
int sym_get_lb_for_new_rhs(sym_environment *env, int cnt, int *new_rhs_ind,
                           double *new_rhs_val, double *lb_for_new_rhs)
```

Description:

This routine is used for a basic sensitivity analysis of the right hand side case. It returns a lower bound for the problem with a modified right hand side using the information gathered from the branching tree of the original solved problem. Note that, in order to use this feature, the `sensitivity_analysis` parameter needs to be set before solving the original problem.

Arguments:

`sym_environment *env`	IN	Pointer to the SYMPHONY environment.
`int cnt`	IN	The number of the non zero elements in the new right hand side vector.
`int *new_rhs_ind`	IN	Array of the column indices of these non zero elements.
`double *new_rhs_val`	IN	Array of the values of these non zero elements.
`double *lb_for_new_rhs`	OUT	Pointer to a double indicating the lower bound obtained for the new problem.

Return values:

`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.
`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.

sym_get_ub_for_new_rhs

```
int sym_get_ub_for_new_rhs(sym_environment *env, int cnt, int *new_rhs_ind,
                           double *new_rhs_val, double *ub_for_new_rhs)
```

Description:

This routine is used for a basic sensitivity analysis of the right hand side case. It returns a quick upper bound for the problem with a modified right hand side using the information gathered from the branching tree of the original solved problem. Note that, in order to use this feature, the `sensitivity_analysis` parameter needs to be set before solving the original problem.

Arguments:

`sym_environment *env`	IN	Pointer to the SYMPHONY environment.
`int cnt`	IN	The number of the non zero elements in the new right hand side vector.
`int *new_rhs_ind`	IN	Array of the column indices of these non zero elements.
`double *new_rhs_val`	IN	Array of the values of these non zero elements.
`double *ub_for_new_rhs`	OUT	Pointer to a double indicating the lower bound obtained for the new problem. This value will be set to `SYM_INFINITY` if an upper bound can not be found.

Return values:

`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.
`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.

sym_get_lb_for_new_obj

```
int sym_get_lb_for_new_rhs(sym_environment *env, int cnt, int *new_obj_ind,
                           double *new_obj_val, double *lb_for_new_obj)
```

Description:

This routine is used for a basic sensitivity analysis of the objective function case. It returns a quick lower bound for the problem with a modified objective vector using the information gathered from the branching tree of the original solved problem. Note that, in order to use this feature, the `sensitivity_analysis` parameter needs to be set before solving the original problem.

Arguments:

`sym_environment *env`	IN	Pointer to the SYMPHONY environment.
`int cnt`	IN	The number of the non zero elements in the new objective coefficients.
`int *new_obj_ind`	IN	Array of the column indices of these non zero elements.
`double *new_obj_val`	IN	Array of the values of these non zero elements.
`double *lb_for_new_obj`	OUT	Pointer to a double indicating the lower bound obtained for the new problem.

Return values:

`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.
`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.

sym_get_ub_for_new_obj

```
int sym_get_ub_for_new_rhs(sym_environment *env, int cnt, int *new_obj_ind,
                           double *new_obj_val, double *ub_for_new_obj)
```

Description:

This routine is used for a basic sensitivity analysis of the objective function case. It returns a quick lower bound for the problem with a modified objective vector using the information gathered from the branching tree of the original solved problem. Note that, in order to use this feature, the `sensitivity_analysis` parameter needs to be set before solving the original problem.

Arguments:

`sym_environment *env`	IN	Pointer to the SYMPHONY environment.
`int cnt`	IN	The number of the non zero elements in the new objective coefficients.
`int *new_obj_ind`	IN	Array of the column indices of these non zero elements.
`double *new_obj_val`	IN	Array of the values of these non zero elements.
`double *ub_for_new_obj`	OUT	Pointer to a double indicating the upper bound obtained for the new problem. This value will be set to `SYM_INFINITY` if an upper bound can not be found.

Return values:

`FUNCTION_TERMINATED_NORMALLY`	Function invoked successfully.
`FUNCTION_TERMINATED_ABNORMALLY`	Function invoked unsuccessfully.

7.2 Callable Library C++ API

SYMPHONY's C++ interface is derived from COIN-OR's Open Solver Interface (OSI). The OSI methods are implemented simply as wrapped calls to the SYMPHONY C callable library just described. For instance, when an instance of the OSI interface class is constructed, a call is made to `sym_open_environment()` and a pointer to the environment is stored in the class and when the OSI object is destroyed, `sym_close_environment` is called to destroy the environment object. Most subsequent calls within the class can then be made without any arguments. To fully support SYMPHONY's capabilities, we have extended the OSI interface to include some other methods not in the base class. For example, we added calls equivalent to our `sym_parse_command_line()` and `sym_find_initial_bounds()`. Additionally, SYMPHONY has a warm start class derived from the CoinWarmStart base class to support the new functionalities of the MILP warm starting such as `sym_get_warm_start` and `sym_set_warm_start`. They are also implemented as wrapped calls to the C interface library.

In order to have the whole list of the methods and information regarding their usage, see the OSI SYMPHONY interface and SYMPHONY warm start header files (`OsiSymSolverInterface.hpp` and `SymWarmStart.hpp`). Here, we will give the table of the C library equivalent calls of the C++ interface routines with brief descriptions:

C++ Interface	C Interface	Description
OsiSymSolverInterface	sym_open_environment	create a new environment.
loadProblem	sym_load_problem	load the problem read trough an MPS or GMPL file
branchAndBound	sym_solve/sym_warm_solve	solve the MILP problem from scratch or from a warm start if loaded.
resolve	sym_warm_solve	re-solve the MILP problem after some modifications.
initialSolve	sym_solve	solve the MILP problem from scratch.
multiCriteriaBranchAndBound	sym_mc_solve	solve the multi criteria problem.
setInitialData	sym_set_defaults	set the parameters to their defaults.
parseCommandLine	sym_parse_command_line	read the command line arguments.
findInitialBounds	sym_find_initial_bounds	find the initial bounds via the user defined heuristics.
createPermanentCutPools	sym_create_permanent_cut_pools	save the global cuts.
loadProblem	sym_explicit_load_problem	load the problem through a set of arrays.
getWarmStart	sym_get_warm_start	get the warm start description.
setWarmStart	sym_set_warm_start	set the warm start description.
getLbForNewRhs	sym_get_lb_for_new_rhs	find a lower bound to the new rhs problem using the post solution info.
getUbForNewRhs	sym_get_lb_for_new_rhs	find an upper bound to the new rhs problem. using the post solution info.
getLbForNewObj	sym_get_lb_for_new_rhs	find a lower bound to the new obj problem. using the post solution info.
getUbForNewObj	sym_get_lb_for_new_rhs	find an upper bound to the new obj problem. using the post solution info.
reset	sym_close_environment	return the allocated memory.
setIntParam	sym_set_int_param	set the integer type OSI parameter.
setSymParam(int)	sym_set_int_param	set the integer type SYMPHONY parameter.
setDblParam	sym_set_dbl_param	set the double type OSI parameter.
setSymParam(double)	sym_set_dbl_param	set the double type SYMPHONY parameter.
setStrParam	sym_set_str_param	set the string type OSI parameter.
setSymParam(string)	sym_set_str_param	set the string type SYMPHONY parameter.
getIntParam	sym_get_int_param	get the value of the integer type OSI parameter.
getSymParam(int &)	sym_get_int_param	get the value of the integer type SYMPHONY parameter.
getDblParam	sym_get_dbl_param	get the value of the double type OSI parameter.
getSymParam(double &)	sym_get_dbl_param	get the value of the double type SYMPHONY parameter.
getStrParam	sym_get_str_param	get the value of the string type OSI parameter.
getSymParam(string &)	sym_get_str_param	get the value of the string type SYMPHONY parameter.
isProvenOptimal	sym_is_proven_optimal	query the problem status.
isProvenPrimalInfeasible	sym_is_proven_primal_infeasible	query the problem status.
isPrimalObjectiveLimitReached	sym_is_target_gap_achieved	query the problem status.
isIterationLimitReached	sym_is_iteration_limit_reached	query the problem status.
isTimeLimitReached	sym_is_time_limit_reached	query the problem status.
isTargetGapReached	sym_is_target_gap_achieved	query the problem status.
getNumCols	sym_get_num_cols	get the number of columns.
getNumRows	sym_get_num_rows	get the number of rows.
getNumElements	sym_get_num_elements	get the number of nonzero elements.
getColLower	sym_get_col_lower	get the column lower bounds.
getColUpper	sym_get_col_upper	get the column upper bounds.
getRowSense	sym_get_row_sense	get the row senses.
getRightHandSide	sym_get_rhs	get the rhs values.
getRowRange	sym_get_row_range	get the row range values.
getRowLower	sym_get_row_lower	get the row lower bounds.
getRowUpper	sym_get_row_upper	get the row upper bounds.
getObjCoefficients	sym_get_obj_coeff	get the objective function vector.

C++ Interface	C Interface	Description
getObjSense	sym_get_obj_sense	get the objective sense.
isContinuous	sym_is_continuous	query the variable type.
isBinary	sym_is_binary	query the variable type.
isInteger	sym_is_integer	query the variable type.
isIntegerNonBinary	-	query the variable type.
isFreeBinary	sym_is_binary	query the variable type.
getMatrixByRow	-	get the constraint matrix by row oriented.
getMatrixByCol	-	get the constraint matrix by column oriented.
getInfinity	-	get the infinity definition of SYMPHONY.
getColSolution	sym_get_col_solution	get the current best column solution.
getRowActivity	sym_get_row_activity	get the current row activity.
getObjValue	sym_get_obj_val	get the current best objective value.
getPrimalBound	sym_get_primal_bound	get the primal upper bound.
getIterationCount	sym_get_iteration_count	get the number of the analyzed tree nodes.
setObjCoeff	sym_set_obj_coeff	set the objective function vector.
setObj2Coeff	sym_set_obj2_coeff	set the second objective function vector.
setColLower	sym_set_col_lower	set the column lower bounds.
setColUpper	sym_set_col_upper	set the column upper bounds.
setRowLower	sym_set_row_lower	set the row lower bounds.
setRowUpper	sym_set_row_upper	set the row upper bounds.
setRowType	sym_set_row_type	set the row characteristics.
setObjSense	sym_set_obj_sense	set the objective sense.
setColSolution	sym_set_col_solution	set the current solution.
setContinuous	sym_set_continuous	set the variable type.
setInteger	sym_set_integer	set the variable type.
setColName	sym_set_col_names	set the column names.
addCol	sym_add_col	add columns to the constraint matrix.
addRow	sym_add_row	add rows to the constraint matrix.
deleteCols	sym_delete_cols	delete some columns from the constraint matrix.
deleteRows	sym_delete_rows	delete some rows from the constraint matrix.
writeMps	-	write the current problem in MPS format.
applyRowCut	-	add some row cuts.
applyColCut	-	add some column cuts.
SymWarmStart(warm_start_desc *)	sym_create_copy_warm_start	create a SYMPHONY warm start by copying the given one.
SymWarmStart(char *)	sym_read_warm_start	create a SYMPHONY warm start reading from file.
getCopyOfWarmStartDesc	sym_create_copy_warm_start	get the copy of the warm start structure.
writeToFile	sym_write_warm_start_desc	write the loaded warm start to a file.

7.3 User Callback API

7.3.1 Master module callbacks

user_usage

```
void user_usage()
```

Description:

SYMPHONY's command-line switches are all lower case letters. The user can use any upper case letter (except 'H' and as specified below) for command line switches to control user-defined parameter settings without the use of a parameter file. The function `user_usage()` can optionally print out usage information for the user-defined command line switches. The command line switch `-H` automatically calls the user's usage subroutine. The switch `-h` prints SYMPHONY's own usage information. In its default configuration, the command-line switch `-F` is used to specify the file in which the instance data is contained (either an MPS file or an GMPL/AMPL file). The `-D` switch is used to specify the data file if an GMPL/AMPL file is being read in (see the README file).

user_initialize

```
int user_initialize(void **user)
```

Description:

The user allocates space for and initializes the user-defined data structures for the master module.

Arguments:

`void **user`	OUT	Pointer to the user-defined data structure.

Return values:

`USER_ERROR`	Error. SYMPHONY exits.
`USER_SUCCESS`	Initialization is done.
`USER_DEFAULT`	There is no user-defined data structure (this can be the case if the default parser is being used to read in either an MPS or GMPL/AMPL file.

user_readparams

```
int user_readparams(void *user, char *filename, int argc, char **argv)
```

Description:

The user can optionally read in parameter settings from the file named `filename`, specified on the command line using the `-f` switch. The parameter file `filename` can contain both SYMPHONY's built-in parameters and user-defined parameters. If desired, the user can open this file for reading, scan the file for lines that contain user parameter settings and then read the parameter settings. A shell for doing this is set up in the in the file `SYMPHONY-5.0/USER/user_master.c`. Also, see the file `Master/master_io.c` to see how SYMPHONY does this.

The user can also parse the command line arguments for user settings. A shell for doing this is also set up in the file `SYMPHONY-5.0/USER/user_master.c`. Upper case letters are reserved for user-defined command line switches. The switch `-H` is reserved for help and calls the user's usage subroutine (see `user_usage()`7.3.1). If the user returns '`USER_DEFAULT`', then SYMPHONY will look for the command-line switch `-F` to specify the file name for reading in the model from either an MPS or a GMPL/AMPL file. The `-D` command-line switch is used to specify an additional data file for GMPL/AMPL models. If the `-D` option is not present, SYMPHONY assumes the file is an MPS file.

Arguments:

`void *user`	IN	Pointer to the user-defined data structure.
`char *filename`	IN	The name of the parameter file.

Return values:

`USER_ERROR`	Error. SYMPHONY stops.
`USER_SUCCESS`	User parameters were read successfully.
`USER_DEFAULT`	SYMPHONY will read in the problem instance from either an MPS or an GMPL/AMPL file. The command-line switches `-F` and `-D` will be used to specify the model file and (in the case of GMPL/AMPL) the data file.

user_io

```
int user_io(void *user)
```

Description:

Here, the user can read in an instance in a custom format and set up appropriate data structures. If the user wants to use the default parsers to read either an MPS file or a GMPL/AMPL file, then the return value `USER_DEFAULT` should be specified (see `user_readparams()`7.3.1 for the command-line switches to use to specify this behavior).

Arguments:

`void *user`	IN	Pointer to the user-defined data structure.

Return values:

`USER_ERROR`	Error. SYMPHONY stops.
`USER_SUCCESS`	User I/O was completed successfully.
`USER_DEFAULT`	Input will be read in from an MPS or GMPL/AMPL file.

user_init_draw_graph

```
int user_init_draw_graph(void *user, int dg_id)
```

Description:

This function is invoked only if the `do_draw_graph` parameter is set. The user can initialize the graph drawing module by sending some initial information (e.g., the location of the nodes of a graph, like in the TSP.)

Arguments:

`void *user`	IN	Pointer to the user-defined data structure.
`int dg_id`	IN	The process id of the graph drawing module.

Return values:

`USER_ERROR`	Error. SYMPHONY stops.
`USER_SUCCESS`	The user completed initialization successfully.
`USER_DEFAULT`	No action.

user_start_heurs

```
int user_start_heurs(void *user, double *ub, double *ub_estimate)
```

Description:

The user invokes heuristics and generates the initial global upper bound and also perhaps an upper bound estimate. This is the last place where the user can do things before the branch and cut algorithm starts. She might do some preprocessing, in addition to generating the upper bound.

Arguments:

`void *user`	IN	Pointer to the user-defined data structure.
`double *ub`	OUT	Pointer to the global upper bound. Initially, the upper bound is set to either `-MAXDOUBLE` or the bound read in from the parameter file, and should be changed by the user only if a better valid upper bound is found.
`double *ub_estimate`	OUT	Pointer to an estimate of the global upper bound. This is useful if the `BEST_ESTIMATE` diving strategy is used (see the treemanager parameter `diving_strategy` (Section 7.4.4))

Return values:

`USER_ERROR`	Error. This error is probably not fatal.
`USER_SUCCESS`	User executed function successfully.
`USER_DEFAULT`	No action (someday, there may be a default MIP heuristic here).

user_initialize_root_node

```
int user_initialize_root_node(void *user, int *basevarnum, int **basevars,
                              int *basecutnum, int *extravarnum,  int **extravars,
                              char *obj_sense, double *obj_offset,
                              char ***colnames, int *colgen_strat)
```

Description:

In this function, the user must specify the list of indices for the base and extra variables. The option to specify a variable as base is provided simply for efficiency reasons. If there is no reasonable way to come up with a set of base variables, then all variables should be specified as extra (see Section 4.1.2 for a discussion of base and extra variables). If the function returns `USER_DEFAULT` and sets `extravarnum`, then SYMPHONY will put all variables indexed from 0 to `extravarnum` in the set of extra variables by default. If an MPS or GMPL/AMPL file was read in using SYMPHONY's built-in parser, i.e., the default behavior of `user_io()`7.3.1 was not modified, then `extravarnum` need not be set.

In this function, the user may also specify column names for display purposes. If the `colnames` array is allocated, then SYMPHONY will use for displaying solutions. If the data was read in from either an MPS or GMPL/AMPL file, then the column names will be set automatically.

Arguments:

`void *user`	IN	Pointer to the user-defined data structure.
`int *basevarnum`	OUT	Pointer to the number of base variables.
`int **basevars`	OUT	Pointer to an array containing a list of user indices of the base variables to be active in the root.
`int *basecutnum`	OUT	The number of base constraints.
`int *extravarnum`	OUT	Pointer to the number of extra active variables in the root.
`int **extravars`	OUT	Pointer to an array containing a list of user indices of the extra variables to be active in the root.
`char *obj_sense`	INOUT	Whether to negate the objective function value when printing the solution, set to either `MAXIMIZE` or `MINIMIZE`. Note that SYMPHONY always minimizes—**this only effects the printing of the solution**. The default is `MINIMIZE`.
`double *obj_offset`	INOUT	A specified constant to be added to the objective function value when printing out the solution.
`int ***colnames`	OUT	Pointer to an array containing a list of column names to be used for display purposes.
`int *colgen_strat`	INOUT	The default strategy or one that has been read in from the parameter file is passed in, but the user is free to change it. See `colgen_strat` in the description of parameters for details on how to set it.

Return values:

`USER_ERROR`	Error. SYMPHONY stops.
`USER_SUCCESS`	The required data are filled in.
`USER_DEFAULT`	All variables indexed 0 to `extravarnum` are put in the extra set (The user must set `extravarnum` unless an MPS or GMPL/AMPL file was read in by SYMPHONY.

Post-processing:

The array of base and extra indices are sorted.

user_receive_feasible_solution

```
int user_receive_feasible_solution(void *user, int msgtag, double cost,
                                   int numvars, int *indices, double *values)
```

Description:

This function is only used for parallel execution. Feasible solutions can be sent and/or stored in a user-defined packed form if desired. For instance, the TSP, a tour can be specified simply as a permutation, rather than as a list of variable indices. In the LP module, a feasible solution is packed either by the user or by a default packing routine. If the default packing routine was used, the `msgtag` will be `FEASIBLE_SOLUTION_NONZEROS`. In this case, `cost`, `numvars`, `indices` and `values` will contain the solution value, the number of nonzeros in the feasible solution, and their user indices and values. The user has only to interpret and store the solution. Otherwise, when `msgtag` is `FEASIBLE_SOLUTION_USER`, SYMPHONY will send and receive the solution value only and the user has to unpack exactly what she has packed in the LP module. In this case the contents of the last three arguments are undefined.

In most cases, SYMPHONY's default routines for sending and receiving feasible solutions, as well as displaying them, will suffice. These routines simply display all nonzeros by either index or name, depending on whether the user set the column names. See `user_receive_lp_data()` in Section 7.3.2 for more discussion.

Arguments:

`void *user`	IN	Pointer to the user-defined data structure.
`int msgtag`	IN	`FEASIBLE_SOLUTION_NONZEROS` or `FEASIBLE_SOLUTION_USER`
`double cost`	IN	The cost of the feasible solution.
`int numvars`	IN	The number of variables whose user indices and values were sent (length of `indices` and `values`).
`int *indices`	IN	The user indices of the nonzero variables.
`double *values`	IN	The corresponding values.

Return values:

`USER_ERROR`	Ignored. This is probably not a fatal error.
`USER_SUCCESS`	The solution has been unpacked and stored.
`USER_DEFAULT`	Store the nonzeros in the solutions for later display.

user_send_lp_data

```
int user_send_lp_data(void *user, void **user_lp)
```

Description:

If the user wishes to employ parallelism, she has to send all problem-specific data that will be needed to implement user functions in the LP module in order to set up the initial LP relaxation and perform later computations. This could include instance data, as well as user parameter settings (see Section 6.4 for a discussion of this). This is one of the few places where the user may need to worry about the configuration of the modules. If either the tree manager or the LP are running as a separate process (either `COMPILE_IN_LP` or `COMPILE_IN_TM` are `FALSE` in the make file), then the data will be sent and received through message-passing. See `user_receive_lp_data()` in Section 7.3.2 for more discussion. Otherwise, it can be copied through shared memory. The easiest solution, which is set up by default is to simply copy over a pointer to a single user data structure where instance data is stored. The code for the two cases is put in the same source file by use of `#ifdef` statements. See the comments in the code stub for this function for more details.

Arguments:

`void *user`	IN	Pointer to the user-defined data structure.
`void **user_lp`	OUT	Pointer to the user-defined data structure for the LP module.

Return values:

`USER_ERROR`	Error. SYMPHONY stops.
`USER_SUCCESS`	Packing is done.
`USER_DEFAULT`	User has no data to send. This would be used when SYMPHONY has read in an MPS or GMPL/AMPL model file.

user_send_cg_data

```
int user_pack_cg_data(void *user, void **user_cg)
```

Description:

If the user wishes to employ parallelism and wants a separate cut generator module, this function can be used to send all problem-specific data that will be needed by the cut generator module to perform separation. This could include instance data, as well as user parameter settings (see Section 6.4 for a discussion of this). This is one of the few places where the user may need to worry about the configuration of the modules. If either the tree manager or the LP are running as a separate process (either `COMPILE_IN_LP` or `COMPILE_IN_TM` are `FALSE` in the `make` file), then the data will be sent and received through message-passing. See `user_receive_cg_data()` in Section 7.3.3 for more discussion. Otherwise, it can be copied through shared memory. The easiest solution, which is set up by default is to simply copy over a pointer to a single user data structure where instance data is stored. The code for the two cases is put in the same source file by use of `#ifdef` statements. See the comments in the code stub for this function for more details.

Arguments:

`void *user`	IN	Pointer to the user-defined data structure.
`void **user_cg`	OUT	Pointer to the user-defined data structure for the cut generator module.

Return values:

`USER_ERROR`	Error. SYMPHONY stops.
`USER_SUCCESS`	Packing is done.
`USER_DEFAULT`	No data to send to the cut generator (no separation performed).

user_send_cp_data

```
int user_pack_cp_data(void *user, void **user_cp)
```

Description:

If the user wishes to employ parallelism and wants to use the cut pool to store user-defined cuts, this function can be used to send all problem-specific data that will be needed by the cut pool module. This could include instance data, as well as user parameter settings (see Section 6.4 for a discussion of this). This is one of the few places where the user may need to worry about the configuration of the modules. If either the tree manager or the LP are running as a separate process (either `COMPILE_IN_LP` or `COMPILE_IN_TM` are `FALSE` in the `make` file), then the data will be sent and received through message-passing. See `user_receive_cp_data()` in Section 7.3.4 for more discussion. Otherwise, it can be copied through shared memory. The easiest solution, which is set up by default is to simply copy over a pointer to a single user data structure where instance data is stored. The code for the two cases is put in the same source file by use of `#ifdef` statements. See the comments in the code stub for this function for more details.

Note that there is support for cuts generated and stored as explicit matrix rows. The cut pool module is already configured to deal with such cuts, so no user implementation is required. Only the use of user-defined cuts requires customization of the Cut Pool module.

Arguments:

`void *user`	IN	Pointer to the user-defined data structure.
`void **user_cp`	OUT	Pointer to the user-defined data structure for the cut pool module.

Return values:

`USER_ERROR`	Error. SYMPHONY stops.
`USER_SUCCESS`	Packing is done.
`USER_DEFAULT`	No data to send to the cut pool (no user-defined cut classes or cut pool not used).

user_display_solution

```
int user_display_solution(void *user, double lpetol, int varnum, int *indices,
                          double *values, double objval)
```

Description:

This function is invoked when a best solution found so far is to be displayed (after heuristics, after the end of the first phase, or the end of the whole algorithm). This can be done using either a text-based format or using the `drawgraph` module. By default, SYMPHONY displays the indices (or column names, if specified) and values for each nonzero variable in the solution. The user may wish to display a custom version of the solution by interpreting the variables.

Arguments:

`void *user`	IN	Pointer to the user-defined data structure. For sequential computation, a pointer to the user's LP data structure is passed in. For parallel computation, a pointer to the user's Master data structure is passed in.
`double lpetol`	IN	The LP zero tolerance used.
`int varnum`	IN	The number of nonzeros in the solution.
`int *indices`	IN	The indices of the nonzeros.
`double *values`	IN	The values of the nonzeros.
`double objval`	IN	The objective function value of the solution.

Return values:

`USER_ERROR`	Ignored.
`USER_SUCCESS`	User displayed the solution. SYMPHONY should do nothing.
`USER_DEFAULT`	SYMPHONY should display the solution in default format.

Post-processing:

If requested, SYMPHONY will display a best solution found so far in the default format.

user_send_feas_sol

```
int user_send_feas_sol(void *user, int *feas_sol_size, int **feas_sol)
```

Description:

This function is useful for debugging purposes. It passes a known feasible solution to the tree manager. The tree manager then tracks which current subproblem admits this feasible solution and notifies the user when it gets pruned. It is useful for finding out why a known optimal solution never gets discovered. Usually, this is due to either an invalid cut of an invalid branching. Note that this feature only works when branching on binary variables. See Section 6.8.4 for more on how to use this feature.

Arguments:

`void *user`	IN	Pointer to the user-defined data structure.
`int *feas_sol_size`	INOUT	Pointer to size of the feasible solution passed by the user.
`int **feas_sol`	INOUT	Pointer to the array of user indices containing the feasible solution. This array is simply copied by the tree manager and must be freed by the user.

Return values:

Arguments:

`USER_ERROR`	Solution tracing is not enabled.
`USER_SUCCESS`	Tracing of the given solution is enabled.
`USER_DEFAULT`	No feasible solution given.

user_process_own_messages

```
int user_process_own_messages(void *user, int msgtag)
```

Description:

The user must receive any message he sends to the master module (independently of SYMPHONY's own messages). An example for such a message is sending feasible solutions from separate heuristics processes fired up in `user_start_heurs()`.

Arguments:

`void *user`	`IN`	`Pointer to the user-defined data structure.`
`int msgtag`	`IN`	`The message tag of the message.`

Return values:

`USER_ERROR`	`Ignored.`
`USER_SUCCESS`	`Message is processed.`
`USER_DEFAULT`	`No user message types defined.`

user_free_master

```
int user_free_master(void **user)
```

Description:

The user frees all the data structures within `*user`, and also free `*user` itself. This can be done using the built-in macro `FREE` that checks the existence of a pointer before freeing it.

Arguments:

`void **user`	INOUT	Pointer to the user-defined data structure (should be `NULL` on return).

Return values:

`USER_ERROR`	Ignored. This is probably not a fatal error.
`USER_SUCCESS`	Everything was freed successfully.
`USER_DEFAULT`	There is no user memory to free.

7.3.2 LP module callbacks

Data Structures

We first describe a few structures that are used to pass data into and out of the user functions of the LP module.

MIPdesc

One of the few internally defined data structures that the user has to deal with frequently is the `MIPdesc` data structure, which holds the data needed to describe a MILP. This data structure is used by SYMPHONY for two purposes. First, it is used to store the description of a generic MILP that is either read in from an MPS or AMPL file. More importantly, it is the data structure the user must use to pass the subproblem descriptions back to SYMPHONY at the time a new search tree node is created in the function `user_create_subproblem()` (see Section 7.3.2). The structure has 14 fields (listed below) that must be filled out to describe a subproblem (some fields are optional).

A subproblem is a mixed-integer program defined by a matrix of constraints, an objective function, bounds on both the right hand side values of the constraints and the variables, an array indicating which variables are integer, and (optionally) an array of column names that allows SYMPHONY to report the solution back in terms of column names instead of user indices. If the subproblem has `n` variables and `m` constraints, the constraints are given by a constraint coefficient matrix of size `m` × `n` (described in the next paragraph). The sense of each constraint, the right hand side values and bounds on the right hand side (called *range*) are vectors are of size `m`. The objective function coefficients and the lower and upper bounds on the variables are vectors of length `n`. The sense of each constraint can be either 'L' (≤), 'E' (=), 'G' (≥) or 'R' (ranged). For non-ranged rows the range value is `0`, for a ranged row the range value must be non-negative and the constraint means that the row activity level has to be between the right hand side value and the right hand side increased by the range value.

Since the coefficient matrix is very often sparse, only the nonzero entries are stored. Each entry of the matrix has a column index, a row index and a coefficient value associated with it. A matrix is specified in the form of the three arrays `matval`, `matind`, and `matbeg`. The array `matval` contains the values of the nonzero entries of the matrix in *column order*; that is, all the entries for the 0^th column come first, then the entries for the 1^st column, etc. The row index corresponding to each entry of `matval` is listed in `matind` (both of them are of length `nz`, the number of nonzero entries in the matrix). Finally, `matbeg` contains the starting positions of each of the columns in `matval` and `matind`. Thus, `matbeg[i]` is the position of the first entry of column `i` in both `matval` and `matind`). By convention `matbeg` is allocated to be of length `n+1`, with `matbeg[n]` containing the position after the very last entry in `matval` and `matind` (so it is very conveniently equal to `nz`). This representation of a matrix is known as a *column ordered* or *column major* representation. Below are listed the fields that must be filled out to describe a subproblem.

`int n` – The number of columns.

`int m` – The number of rows.

`int nz` – The number of nonzeros.

`double obj_offset` – Constant to be added to the objective function value when printed.

`char obj_sense` – Objective sense (set to `MAXIMIZE` or `MINIMIZE`).

`char *is_int` – Indicates which variables are required to be integer.

`int *matbeg` – The array containing the starting positions for each column.

`int *matind` – The array containing the indices for each column.

`double *matval` – The array containing the matrix values for each column.

`double *obj` – The objective function coefficients for the second objective (for multicriteria solve).

`double *obj2` – The objective function coefficients.

`double *rhs` – The right hand side values for the constraints.

`double *rngval` – The range values for the constraints (optional).

`char *sense` – The senses of the constraints.

`double *lb` – The lower bounds of the variables.

`double *ub` – The upper bounds of the variables.

`char **colname` – The column names.

cut_data

Another of the internally defined data structures that the user has to deal with frequently is the `cut_data` data structure, used to store the packed form of cuts. This structure has 8 fields listed below.

`int size` – The size of the `coef` array.

`char *coef` – An array containing the packed form of the cut, which is defined and constructed by the user. Given this packed form and a list of the variables active in the current relaxation, the user must be able to construct the corresponding constraint.

`double rhs` – The right hand side of the constraint.

`double range` – The range of the constraint. It is zero for a standard form constraint. Otherwise, the row activity level is limited to between `rhs` and `rhs` + `range`.

`char type` – A user-defined type identifier that represents the general class that the cut belongs to.

`char sense` – The sense of the constraint. Can be either 'L' (≤), 'E' (=), 'G' (≥) or 'R' (ranged). This may be evident from the `type`.

`char deletable` – Determines whether or not a cut can be deleted once added to the formulation. `TRUE` by default.

`char branch` – Determines whether the cut can be branched on or not. Possible initial values are `DO_NOT_BRANCH_ON_THIS_ROW` and `ALLOWED_TO_BRANCH_ON`.

`int name` – Identifier used by SYMPHONY. The user should not set this.

waiting_row

A closely related data structure is the `waiting_row`, essentially the "unpacked" form of a cut. There are six fields.

`source_pid` – Used internally by SYMPHONY.

`cut_data *cut` – Pointer to the cut from which the row was generated.

`int nzcnt, *matind, *matval` – Fields describing the row. `nzcnt` is the number of nonzeros in the row, i.e., the length of the `matind` and `matval` arrays, which are the variable indices (wrt. the current LP relaxation) and nonzero coefficients in the row.

`double violation` – If the constraint corresponding to the cut is violated, this value contains the degree of violation (the absolute value of the difference between the row activity level (i.e., lhs) and the right hand side). This value does not have to be set by the user.

var_desc

The `var_desc` structure is used list the variables in the current relaxation. There are four fields.

`int userind` – The user index of the variables,

`int colind` – The column index of the variables (in the current relaxation),

`double lb` – The lower bound of the variable,

`double ub` – The upper bound of the variable.

Function Descriptions

Now we describe the functions themselves.

user_receive_lp_data

```
int user_receive_lp_data (void **user)
```

Description:

This function only has to be filled out for parallel execution and only if either the TM or LP modules are configured as separate processes. Otherwise, data will have been copied into appropriate locations in the master function `user_send_lp_data()` (see Section 7.3.1). The two cases can be handled by means of `#ifdef` statements. See comments in the source code stubs for more details.

Here, the user must receive all problem-specific information sent from the master, set up necessary data structures, etc. Note that the data must be received in exactly the same order as it was sent in `user_send_lp_data()` (see Section 7.3.1). See Section 6.4 for more notes on receiving data.

Arguments:

`void **user`	OUT	Pointer to the user-defined LP data structure.

Return values:

`USER_ERROR`	Error. SYMPHONY aborts this LP module.
`USER_SUCCESS`	User received the data successfully.
`USER_DEFAULT`	User did not send any data.

Wrapper invoked from: `lp_initialize()` at process start.

user_create_subproblem

```
int user_create_subproblem(void *user, int *indices, MIPdesc *mip,
                           int *maxn, int *maxm, int *maxnz)
```

Description:

Based on the instance data contained in the user data structure and the list of base and extra variables that are active in the current subproblem, the user has to create the subproblem for the search node. The matrix describing the subproblem must contain those variables whose user indices are listed in the array `indices` (in the same order) and the base constraints. The extra (dynamically generated) constraints are added automatically by SYMPHONY after the initial subproblem description is created.

In this function, the user is required to construct a description of the subproblem in column-ordered format and pass it back to SYMPHONY by filling out the `MIPdesc` data structure, described in Section 7.3.2. The user is not responsible for allocating extra memory to allow for the addition of dynamically generated cuts and variables. The arrays allocated in `user_create_subproblem()` are owned by SYMPHONY after allocation and are freed as soon as the relaxation is loaded into the solver. However, if the user has an idea as to the maximum number of variables and constraints that are likely to be generated during processing of the subproblem, this information can be passed to SYMPHONY in the variables `*maxn`, `*maxm`, and `*maxnz`. These numbers are only estimates that SYMPHONY can use to perform memory allocation. They do not have to be exact numbers. In fact, these estimates need not be provided at all. The `obj_sense` and `obj_offset` fields are set globally in the function `user_initialize_root_node()` (see Section7.3.1). Setting them here will have no effect.

Note that, the user should return "USER_DEFAULT" if an MPS or GMPL/AMPL file was read in to describe the original MILP. SYMPHONY will allocate the corresponding arrays and specify the constraint matrix automatically in this case.

Arguments:

`void *user`	IN	Pointer to the user-defined LP data structure.
`int *indices`	IN	The list of the active variables (base and extra).
`MIPdesc *mip`	OUT	Pointer to the MIPdesc data structure.
`int *maxn`	OUT	Estimated maximum number of variables.
`int *maxm`	OUT	Estimated maximum number of constraints.
`int *maxnz`	OUT	Estimated maximum number of nonzeros.

Return values:

`USER_ERROR`	Error. The LP module is aborted.
`USER_SUCCESS`	User created the constraint matrix with the base constraints.
`USER_DEFAULT`	This return code is used when the default routines for reading in an MPS or AMPL file were used and the user wants to let SYMPHONY set up the subproblem automatically. This return will cause an error if the default I/O routines were not used.

Post-processing:

The extra constraints are added to the matrix by calling the `user_unpack_cuts()` subroutine and then adding the corresponding rows to the matrix. This is easier for the user to implement, but less efficient than adding the cuts at the time the original matrix was being constructed.

Wrapper invoked from: `process_chain()` which is invoked when setting up a the initial search node in a chain.

user_is_feasible

```
int user_is_feasible(void *user, double lpetol, int varnum, int
                     *indices, double *values, int *feasible,
                     double *objval)
```

Description:

User tests the feasibility of the solution to the current LP relaxation. There is no post-processing. Possible defaults are testing integrality (TEST_INTEGRALITY) and testing whether the solution is binary (TEST_ZERO_ONE).

Arguments:

void *user	INOUT	Pointer to the user-defined LP data structure.
double lpetol	IN	The ϵ tolerance of the LP solver.
int varnum	IN	The length of the indices and values arrays.
int *indices	IN	User indices of variables at nonzero level in the current solution.
double *values	IN	Values of the variables listed in indices.
int *feasible	OUT	Feasibility status of the solution (NOT_FEASIBLE, or FEASIBLE).
double *objval	OUT	The user can return the "true" objective function value of the solution in the case it is feasible, i.e., eliminating the round-off error.

Return values:

USER_ERROR	Error. Solution is considered to be not feasible.
USER_SUCCESS	User checked IP feasibility.
USER_DEFAULT	Regulated by the parameter is_feasible_default, but set to TEST_INTEGRALITY unless overridden by the user.
TEST_INTEGRALITY	Test integrality of the given solution.
TEST_ZERO_ONE	Tests whether the solution is binary.

Wrapper invoked from: select_branching_object() after pre-solving the LP relaxation of a child corresponding to a candidate and from fathom_branch() after solving an LP relaxation.

user_send_feasible_solution

```
int user_send_feasible_solution(void *user, double lpetol,
                                int varnum, int *indices, double *values)
```

Description:

This function is only used for parallel computation. The user can send a feasible solution in custom format to the master module if desired. However, the default routine suffices in most situations. The solution is sent using the communication functions described in Section 6.4 in whatever logical format the user wants to use. The default is to pack the user indices and values of variables at non-zero level. If the user packs the solution herself then the same data must be packed here that will be received in the `user_receive_feasible_solution()` function in the master module. See the description of that function for details. This function will only be called when either the LP or tree manager are running as a separate executable. Otherwise, the solution gets stored within the LP user data structure.

Arguments:

`void *user`	IN	Pointer to the user-defined LP data structure.
`double lpetol`	IN	The ϵ tolerance of the LP solver.
`int varnum`	IN	The length of the `indices` and `values` arrays.
`int *indices`	IN	User indices of variables at nonzero level in the current solution.
`double *values`	IN	Values of the variables listed in `indices`.

Return values:

`USER_ERROR`	Error. Do the default.
`USER_SUCCESS`	User packed the solution.
`USER_DEFAULT`	Regulated by the parameter `pack_feasible_solution_default`, but set to `SEND_NONZEROS` unless overridden by the user.
`SEND_NONZEROS`	Pack the nonzero values and their indices.

Wrapper invoked: as soon as feasibility is detected anywhere.

user_display_lp_solution

```
int user_display_lp_solution(void *user, int which_sol,
                             int varnum, int *indices, double *values)
```

Description:

Given a solution to an LP relaxation (the indices and values of the nonzero variables) the user can (graphically) display it. The `which_sol` argument shows what kind of solution is passed to the function: `DISP_FEAS_SOLUTION` indicates a solution feasible to the original IP problem, `DISP_RELAXED_SOLUTION` indicates the solution to any LP relaxation and `DISP_FINAL_RELAXED_SOLUTION` indicates the solution to an LP relaxation when no cut has been found. There is no post-processing. Default options print out user indices and values of nonzero or fractional variables on the standard output.

Arguments:

`void *user`	IN	Pointer to the user-defined LP data structure.
`int which_sol`	IN	The type of solution passed on to the displaying function. Possible values are `DISP_FEAS_SOLUTION`, `DISP_RELAXED_SOLUTION` and `DISP_FINAL_RELAXED_SOLUTION`.
`int varnum`	IN	The number of variables in the current solution at nonzero level (the length of the `indices` and `values` arrays).
`int *indices`	IN	User indices of variables at nonzero level in the current solution.
`double *values`	IN	Values of the nonzero variables.

Return values:

`USER_ERROR`	Error. SYMPHONY ignores error message.
`USER_SUCCESS`	User displayed whatever she wanted to.
`USER_DEFAULT`	Regulated by the parameter `display_solution_default`, but set to `DISP_NZ_INT` unless overridden by the user.
`DISP_NOTHING`	Display nothing.
`DISP_NZ_INT`	Display user indices (as integers) and values of nonzero variables.
`DISP_NZ_HEXA`	Display user indices (as hexadecimals) and values of nonzero variables.
`DISP_FRAC_INT`	Display user indices (as integers) and values of variables not at their lower or upper bounds.
`DISP_FRAC_HEXA`	Display user indices (as hexadecimals) and values of variables not at their lower and upper bounds.

Wrapper invoked from: `fathom_branch()` with `DISP_FEAS_SOLUTION` or `DISP_RELAXED_SOLUTION` after solving an LP relaxation and checking its feasibility status. If it was not feasible and no cut could be added either then the wrapper is invoked once more, now with `DISP_FINAL_RELAXED_SOLUTION`.

user_shall_we_branch

```
int user_shall_we_branch(void *user, double lpetol, int cutnum,
                         int slacks_in_matrix_num,
                         cut_data **slacks_in_matrix,
                         int slack_cut_num, cut_data **slack_cuts,
                         int varnum, var_desc **vars, double *x,
                         char *status, int *cand_num,
                         branch_obj ***candidates, int *action)
```

Description:

There are two user-written functions invoked from `select_candidates_u`. The first one (`user_shall_we_branch()`) decides whether to branch at all, the second one (`user_select_candidates()`) chooses the branching objects. The argument lists of the two functions are the same, and if branching occurs (see discussion below) then the contents of `*cand_num` and `*candidates` will not change between the calls to the two functions.

The first of these two functions is invoked in each iteration after solving the LP relaxation and (possibly) generating cuts. Therefore, by the time it is called, some violated cuts might be known. Still, the user might decide to branch anyway. The second function is invoked only when branching is decided on.

Given (1) the number of known violated cuts that can be added to the problem when this function is invoked, (2) the constraints that are slack in the LP relaxation, (3) the slack cuts not in the matrix that could be branched on (more on this later), and (4) the solution to the current LP relaxation, the user must decide whether to branch or not. Branching can be done either on variables or slack cuts. A pool of slack cuts which has been removed from the problem and kept for possible branching is passed to the user. If any of these happen to actually be violated (it is up to the user to determine this), they can be passed back as branching candidate type `VIOLATED_SLACK` and will be added into the current relaxation. In this case, branching does not have to occur (the structure of the `*candidates` array is described below in `user_select_candidates()`).

This function has two outputs. The first output is `*action` which can take four values: `USER__DO_BRANCH` if the user wants to branch, `USER__DO_NOT_BRANCH` if he doesn't want to branch, `USER__BRANCH_IF_MUST` if he wants to branch only if there are no known violated cuts, or finally `USER__BRANCH_IF_TAILOFF` if he wants to branch in case tailing off is detected. The second output is the number of candidates and their description. In this function the only sensible "candidates" are `VIOLATED_SLACK`s.

There is no post processing, but in case branching is selected, the `col_gen_before_branch()` function is invoked before the branching would take place. If that function finds dual infeasible variables then (instead of branching) they are added to the LP relaxation and the problem is resolved. (Note that the behavior of the `col_gen_before_branch()` is governed by the `colgen_strat[]` TM parameters.)

Arguments:

`void *user`	IN	Pointer to the user-defined LP data structure.
`double lpetol`	IN	The ϵ tolerance of the LP solver.
`int cutnum`	IN	The number of violated cuts (known before invoking this function) that could be added to the problem (instead of branching).
`int slacks_in_matrix_num`	IN	Number of slack constraints in the matrix.
`cut_data **slacks_in_matrix`	IN	The description of the cuts corresponding to these constraints (see Section 7.3.2).
`int slack_cut_num`	IN	The number of slack cuts not in the matrix.
`cut_data **slack_cuts`	IN	Array of pointers to these cuts (see Section 7.3.2).
`int varnum`	IN	The number of variables in the current lp relaxation (the length of the following three arrays).
`var_desc **vars`	IN	Description of the variables in the relaxation.
`double *x`	IN	The corresponding solution values (in the optimal solution to the relaxation).
`char *status`	IN	The stati of the variables. There are five possible status values: `NOT_FIXED`, `TEMP_FIXED_TO_UB`, `PERM_FIXED_TO_UB`, `TEMP_FIXED_TO_LB` and `PERM_FIXED_TO_LB`.
`int *cand_num`	OUT	Pointer to the number of candidates returned (the length of `*candidates`).
`candidate ***candidates`	OUT	Pointer to the array of candidates generated (see description below).
`int *action`	OUT	What to do. Must be one of the four above described values unless the return code is `USER_DEFAULT`.

Return values:

`USER_ERROR`	Error. `DEFAULT` is used.
`USER_SUCCESS`	The user filled out `*action` (and possibly `*cand_num` and `*candidates`).
`USER_DEFAULT`	Action taken is controlled by the parameter `shall_we_branch_default`, which is initially `USER__BRANCH_IF_MUST` unless overridden by the user.

Notes:

- The user has to allocate the pointer array for the candidates and place the pointer for the array into `***candidates` (if candidates are returned).
- Candidates of type `VIOLATED_SLACK` are always added to the LP relaxation regardless of what `action` is chosen and whether branching will be carried out or not.
- Also note that the user can change his mind in `user_select_candidates()` and not branch after all, even if she chose to branch in this function. A possible scenario: `cut_num` is zero when this function is invoked and the user asks for

USER__BRANCH_IF_MUST without checking the slack constraints and slack cuts. Afterward no columns are generated (no dual infeasible variables found) and thus SYMPHONY decides branching is called for and invokes user_select_candidates(). However, in that function the user checks the slack cuts, finds that some are violated, cancels the branching request and adds the violated cuts to the relaxation instead.

Warning: The cuts the user unpacks and wants to be added to the problem (either because they are of type VIOLATED_SLACK or type CANDIDATE_CUT_NOT_IN_MATRIX) will be deleted from the list of slack cuts after this routine returns. Therefore the same warning applies here as in the function user_unpack_cuts().

Wrapper invoked from: select_branching_object().

user_select_candidates

```
int user_select_candidates(void *user, double lpetol, int cutnum,
                           int slacks_in_matrix_num,
                           cut_data **slacks_in_matrix,
                           int slack_cut_num, cut_data **slack_cuts,
                           int varnum, var_desc **vars, double *x,
                           char *status, int *cand_num,
                           branch_obj ***candidates, int *action,
                           int bc_level)
```

Description:

The purpose of this function is to generate branching candidates. Note that `*action` from `user_shall_we_branch()` is passed on to this function (but its value can be changed here, see notes at the previous function), as well as the candidates in `**candidates` and their number in `*cand_num` if there were any.

Violated cuts found among the slack cuts (not in the matrix) can be added to the candidate list. These violated cuts will be added to the LP relaxation regardless of the value of `*action`.

The `branch_obj` structure contains fields similar to the `cut_data` data structure. Branching is accomplished by imposing inequalities which divide the current subproblem while cutting off the corresponding fractional solution. Branching on cuts and variables is treated symmetrically and branching on a variable can be thought of as imposing a constraint with a single unit entry in the appropriate column. Following is a list of the fields of the `branch_obj` data structure which must be set by the user.

`char type` Can take five values:

- `CANDIDATE_VARIABLE` The object is a variable.
- `CANDIDATE_CUT_IN_MATRIX` The object is a cut (it must be slack) which is in the current formulation.
- `CANDIDATE_CUT_NOT_IN_MATRIX` The object is a cut (it must be slack) which has been deleted from the formulation and is listed among the slack cuts.
- `VIOLATED_SLACK` The object is not offered as a candidate for branching, but rather it is selected because it was among the slack cuts but became violated again.
- `SLACK_TO_BE_DISCARDED` The object is not selected as a candidate for branching rather it is selected because it is a slack cut which should be discarded even from the list of slack cuts.

`int position` The position of the object in the appropriate array (which is one of `vars`, `slacks_in_matrix`, or `slack_cuts`.

`waiting_row *row` Used only if the type is `CANDIDATE_CUT_NOT_IN_MATRIX` or `VIOLATED_SLACK`. In these cases this field holds the row extension corresponding to the cut. This structure can be filled out easily using a call to `user_unpack_cuts()`.

`int child_num`
The number of children of this branching object.

`char *sense, double *rhs, double *range, int *branch`
The description of the children. These arrays determine the sense, rhs, etc. for the cut to be imposed in each of the children. These are defined and used exactly as in the `cut_data` data structure. **Note:** If a limit is defined on the number of children by defining the `MAX_CHILDREN_NUM` macro to be a number (it is pre-defined to be 4 as a default), then these arrays will be statically defined to be the correct length and don't have to be allocated. This option is highly recommended. Otherwise, the user must allocate them to be of length `child_num`.

`double lhs` The activity level for the row (for branching cuts). This field is purely for the user's convenience. SYMPHONY doesn't use it so it need not be filled out.

`double *objval, int *termcode, int *iterd, int *feasible`
The objective values, termination codes, number of iterations and feasibility stati of the children after pre-solving them. These are all filed out by SYMPHONY during strong branching. The user may access them in `user_compare_candidates()` (see below).

There are three default options (see below), each chooses a few variables (the number is determined by the strong branching parameters (see Section 7.4.5).

Arguments:

Same as for `user_shall_we_branch(), except that *action must be either USER__DO_BRANCH or USER__DO_NOT_BRANCH, and if branching is asked for, there must be a real candidate in the candidate list (not only VIOLATED_SLACKs and SLACK_TO_BE_DISCARDEDs). Also, the argument bc_level is the level in the tree. This could be used in deciding how many strong branching candidates to use.`

Return values:

`USER_ERROR`	`Error. DEFAULT is used.`
`USER_SUCCESS`	`User generated branching candidates.`
`USER_DEFAULT`	`Regulated by the select_candidates_default parameter, but set to USER__CLOSE_TO_HALF unless overridden by the user.`
`USER__CLOSE_TO_HALF`	`Choose variables with values closest to half.`
`USER__CLOSE_TO_HALF_AND_EXPENSIVE`	`Choose variables with values close to half and with high objective function coefficients.`
`USER__CLOSE_TO_ONE_AND_CHEAP`	`Choose variables with values close to one and with low objective function coefficients.`

Wrapper invoked from: `select_branching_object()`.

Notes: `See the notes at user_shall_we_branch().`

user_compare_candidates

```
int user_compare_candidates(void *user, branch_obj *can1, branch_obj *can2,
                            double ub, double granularity,
                            int *which_is_better)
```

Description:

By the time this function is invoked, the children of the current search tree node corresponding to each branching candidate have been pre-solved, i.e., the `objval`, `termcode`, `iterd`, and `feasible` fields of the `can1` and `can2` structures are filled out. Note that if the termination code for a child is `LP_D_UNBOUNDED` or `LP_D_OBJLIM`, i.e., the dual problem is unbounded or the objective limit is reached, then the objective value of that child is set to `MAXDOUBLE / 2`. Similarly, if the termination code is one of `LP_D_ITLIM` (iteration limit reached), `LP_D_INFEASIBLE` (dual infeasible) or `LP_ABANDONED` (because of numerical difficulties) then the objective value of that child is set to that of the parent's.

Based on this information the user must choose which candidate he considers better and whether to branch on this better one immediately without checking the remaining candidates. As such, there are four possible answers: `FIRST_CANDIDATE_BETTER`, `SECOND_CANDIDATE_BETTER`, `FIRST_CANDIDATE_BETTER_AND_BRANCH_ON_IT` and `SECOND_CANDIDATE_BETTER_AND_BRANCH_ON_IT`. An answer ending with `_AND_BRANCH_ON_IT` indicates that the user wants to terminate the strong branching process and select that particular candidate for branching.

There are several default options. In each of them, objective values of the pre-solved LP relaxations are compared.

Arguments:

`void *user`	IN	Pointer to the user-defined LP data structure.
`branch_obj *can1`	IN	One of the candidates to be compared.
`branch_obj *can2`	IN	The other candidate to be compared.
`double ub`	IN	The current best upper bound.
`double granularity`	IN	Defined tolerance
`int *which_is_better`	OUT	The user's choice. See the description above.

Return values:

`USER_ERROR`	Error. `DEFAULT` is used.
`USER_SUCCESS`	User filled out `*which_is_better`.
`USER_DEFAULT`	Regulated by the `compare_candidates_default` parameter, initially set to `LOWEST_LOW_OBJ` unless overridden by the user.
`BIGGEST_DIFFERENCE_OBJ`	Prefer the candidate with the biggest difference between highest and lowest objective function values.
`LOWEST_LOW_OBJ`	Prefer the candidate with the lowest minimum objective function value. The minimum is taken over the objective function values of all the children.
`HIGHEST_LOW_OBJ`	Prefer the candidate with the highest minimum objective function value.
`LOWEST_HIGH_OBJ`	Prefer the candidate with the lowest maximum objective function value.
`HIGHEST_HIGH_OBJ`	Prefer the candidate with the highest maximum objective function value .
`LOWEST_LOW_FRAC`	Prefer the candidate with the lowest minimum number of fractional variables. The minimum is taken over the number of fractional variables in all the children. Fractional branching options are only available if the fractional branching compile-time option is set in the makefile.
`HIGHEST_LOW_FRAC`	Prefer the candidate with the highest minimum number of fractional variables.
`LOWEST_HIGH_FRAC`	Prefer the candidate with the lowest maximum number of fractional variables.
`HIGHEST_HIGH_FRAC`	Prefer the candidate with the highest maximum number of fractional variables.

Wrapper invoked from: `select_branching_object()` after the LP relaxations of the children have been pre-solved.

user_select_child

```
int user_select_child(void *user, double ub, branch_obj *can, char *action)
```

Description:

By the time this function is invoked, the candidate for branching has been chosen. Based on this information and the current best upper bound, the user has to decide what to do with each child. Possible actions for a child are `KEEP_THIS_CHILD` (the child will be kept at this LP for further processing, i.e., the process *dives* into that child), `PRUNE_THIS_CHILD` (the child will be pruned based on some problem specific property—no questions asked...), `PRUNE_THIS_CHILD_FATHOMABLE` (the child will be pruned based on its pre-solved LP relaxation) and `RETURN_THIS_CHILD` (the child will be sent back to tree manager). Note that at most one child can be kept at the current LP module.

There are two default options—in both of them, objective values of the pre-solved LP relaxations are compared (for those children whose pre-solve did not terminate with primal infeasibility or high cost). One rule prefers the child with the lowest objective function value and the other prefers the child with the higher objective function value.

Arguments:

`void *user`	IN	Pointer to the user-defined LP data structure.
`int ub`	IN	The current best upper bound.
`branch_obj *can`	IN	The branching candidate.
`char *action`	OUT	Array of actions for the children. The array is already allocated to length `can->number`.

Return values:

`USER_ERROR`	Error. `DEFAULT` is used.
`USER_SUCCESS`	User filled out `*action`.
`USER_DEFAULT`	Regulated by the `select_child_default` parameter, which is initially set to `PREFER_LOWER_OBJ_VALUE`, unless overridden by the user.
`PREFER_HIGHER_OBJ_VALUE`	Choose child with the highest objective value.
`PREFER_LOWER_OBJ_VALUE`	Choose child with the lowest objective value.
`PREFER_MORE_FRACTIONAL`	Choose child with the most fractional variables. Fractional branching options are only available if the fractional branching compile-time option is set in the makefile.
`PREFER_LESS_FRACTIONAL`	Choose child with the lowest number of fractional variables.

Post-processing:

Checks which children can be fathomed based on the objective value of their pre-solved LP relaxation.

Wrapper invoked from: `branch()`.

user_print_branch_stat

```
int user_print_branch_stat(void *user, branch_obj *can, cut_data *cut,
                           int n, var_desc **vars, char *action)
```

Description:

Print out information about branching candidate `can`, such as a more explicit problem-specific description than SYMPHONY can provide (for instance, end points of an edge). If `verbosity` is set high enough, the identity of the branching object and the children (with objective values and termination codes for the pre-solved LPs) is printed out to the standard output by SYMPHONY.

Arguments:

`void *user`	IN	Pointer to the user-defined LP data structure.
`branch_obj *can`	IN	The branching candidate.
`cut_data *cut`	IN	The description of the cut if the branching object is a cut.
`int n`	IN	Number of variables.
`var_desc **vars`	IN	Array of variables in the current relaxation.
`char *action`	IN	Array of actions for the children.

Return values:

`USER_ERROR`	Error. Ignored by SYMPHONY.
`USER_SUCCESS`	The user printed out whatever she wanted to.
`USER_DEFAULT`	SYMPHONY prints out its own branching information.

Wrapper invoked from: `branch()` after the best candidate has been selected, pre-solved, and the action is decided on for the children.

user_add_to_desc

```
int user_add_to_desc(void *user, int *desc_size, char **desc)
```

Description:

Before a node description is sent to the TM, the user can provide a pointer to a data structure that will be appended to the description for later use by the user in reconstruction of the node. This information must be placed into `*desc`. Its size should be returned in `*desc_size`.

There is only one default option: the description to be added is considered to be of zero length, i.e., there is no additional description.

Arguments:

`void *user`	IN	Pointer to the user-defined LP data structure.
`int *desc_size`	OUT	The size of the additional information, the length of `*desc` in bytes.
`char **desc`	OUT	Pointer to the additional information (space must be allocated by the user).

Return values:

`USER_ERROR`	Error. `DEFAULT` is used.
`USER_SUCCESS`	User filled out `*desc_size` and `*desc`.
`USER_DEFAULT`	No description is appended.

Wrapper invoked from: `create_explicit_node_desc()` before a node is sent to the tree manager.

user_same_cuts

```
int user_same_cuts (void *user, cut_data *cut1, cut_data *cut2,
                    int *same_cuts)
```

Description:

Determine whether the two cuts are comparable (the normals of the half-spaces corresponding to the cuts point in the same direction) and if yes, which one is stronger. The default is to declare the cuts comparable only if the `type`, `sense` and `coef` fields of the two cuts are the same byte by byte; and if this is the case to compare the right hand sides to decide which cut is stronger.

Arguments:

`void *user`	IN	Pointer to the user-defined LP data structure.
`cut_data *cut1`	IN	The first cut.
`cut_data *cut2`	IN	The second cut.
`int *same_cuts`	OUT	Possible values: `SAME`, `FIRST_CUT_BETTER`, `SECOND_CUT_BETTER` and `DIFFERENT` (i.e., not comparable).

Return values:

`USER_ERROR`	Error. `DEFAULT` is used.
`USER_SUCCESS`	User did the comparison, filled out `*same_cuts`.
`USER_DEFAULT`	Compare byte by byte (see above).

Wrapper invoked from: `process_message()` when a `PACKED_CUT` arrives.

Note:

This function is used to check whether a newly arrived cut is already in the local pool. If so, or if it is weaker than a cut in the local pool, then the new cut is discarded; if it is stronger then a cut in the local pool, then the new cut replaces the old one and if the new is different from all the old ones, then it is added to the local pool.

user_unpack_cuts

```
int user_unpack_cuts(void *user, int from, int type, int varnum,
                     var_desc **vars, int cutnum, cut_data **cuts,
                     int *new_row_num, waiting_row ***new_rows)
```

Description:

If the user has defined application-specific cut classes, these cuts must be interpreted as constraints for the current LP relaxation, i.e., the user must decode the compact representation of the cuts (see the `cut_data` structure) into rows for the matrix. A pointer to the array of generated rows must be returned in `***new_rows` (the user has to allocate this array) and their number in `*new_row_num`.

Note that SYMPHONY has built-in support for cuts generated explicitly as matrix rows with no user-defined packed form, i.e., cuts whose indices and coefficients are given explicitly (see the function `user_find_cuts()` in Section 7.3.3. These cuts can be constructed and added using the helper function `cg_add_explicit_cut()` (see the description of `user_find_cuts()` in Section 7.3.3) and are packed and unpacked automatically, so the user does not need to implement this function. In post processing, SYMPHONY unpacks explicitly defined cuts and internally generated generic cuts.

Arguments:

`void *user`	IN	Pointer to the user-defined LP data structure.
`int from`	IN	See below in "Notes".
`int type`	IN	`UNPACK_CUTS_SINGLE` or `UNPACK_CUTS_MULTIPLE` (see notes below).
`int varnum`	IN	The number of variables.
`var_desc **vars`	IN	The variables currently in the problem.
`int cutnum`	IN	The number of cuts to be decoded.
`cut_data **cuts`	IN	Cuts that need to be converted to rows for the current LP. See "Warning" below.
`int *new_row_num`	OUT	Pointer to the number of rows in `**new_rows`.
`waiting_row ***new_rows`	OUT	Pointer to the array of pointers to the new rows.

Return values:

`USER_ERROR`	Error. The cuts are discarded.
`USER_SUCCESS`	User unpacked the cuts.
`USER_DEFAULT`	There are no user cut types defined. In this case, SYMPHONY deals with only explicit cuts and internally generated cuts.

Wrapper invoked from: Wherever a cut needs to be unpacked (multiple places).

Post-processing:

Explicit row cuts are processed, as well as SYMPHONY's internally generated cuts. Also, the pointers to each cut are transferred to the `waiting_rows` data structure (in previous version, this was done by the user).

Notes:

- When decoding the cuts, the expanded constraints have to be adjusted to the current LP, i.e., coefficients corresponding to variables currently not in the LP have to be left out.
- If the `one_row_only` flag is set to `UNPACK_CUTS_MULTIPLE`, then the user can generate as many constraints (even zero!) from a cut as she wants (this way she can lift the cuts, thus adjusting them for the current LP). However, if the flag is set to `UNPACK_CUTS_SINGLE`, then for each cut the user must generate a unique row, the same one that had been generated from the cut before. (The flag is set to this value only when regenerating a search tree node.)
- The `from` argument can take on six different values: `CUT_FROM_CG`, `CUT_FROM_CP`, `CUT_FROM_TM`, `CUT_LEFTOVER` (these are cuts from a previous LP relaxation that are still in the local pool), `CUT_NOT_IN_MATRIX_SLACK` and `CUT_VIOLATED_SLACK` indicating where the cut came from. This might be useful in deciding whether to lift the cut or not.
- The `matind` fields of the rows must be filled with indices with respect to the position of the variables in `**vars`.
- **Warning:** For each row, the user must make sure that the cut the row was generated from (and can be uniquely regenerated from if needed later) is safely stored in the `waiting_row` structure. SYMPHONY will free the entries in `cuts` after this function returns. If a row is generated from a cut in `cuts` (and not from a lifted cut), the user has the option of physically copying the cut into the corresponding part of the `waiting_row` structure, or copying the pointer to the cut into the `waiting_row` structure and erasing the pointer in `cuts`. If a row is generated from a lifted cut, the user should store a copy of the lifted cut in the corresponding part of `waiting_row`.

user_send_lp_solution

```
int user_send_lp_solution(void *user, int varnum, var_desc **vars,
                          double *x, int where)
```

Description:

This function is only used in the case of parallel execution. The user has the option to send the LP solution to either the cut pool or the cut generator in some user-defined form if desired. There are two default options—sending the indices and values for all nonzero variables (`SEND_NONZEROS`) and sending the indices and values for all fractional variables (`SEND_FRACTIONS`).

Arguments:

`void *user`	IN	Pointer to the user-defined LP data structure.
`int varnum`	IN	The number of variables currently in the LP relaxation. (The length of the `*vars` and `x` arrays.)
`var_desc **vars`	IN	The variables currently in the LP relaxation.
`double *x`	IN	Values of the above variables.
`int where`	IN	Where the solution is to be sent—`LP_SOL_TO_CG` or `LP_SOL_TO_CP`.

Return values:

`USER_ERROR`	Error. No message will be sent.
`USER_SUCCESS`	User packed and sent the message.
`USER_DEFAULT`	Regulated by the `pack_lp_solution_default` parameter, initially set to `SEND_NOZEROS`.
`SEND_NONZEROS`	Send user indices and values of variables at nonzero level.
`SEND_FRACTIONS`	Send user indices and values of variables at fractional level.

Wrapper invoked from: `fathom_branch()` after an LP relaxation has been solved. The message is always sent to the cut generator (if there is one). The message is sent to the cut pool if a search tree node at the top of a chain is being processed (except at the root in the first phase), or if a given number (`cut_pool_check_freq`) of LP relaxations have been solved since the last check.

Note:

The wrapper automatically packs the level, index, and iteration number corresponding to the current LP solution within the current search tree node, as well as the objective value and upper bound in case the solution is sent to a cut generator. This data will be unpacked by SYMPHONY on the receiving end, the user will have to unpack there exactly what he has packed here.

user_logical_fixing

```
int user_logical_fixing(void *user, int varnum, var_desc **vars,
                        double *x, char *status, int *num_fixed)
```

Description:

Logical fixing is modifying the stati of variables based on logical implications derived from problem-specific information. In this function the user can modify the status of any variable. Valid stati are: `NOT_FIXED`, `TEMP_FIXED_TO_LB`, `PERM_FIXED_TO_LB`, `TEMP_FIXED_TO_UB` and `PERM_FIXED_TO_UB`. Be forewarned that fallaciously fixing a variable in this function can cause the algorithm to terminate improperly. Generally, a variable can only be fixed permanently if the matrix is ***full*** at the time of the fixing (i.e. all variables that are not fixed are in the matrix). There are no default options.

Arguments:

`void *user`	IN	Pointer to the user-defined LP data structure.
`int varnum`	IN	The number of variables currently in the LP relaxation. (The length of the `*vars` and `x` arrays.)
`var_desc **vars`	IN	The variables currently in the LP relaxation.
`double *x`	IN	Values of the above variables.
`char *status`	INOUT	Stati of variables currently in the LP relaxation.
`int *num_fixed`	OUT	Number of fixed variables.

Return values:

`USER_ERROR`	Error. Ignored by SYMPHONY.
`USER_SUCCESS`	User changed the stati of the variables she wanted.
`USER_DEFAULT`	No logical fixing rules are implemented.

Wrapper invoked from: `fix_variables()` after doing reduced cost fixing, but only when a specified number of variables have been fixed by reduced cost (see LP parameter settings).

user_generate_column

```
int user_generate_column(void *user, int generate_what, int cutnum,
                         cut_data **cuts, int prevind, int nextind,
                         int *real_nextind, double *colval,
                         int *colind, int *collen, double *obj,
                         double *lb, double *ub)
```

Description:

This function is called when pricing out the columns that are not already fixed and are not explicitly represented in the matrix. Only the user knows the explicit description of these columns. When a missing variable need to be priced, the user is asked to provide the corresponding column. SYMPHONY scans through the known variables in the order of their user indices. After testing a variable in the matrix (`prevind`), SYMPHONY asks the user if there are any missing variables to be priced before the next variable in the matrix (`nextind`). If there are missing variables before `nextind`, the user has to supply the user index of the real next variable (`real_nextind`) along with the corresponding column. Occasionally SYMPHONY asks the user to simply supply the column corresponding to `nextind`. The `generate_what` flag is used for making a distinction between the two cases: in the former case it is set to `GENERATE_REAL_NEXTIND` and in the latter it is set to `GENERATE_NEXTIND`.

Arguments:

void *user	IN	Pointer to the user-defined LP data structure.
int generate_what	IN	GENERATE_NEXTIND or GENERATE_REAL_NEXTIND (see description above).
int cutnum	IN	The number of added rows in the LP formulation (i.e., the total number of rows less the number of base constraints). This is the length of the **cuts array.
cut_data **cuts	IN	Description of the cuts corresponding to the added rows of the current LP formulation. The user is supposed to know about the cuts corresponding to the base constraints.
int prevind	IN	The last variable processed (−1 if there was none) by SYMPHONY.
int nextind	IN	The next variable (−1 if there are none) known to SYMPHONY.
int *real_nextind	OUT	Pointer to the user index of the next variable (−1 if there is none).
double *colval	OUT	Values of the nonzero entries in the column of the next variable. (Sufficient space is already allocated for this array.)
int *colind	OUT	Row indices of the nonzero entries in the column. (Sufficient space is already allocated for this array.)
int *collen	OUT	The length of the colval and colind arrays.
double *obj	OUT	Objective coefficient corresponding to the next variable.
double *lb	OUT	Lower bound of the next variable.
double *ub	OUT	Upper bound of the next variable.

Return values:

USER_ERROR	Error. The LP process is aborted.
USER_SUCCESS	User filled out *real_nextind and generated its column if needed.
USER_DEFAULT	No column generation is done.

Wrapper invoked from: price_all_vars() and restore_lp_feasibility().

Note:

colval, colind, collen and obj do not need to be filled out if real_nextind is the same as nextind and generate_what is GENERATE_REAL_NEXTIND.

user_generate_cuts_in_lp

```
int user_generate_cuts_in_lp(void *user, LPdata *lp_data, int varnum,
                             var_desc **vars, double *x, int *new_row_num,
                             cut_data ***cuts)
```

Description:

The user might decide to generate cuts directly within the LP module instead of using the cut generator. This can be accomplished either through a call to this function or simply by configuring SYMPHONY such that the cut generator is called directly from the LP solver. One example of when this might be done is when generating Gomory cuts or something else that requires knowledge of the current LP tableau. The user must return the list of generated cuts by allocating an array of `cut_data` structures and setting `*cuts` to point to this array. Post-processing consists of checking if any of the new cuts are already in the local pool (or dominated by a cut in the local pool).

Arguments:

`void *user`	IN
`LPdata *lp_data`	A pointer to SYMPHONY's internal data structure for storing the LP relaxatio
`int varnum`	IN
`var_desc **vars`	IN
`double *x`	IN
`int *new_row_num`	OUT
`cut_data ***cuts`	OUT

Return values:

`USER_ERROR`	Error. Interpreted as if no cuts were generated.
`USER_SUCCESS`	Cuts were generated.
`USER_DEFAULT`	No cuts were generated. By default, SYMPHONY uses the CGL to generate generic cuts, according to parameter settings.
`GENERATE_CGL_CUTS`	Generate generic CGL cuts, according to parameter settings.
`DO_NOT_GENERATE_CGL_CUTS`	No additional cuts are generated.

Post-processing:

SYMPHONY checks if any of the newly generated cuts are already in the local pool.

Wrapper invoked from: `receive_cuts()` before the cuts from the CG module are received. Since the user will probably use this function to generate tableau-dependent cuts, it is highly unlikely that any of the new cuts would already be in the pool. Therefore the user will probably return `USER_AND_PP` to force SYMPHONY to skip post-processing.

Notes:

- Just like in `user_unpack_cuts()`, the user has to allocate space for the rows.
- Unless the `name` field of a cut is explicitly set to `CUT__SEND_TO_CP`, SYMPHONY will assume that the cut is locally valid only and set that field to `CUT__DO_NOT_SEND_TO_CP`.

user_print_stat_on_cuts_added

```
int user_print_stat_on_cuts_added(void *user, int rownum, waiting_row **rows)
```

Description:

The user can print out some information (if he wishes to) on the cuts that will be added to the LP formulation. The default is to print out the number of cuts added.

Arguments:

`void *user`	IN	Pointer to the user-defined LP data structure.
`int rownum`	IN	The number of cuts added.
`waiting_row **rows`	IN	Array of waiting rows containing the cuts added.

Return values:

`USER_ERROR`	Revert to default.
`USER_SUCCESS`	User printed whatever he wanted.
`USER_DEFAULT`	Print out the number of cuts added.

Wrapper invoked from: `add_best_waiting_rows()` after it has been decided how many cuts to add and after the cuts have been selected from the local pool.

user_purge_waiting_rows

```
int user_purge_waiting_rows(void *user, int rownum,
                            waiting_row **rows, char *delete_rows)
```

Description:

The local pool is purged from time to time to control its size. In this function the user has the power to decide which cuts to purge from this pool if desired. To mark the $\texttt{i}^{\texttt{th}}$ waiting row (an element of the pre-pool) for removal she has to set `delete_rows[i]` to be `TRUE` (`delete_rows` is allocated before the function is called and its elements are set to `FALSE` by default).

Post-processing consists of actually deleting those entries from the waiting row list and compressing the list. The default is to discard the least violated waiting rows and keep no more than what can be added in the next iteration (this is determined by the `max_cut_num_per_iter` parameter).

Arguments:

`void *user`	IN	Pointer to the user-defined LP data structure.
`int rownum`	IN	The number of waiting rows.
`waiting_row **rows`	IN	The array of waiting rows.
`char *delete_rows`	OUT	An array of indicators showing which waiting rows are to be deleted.

Return values:

`USER_ERROR`	Purge every single waiting row.
`USER_SUCCESS`	The user marked in `delete` the rows to be deleted.
`USER_DEFAULT`	Described above.

Post-processing:

The marked rows are deleted.

Wrapper invoked from: `receive_cuts()` after cuts have been added.

user_free_lp

```
int user_free_lp(void **user)
```

Description:

The user has to free all the data structures within `*user`, and also free `user` itself. The user can use the built-in macro `FREE` that checks the existence of a pointer before freeing it.

Arguments:

`void **user`	INOUT	Pointer to the user-defined LP data structure.

Return values:

`USER_ERROR`	Error. SYMPHONY ignores error message.
`USER_SUCCESS`	User freed everything in the user space.
`USER_DEFAULT`	There is no user memory to free.

Wrapper invoked from: `lp_close()` at module shutdown.

7.3.3 Cut generator module callbacks

Due to the relative simplicity of the cut generator, there are no wrapper functions implemented for CG. Consequently, there are no default options and no post-processing.

user_receive_cg_data

```
int user_receive_cg_data (void **user)
```

Description:

This function only has to be filled out for parallel execution and only if the TM, LP, and CG modules are all compiled as separate modules. This would not be typical. If needed, the user can use this function to receive problem-specific data needed for computation in the CG module. The same data must be received here that was sent in the `user_send_cg_data()` (see Section 7.3.1) function in the master module. The user has to allocate space for all the data structures, including `user` itself. Note that some or all of this may be done in the function `user_send_cg_data()` if the Tree Manager, LP, and CG are all compiled together. See that function for more information.

Arguments:

`void **user`	INOUT	Pointer to the user-defined data structure.

Return values:

`USER_ERROR`	Error. CG exits.
`USER_SUCCESS`	The user received the data properly.
`USER_DEFAULT`	User did not send any data.

Invoked from: `cg_initialize()` at process start.

user_receive_lp_solution_cg

```
int user_receive_lp_solution_cg(void *user)
```

Description:

This function is invoked only in the case of parallel computation and only if in the `user_send_lp_solution()` function of the LP module the user opted for packing the current LP solution herself. Here she must receive the data sent from there.

Arguments:

`void *user`	IN	Pointer to the user-defined data structure.

Invoked from: Whenever an LP solution is received.

Return values:

`USER_ERROR`	Error. The LP solution was not received and will not be processed.
`USER_SUCCESS`	The user received the LP solution.
`USER_DEFAULT`	The solution was sent by SYMPHONY and will be received automatically.

Note:

SYMPHONY automatically unpacks the level, index and iteration number corresponding to the current LP solution within the current search tree node as well as the objective value and upper bound.

user_find_cuts

```
int user_find_cuts(void *user, int varnum, int iter_num, int level,
    int index, double objval, int *indices, double *values,
    double ub, double lpetol, int *cutnum)
```

Description:

In this function, the user can generate cuts based on the current LP solution stored in `soln`. Cuts found are added to the LP by calling the `cg_add_user_cut(cut_data *new_cut)` function, which then transfers the cut to the LP module, either through message passing or shared memory. The argument of this function is a pointer to the cut to be sent. See Section 7.3.2 for a description of this data structure. Each user-defined cut assigned a type and a designated packed form. Each user-defined type must be recognized by the user's `user_unpack_cuts()`7.3.2 function in the master module. If the user wants a user-defined cut to be added to the cut pool in case it proves to be effective in the LP, then `new_cut->name` should be set to `CUT__SEND_TO_CP`. In this case, the cut must be globally valid. Otherwise, it should be set to `CUT__DO_NOT_SEND_TO_CP`.

Alternatively, SYMPHONY provides automated support for the generation of cuts represented explicitly as matrix rows. These cuts are passed as sparse vectors and can be added by calling the routine `cg_add_explicit_cut()`, which has the following interface.

```
int cg_add_explicit_cut(int nzcnt, int *indices, double *values,
                        double rhs, double range, char sense,
                        char send_to_cp)
```

Here, `nzcnt` is the number of nonzero coefficients in the cut, `indices` is an array containing the indices of the columns with nonzero entries, and `values` is an array of the corresponding values. The right hand side value is passed in through the variable `rhs`, the range is passed in through the variable `range`, and the sense of the inequality is passed through the variable `sense`. Finally, the variable `send_to_cp` indicates to SYMPHONY whether the cut is globally valid and should be sent to the cut pool, or whether it is only to be used locally.

The only output of the `user_find_cuts()` function is the number of cuts generated and this value is returned in the last argument. For options to generate generic cuts automatically using the COIN Cut Generation Library, see the function `user_generate_cuts_in_lp()`7.3.2

Arguments:

`void *user`	IN	Pointer to the user-defined data structure.
`int iter_num`	IN	The iteration number of the current LP solution.
`int level`	IN	The level in the tree on which the current LP solution was generated.
`index`	IN	The index of the node in which LP solution was generated.
`objval`	IN	The objective function value of the current LP solution.
`int varnum`	IN	The number of nonzeros in the current LP solution.
`indices`	IN	The column indices of the nonzero variables in the current LP solution.
`values`	IN	The values of the nonzero variables listed in `indices`.
`double ub`	IN	The current global upper bound.
`double lpetol`	IN	The current error tolerance in the LP.
`int *cutnum`	OUT	Pointer to the number of cuts generated and sent to the LP.

Return values:

`USER_ERROR`	Ignored.
`USER_SUCCESS`	The user function exited properly.
`USER_DEFAULT`	No cuts were generated.

Invoked from: Whenever an LP solution is received.

user_check_validity_of_cut

```
int user_check_validity_of_cut(void *user, cut_data *new_cut)
```

Description:

This function is provided as a debugging tool. Every cut that is to be sent to the LP solver is first passed to this function where the user can independently verify that the cut is valid by testing it against a known feasible solution (usually an optimal one). This is useful for determining why a particular known feasible (optimal) solution was never found. Usually, this is due to an invalid cut being added. See Section 6.8.4 for more on this feature.

Arguments:

`void *user`	IN	Pointer to the user-defined data structure.
`cut_data *new_cut`	IN	Pointer to the cut that must be checked.

Return values:

`USER_ERROR`	Ignored.
`USER_SUCCESS`	The user is done checking the cut.
`USER_DEFAULT`	The cut is ignored.

Invoked from: Whenever a cut is being sent to the LP.

user_free_cg

```
int user_free_cg(void **user)
```

Description:

The user has to free all the data structures within `user`, and also free `user` itself. The user can use the built-in macro `FREE` that checks the existence of a pointer before freeing it.

Arguments:

`void **user`	INOUT	Pointer to the user-defined data structure (should be `NULL` on exit from this function).

Return values:

`USER_ERROR`	Ignored.
`USER_SUCCESS`	The user freed all data structures.
`USER_DEFAULT`	The user has no memory to free.

Invoked from: `cg_close()` at module shutdown.

7.3.4 Cut pool module callbacks

Due to the relative simplicity of the cut pool, there are no wrapper functions implemented for CP. Consequently, there are no default options and no post-processing.

user_receive_cp_data

```
int user_receive_cp_data(void **user)
```

Description:

The user has to receive here all problem-specific information sent from `user_send_cp_data()` (see Section 7.3.1) function in the master module. The user has to allocate space for all the data structures, including `user` itself. Note that this function is only called if the either the Tree Manager, LP, or CP are running as a separate process (i.e. either `COMPILE_IN_TM`, `COMPILE_IN_LP`, or `COMPILE_IN_CP` are set to `FALSE` in the `make` file). Otherwise, this is done in `user_send_cp_data()`. See the description of that function for more details.

Arguments:

`void **user`	INOUT	Pointer to the user-defined data structure.

Return values:

`USER_ERROR`	Error. Cut pool module exits.
`USER_SUCCESS`	The user received data successfully.
`USER_DEFAULT`	The user did not send any data to be received.

Invoked from: `cp_initialize` at module start-up.

user_receive_lp_solution_cp

```
void user_receive_lp_solution_cp(void *user)
```

Description:

This function is invoked only in the case parallel computation and only if in the `user_send_lp_solution()` function of the LP module, the user opted for packing the current LP solution in a custom format. Here she must receive the data she sent there.

Arguments:

`void *user`	IN	Pointer to the user-defined data structure.

Return values:

`USER_ERROR`	Cuts are not checked for this LP solution.
`USER_SUCCESS`	The user function executed properly.
`USER_DEFAULT`	SYMPHONY's default format should be used.

Invoked from: Whenever an LP solution is received.

Note:

SYMPHONY automatically unpacks the level, index and iteration number corresponding to the current LP solution within the current search tree node.

user_prepare_to_check_cuts

```
int user_prepare_to_check_cuts(void *user, int varnum, int *indices,
                               double *values)
```

Description:

This function is invoked after an LP solution is received but before any cuts are tested. Here the user can build up data structures (e.g., a graph representation of the solution) that can make the testing of cuts easier in the `user_check_cuts` function.

Arguments:

`void *user`	IN	Pointer to the user-defined data structure.
`int varnum`	IN	The number of nonzero/fractional variables described in `indices` and `values`.
`int *indices`	IN	The user indices of the nonzero/fractional variables.
`double *values`	IN	The nonzero/fractional values.

Return values:

`USER_ERROR`	Cuts are not checked for this LP solution.
`USER_SUCCESS`	The user is prepared to check cuts.
`USER_DEFAULT`	There are no user-defined cuts in the pool.

Invoked from: Whenever an LP solution is received.

user_check_cut

```
int user_check_cut(void *user, double lpetol, int varnum,
                   int *indices, double *values, cut_data *cut,
                   int *is_violated, double *quality)
```

Description:

The user has to determine whether a given cut is violated by the given LP solution (see Section 7.3.2 for a description of the `cut_data data` data structure). Also, the user can assign a number to the cut called the *quality*. This number is used in deciding which cuts to check and purge. See the section on Cut Pool Parameters for more information.

Arguments:

`void *user`	INOUT	The user defined part of p.
`double lpetol`	IN	The ϵ tolerance in the LP module.
`int varnum`	IN	Same as the previous function.
`int *indices`	IN	Same as the previous function.
`double *values`	IN	Same as the previous function.
`cut_data *cut`	IN	Pointer to the cut to be tested.
`int *is_violated`	OUT	TRUE/FALSE based on whether the cut is violated or not.
`double *quality`	OUT	a number representing the relative strength of the cut.

Return values:

`USER_ERROR`	Cut is not sent to the LP, regardless of the value of `*is_violated`.
`USER_SUCCESS`	The user function exited properly.
`USER_DEFAULT`	Same as error.

Invoked from: Whenever a cut needs to be checked.

Note:

The same note applies to `number`, `indices` and `values` as in the previous function.

user_finished_checking_cuts

```
int user_finished_checking_cuts(void *user)
```

Description:

When this function is invoked there are no more cuts to be checked, so the user can dismantle data structures he created in `user_prepare_to_check_cuts`. Also, if he received and stored the LP solution himself he can delete it now.

Arguments:

`void *user`	IN	Pointer to the user-defined data structure.

Return values:

`USER_ERROR`	Ignored.
`USER_SUCCESS`	The user function exited properly.
`USER_DEFAULT`	There are no user-defined cuts in the pool.

Invoked from: After all cuts have been checked.

user_free_cp

```
int user_free_cp(void **user)
```

Description:

The user has to free all the data structures within `user`, and also free `user` itself. The user can use the built-in macro `FREE` that checks the existence of a pointer before freeing it.

Arguments:

`void **user`	INOUT	Pointer to the user-defined data structure (should be `NULL` on exit).

Return values:

`USER_ERROR`	Ignored.
`USER_SUCCESS`	The user freed all data structures.
`USER_DEFAULT`	There is nothing to be freed.

Invoked from: cp_close() at module shutdown.

7.3.5 Draw graph module callbacks

Due to the relative simplicity of the cut pool, there are no wrapper functions implemented for DG. Consequently, there are no default options and no post-processing.

user_dg_process_message

```
void user_dg_process_message(void *user, window *win, FILE *write_to)
```

Description:

The user has to process whatever user-defined messages are sent to the process. A write-to pipe to the wish process is provided so that the user can directly issue commands there.

Arguments:

`void *user`	INOUT	Pointer to the user-defined data structure.
`window *win`	INOUT	The window that received the message.
`FILE *write_to`	IN	Pipe to the wish process.

Return values:

`USER_ERROR`	Error. Message ignored.
`USER_SUCCESS`	The user processed the message.

user_dg_init_window

```
void user_dg_init_window(void **user, window *win)
```

Description:

The user must perform whatever initialization is necessary for processing later commands. This usually includes setting up the user's data structure for receiving and storing display data.

Arguments:

`void **user`	INOUT	Pointer to the user-defined data structure.
`window *win`	INOUT	

Return values:

`USER_ERROR`	Error. Ignored.
`USER_SUCCESS`	The user successfully performed initialization.

user_dg_free_window

```
void user_dg_free_window(void **user, window *win)
```

Description:

The user must free any data structures allocated.

Arguments:

`void **user`	INOUT	Pointer to the user-defined data structure.
`window *win`	INOUT	

Return values:

`USER_ERROR`	Error. Ignored.
`USER_SUCCESS`	The user successfully freed the data structures.

user_interpret_text

```
void user_interpret_text(void *user, int text_length,
 char *text, int owner_tid)
```

Description:

The user can interpret text input from the window.

Arguments:

`void *user`	INOUT	Pointer to the user-defined data structure.
`int text_length`	IN	The length of `text`.
`char *text`	IN	
`int owner_tid`	IN	The tid of the process that initiated this window.

Return values:

`USER_ERROR`	Error. Ignored.
`USER_SUCCESS`	The user successfully interpreted the text.

7.4 Run-time Parameters

Parameters can be set in one of two ways. Some commonly-used parameters can be set on the command line. To see a list of these, run SYMPHONY with no command-line arguments. Other parameters must be set in a parameter file. The name of this file is specified on the command line with "`-f`". Each line of the parameter file contains either a comment or two words – a keyword and a value, separated by white space. If the first word (sequence of non-white-space characters) on a line is not a keyword, then the line is considered a comment line. Otherwise the parameter corresponding to the keyword is set to the listed value. Usually the keyword is the same as the parameter name in the source code. Here we list the keywords, the type of value that should be given with the keywords and the default value. A parameter corresponding to keyword "K" in module "P" can also be set by using the keyword "P_K".

To make this list shorter, occasionally a comma separated list of parameters is given if the meanings of those parameters are strongly connected. For clarity, the constant name is sometimes given instead of the numerical value for default settings and options. The corresponding value is given in curly braces for convenience.

7.4.1 Global parameters

`verbosity` – **integer (0).** Sets the verbosity of all modules to the given value. In general, the greater this number the more verbose each module is. Experiment to find out what this means.

`random_seed` – **integer (17).** A random seed.

`granularity` – **double (1e-6).** should be set to "the minimum difference between two distinct objective function values" less the epsilon tolerance. E.g., if every variable is integral and the objective coefficients are integral then for any feasible solution the objective value is integer, so `granularity` could be correctly set to .99999.

`upper_bound` – **double (none)** . The value of the best known upper bound.

`probname` – **string (empty string)** . The name of the problem name.

`infile_name` – **string (empty string)** . The name of the input file which was read by "-F" flag.

7.4.2 Master module parameters

`M_verbosity` – **integer (0).**

`M_random_seed` – **integer (17).** A random seed just for the Master module.

`upper_bound` – **double (no upper bound).** This parameter is used if the user wants to artificially impose an upper bound (for instance if a solution of that value is already known).

`lower_bound` – **double (no lower bound).** This parameter is used if the user wants to artificially impose a lower bound.

`upper_bound_estimate` – **double (no estimate).** This parameter is used if the user wants to provide an estimate of the optimal value which will help guide the search. This is used in conjunction with the diving strategy `BEST_ESTIMATE`.

tm_exe, dg_exe – **strings ("tm", "dg").** The name of the executable files of the TM and DG modules. Note that the TM executable name may have extensions that depend on the configuration of the modules, but the default is always set to the file name produced by the makefile. If you change the name of the treemanager executable from the default, you must set this parameter to the new name.

tm_debug, dg_debug – **boolean (both FALSE).** Whether these modules should be started under a debugger or not (see 6.8.2 for more details on this).

tm_machine – **string (empty string).** On which processor of the virtual machine the TM should be run. Leaving this parameter as an empty string means arbitrary selection.

do_draw_graph – **boolean (FALSE).** Whether to start up the DG module or not (see Section 6.8.5 for an introduction to this).

do_branch_and_cut – **boolean (TRUE).** Whether to run the branch and cut algorithm or not. (Set this to FALSE to run the user's heuristics only.)

mc_search_order – **integer (MC_FIFO).** Use the fifo (MC_FIFO) or lifo (MC_LIFO) searh order during the multi criteria solution procedure.

mc_warm_start – **boolean(FALSE).** Whether to solve the corresponding problem of each iteration from a warm start loaded from a base iteration (which is the first iteration where gamma = 1.0 and tau = 0.0) or from scratch. Currently, this option is supported if only the supported solutions are desired to be found.

trim_warm_tree – **boolean(FALSE).** Whether to trim the warm start tree before re-solving. This consists of locating nodes whose descendants are all likely to be pruned in the resolve and eliminating those descendants in favor of processing the parent node itself.

mc_compare_solution_tolerance – **double(0.001).** If the difference between the objective values of two solutions to be compared, during the bicriteria solution procedure, are less than this tolerance, then assume them to be equal.

mc_binary_search_tolerance – **double(0).** The tolerance to be used to differentiate the gamma values if binary search is used during the bicriteria solution procedure. A value greater than zero will cause the binary search to be activated.

7.4.3 Draw Graph parameters

source_path – **string (".").** The directory where the DG tcl/tk scripts reside.

echo_commands – **boolean (FALSE).** Whether to echo the tcl/tk commands on the screen or not.

canvas_width, canvas_height – **integers (1000, 700).** The default width and height of the drawing canvas in pixels.

`viewable_width`, `viewable_height` – **integers (600, 400).** The default viewable width and height of the drawing canvas in pixels.

`interactive_mode` – **integer (`TRUE`).** Whether it is allowable to change things interactively on the canvas or not.

`node_radius` – **integer (8).** The default radius of a displayed graph node.

`disp_nodelabels`, `disp_nodeweights`, `disp_edgeweights` – **integers (all `TRUE`).** Whether to display node labels, node weights, and edge weights or not.

`nodelabel_font`, `nodeweight_font`, `edgeweight_font` – **strings (all "-adobe-helvetica-...").** The default character font for displaying node labels, node weights and edge weights.

`node_dash`, `edge_dash` – **strings (both empty string).** The dash pattern of the circles drawn around dashed nodes and that of dashed edges.

7.4.4 Tree Manager parameters

`TM_verbosity` – **integer (0).** The verbosity of the TM module.

`lp_exe`, `cg_exe`, `cp_exe` – **strings ("lp", "cg", "cp").** The name of the LP, CG, and CP module binaries. Note: when running in parallel using PVM, these executables (or links to them) must reside in the `PVM_ROOT/bin/PVM_ARCH/` directory. Also, be sure to note that the executable names may have extensions that depend on the configuration of the modules, but the defaults will always be set to the name that the makefile produces.

`lp_debug`, `cg_debug`, `cp_debug` – **boolean (all `FALSE`).** Whether the modules should be started under a debugger or not.

`max_active_nodes` – **integer (1).** The maximum number of active search tree nodes—equal to the number of LP and CG tandems to be started up.

`max_cp_num` – **integer (0).** The maximum number of cut pools to be used.

`lp_mach_num`, `cg_mach_num`, `cp_mach_num` – **integers (all 0).** The number of processors in the virtual machine to run LP (CG, CP) processes. If this value is 0 then the processes will be assigned to processors in round-robin order. Otherwise the next `xx_mach_num` lines describe the processors where the LP (CG, CP) modules must run. The keyword – value pairs on these lines must be **TM_xx_machine** and the name or IP address of a processor (the processor names need not be distinct). In this case the actual processes are assigned in a round robin fashion to the processors on this list.

This feature is useful if a specific software package is needed for some module, but that software is not licensed for every node of the virtual machine or if a certain process must run on a certain type of machine due to resource requirements.

`use_cg` – **boolean (`FALSE`).** Whether to use a cut generator or not.

`TM_random_seed` – **integer (17).** The random seed used in the TM.

`unconditional_dive_frac` – **double (0.1).** The fraction of the nodes on which SYMPHONY randomly dives unconditionally into one of the children.

`diving_strategy` – **integer (`BEST_ESTIMATE`{0}).** The strategy employed when deciding whether to dive or not.

The `BEST_ESTIMATE`*{0}* strategy continues to dive until the lower bound in the child to be dived into exceeds the parameter `upper_bound_estimate`, which is given by the user.

The `COMP_BEST_K`*{1}* strategy computes the average lower bound on the best `diving_k` search tree nodes and decides to dive if the lower bound of the child to be dived into does not exceed this average by more than the fraction `diving_threshold`.

The `COMP_BEST_K_GAP`*{2}* strategy takes the size of the gap into account when deciding whether to dive. After the average lower bound of the best `diving_k` nodes is computed, the gap between this average lower bound and the current upper bound is computed. Diving only occurs if the difference between the computed average lower bound and the lower bound of the child to be dived into is at most the fraction `diving_threshold` of the gap.

Note that fractional diving settings can override these strategies. See below.

`diving_k, diving_threshold` – **integer, double (1, 0.0).** See above.

`fractional_diving_ratio, fractional_diving_num` – **integer (0.02, 0).** Diving occurs automatically if the number of fractional variables in the child to be dived into is less than `fractional_diving_num` or the fraction of total variables that are fractional is less than `fractional_diving_ratio`. This overrides the other diving rules. Note that in order for this option to work, the code must be compiled with `FRACTIONAL_BRANCHING` defined. This is the default. See the makefile for more details.

`node_selection_rule` – **integer (`LOWEST_LP_FIRST`{0}).** The rule for selecting the next search tree node to be processed. This rule selects the one with lowest lower bound. Other possible values are: `HIGHEST_LP_FIRST`*{1}*, `BREADTH_FIRST_SEARCH`*{2}* and `DEPTH_FIRST_SEARCH`*{3}*.

`load_balance_level` – integer (-1).] A naive attempt at load balancing on problems where significant time is spent in the root node, contributing to a lack of parallel speed-up. Only a prescribed number of iterations (`load_balance_iter`) are performed in the root node (and in each subsequent node on a level less than or equal to `load_balance_level`) before branching is forced in order to provide additional subproblems for the idle processors to work on. This doesn't work well in general.

`load_balance_iter` – integer (-1).] Works in tandem with the `load_balance_level` to attempt some simple load balancing. See the above description.

`keep_description_of_pruned` – **integer (`DISCARD`{0}).** Whether to keep the description of pruned search tree nodes or not. The reasons to do this are (1) if the user wants to write out a proof of optimality using the logging function, (2) for debugging, or (3) to get a visual picture of the tree using the software VBCTOOL. Otherwise, keeping the pruned nodes around just takes up memory.

There are three options if it is desired to keep some description of the pruned nodes around. First, their full description can be written out to disk and freed from memory (`KEEP_ON_DISK_FULL`*{1}*). There is not really too much you can do with this kind of file, but theoretically, it contains a full record of the solution process and could be used to provide a certificate of optimality (if we were using exact arithmetic) using an independent verifier. In this case, the line following `keep_description_of_pruned` should be a line containing the keyword `pruned_node_file_name` with its corresponding value being the name of a file to which a description of the pruned nodes can be written. The file does not need to exist and will be over-written if it does exist.

If you have the software VBCTOOL (see Section 6.9), then you can alternatively just write out the information VBCTOOL needs to display the tree (`KEEP_ON_DISK_VBC_TOOL`*{*2*}*).

Finally, the user can set the value to of this parameter to `KEEP_IN_MEMORY`*{*2*}*, in which case all pruned nodes will be kept in memory and written out to the regular log file if that option is chosen. This is really only useful for debugging. Otherwise, pruned nodes should be flushed.

`keep_warm_start` – **boolean (**`FALSE`**).** Turning this parameter on will have exactly the same impact with setting the `keep_description_of_pruned` to `KEEP_IN_MEMORY`*{*2*}*. This will allow SYMPHONY to keep all the necessary information obtained from the branching tree of the original problem to be able to warm start after a parameter or problem data modification. Thus, if it is intended to warm start later, the user should set this parameter before solving the original problem.

`logging` – **integer (**`NO_LOGGING`**{0}).** Whether or not to write out the state of the search tree and all other necessary data to disk periodically in order to allow a warm start in the case of a system crash or to allow periodic viewing with VBCTOOL.

If the value of this parameter is set to `FULL_LOGGING`*{*1*}*, then all information needed to warm start the calculation will written out periodically. The next two lines of the parameter file following should contain the keywords `tree_log_file_name` and `cut_log_file_name` along with corresponding file names as values. These will be the files used to record the search tree and related data and the list of cuts needed to reconstruct the tree.

If the value of the parameter is set to `VBC_TOOL`*{*2*}*, then only the information VBCTOOL needs to display the tree will be logged. This is not really a very useful option since a "live" picture of the tree can be obtained using the `vbc_emulation` parameter described below (see Section 6.9 for more on this).

`logging_interval` – **integer (1800).** Interval (in seconds) between writing out the above log files.

`warm_start` – **boolean (0).** Used to allow the tree manager to make a warm start by reading in previously written log files. If this option is set, then the two line following must start with the keywords `warm_start_tree_file_name` and `warm_start_cut_file_name` and include the appropriate file names as the corresponding values.

`vbc_emulation` – integer (`NO_VBC_EMULATION`*{*0*}*).] Determines whether or not to employ the VBCTOOL emulation mode. If one of these modes is chosen, then the tree will be displayed in "real time" using the VBCTOOL Software. When using the option `VBC_EMULATION_LIVE`*{*2*}* and piping the output directly to VBCTOOL, the tree will be displayed as it is constructed, with color coding indicating the status of each node. With `VBC_EMULATION_FILE`*{*1*}* selected, a log file will be produced which can later be read into VBCTOOL to produce an emulation of the solution process at any desired speed. If VBC_EMULATION_FILE is selected, the the following line should contain the keyword vbc_emulation_file_name along with the corresponding file name for a value.

price_in_root – **boolean (FALSE).** Whether to price out variables in the root node before the second phase starts (called *repricing the root*).

trim_search_tree – **boolean (FALSE).** Whether to trim the search tree before the second phase starts or not. Useful only if there are two phases. (It is very useful then.)

colgen_in_first_phase, colgen_in_second_phase – **integers (both 4).** These parameters determine if and when to do column generation in the first and second phase of the algorithm. The value of each parameter is obtained by setting the last four bits. The last two bits refer to what to do when attempting to prune a node. If neither of the last two bits are set, then we don't do anything—we just prune it. If only the last bit is set, then we simply save the node for the second phase without doing any column generation (yet). If only the second to last bit is set, then we do column generation immediately and resolve if any new columns are found. The next two higher bits determine whether or not to do column generation before branching. If only the third lowest bit is set, then no column generation occurs before branching. If only the fourth lowest bit is set, then column generation is attempted before branching. The default is not to generate columns before branching or fathoming, which corresponds to only the third lowest bit being set, resulting in a default value of 4.

time_limit – **double (-1.0).** Number of seconds of wall-clock time allowed for solution. When this time limit is reached, the solution process will stop and the best solution found to that point, along with other relevant data, will be output. A time limit less than 0.0 means there is no limit.

node_limit – **integer (-1).** Number of nodes allowed to be analyzed during the solution. When this node limit is reached, the solution process will stop and the best solution found to that point, along with other relevant data, will be output. A node limit less than 0 means there is no limit.

gap_limit – **double (-1.0).** Target gap limit allowed for solution. When the gap between the lower and the upper bound reaches this point, the solution process will stop and the best solution found to that point, along with other relevant data, will be output. A gap limit less than 0 means there is no limit.

find_first_feasible – **boolean (FALSE).** Whether to stop after finding the first feasible solution or not.

sensitivity_analysis – **boolean (FALSE).** If the user wants to do the rudimentary sensitivity analysis, which will give a lower bound for the problem modified by the right hand side, then, this parameter has to be set before solving the original problem. If it is set, SYMPHONY will keep the necessary information from the solution processes of the original problem to be able to do the sensitivity analysis later.

7.4.5 LP parameters

LP_verbosity – **integer (0).** Verbosity level of the LP module.

`set_obj_upper_lim` – **boolean (FALSE).** Whether to stop solving the LP relaxation when it's optimal value is provably higher than the global upper bound. There are some advantages to continuing the solution process anyway. For instance, this results in the highest possible lower bound. On the other hand, if the matrix is full, this node will be pruned anyway and the rest of the computation is pointless. This option should be set at FALSE for column generation since the LP dual values may not be reliable otherwise.

`try_to_recover_from_error` – **boolean (TRUE).** Indicates what should be done in case the LP solver is unable to solve a particular LP relaxation because of numerical problems. It is possible to recover from this situation but further results may be suspect. On the other hand, the entire solution process can be abandoned.

`problem_type` – **integer (ZERO_ONE_PROBLEM{0}).** The type of problem being solved. Other values are INTEGER_PROBLEM{1} or MIXED_INTEGER_PROBLEM{2}. (Caution: The mixed-integer option is not well tested.)

`cut_pool_check_frequency` – **integer (10).** The number of iterations between sending LP solutions to the cut pool to find violated cuts. It is not advisable to check the cut pool too frequently as the cut pool module can get bogged down and the LP solution generally do not change that drastically from one iteration to the next anyway.

`not_fixed_storage_size` – **integer (2048).** The *not fixed list* is a partial list of indices of variables not in the matrix that have not been fixed by reduced cost. Keeping this list allows SYMPHONY to avoid repricing variables (an expensive operation) that are not in the matrix because they have already been permanently fixed. When this array reaches its maximum size, no more variable indices can be stored. It is therefore advisable to keep the maximum size of this array as large as possible, given memory limitations.

`max_non_dual_feas_to_add_min`, `max_non_dual_feas_to_add_max`, `max_non_dual_feas_to_add_frac` – integer, integer, double (20, 200, .05). These three parameters determine the maximum number of non-dual-feasible columns that can be added in any one iteration after pricing. This maximum is set to the indicated fraction of the current number of active columns unless this numbers exceeds the given maximum or is less than the given minimum, in which case, it is set to the max or min, respectively.

`max_not_fixable_to_add_min`, `max_not_fixable_to_add_max`, `max_not_fixable_to_add_frac` – integer, integer, double (100, 500, .1). As above, these three parameters determine the maximum number of new columns to be added to the problem because they cannot be priced out. These variables are only added when trying to restore infeasibility and usually, this does not require many variables anyway.

`mat_col_compress_num`, `mat_col_compress_ratio` – **integer, double (50, .05).** Determines when the matrix should be physically compressed. This only happens when the number of columns is high enough to make it "worthwhile." The matrix is physically compressed when the number of deleted columns exceeds either an absolute number *and* a specified fraction of the current number of active columns.

`mat_row_compress_num`, `mat_row_compress_ratio` – **integer, double (20, .05).** Same as above except for rows.

tailoff_gap_backsteps, tailoff_gap_frac – integer, double (2, .99). Determines when tailoff is detected in the LP module. Tailoff is reported if the average ratio of the current gap to the previous iteration's gap over the last tailoff_gap_backsteps iterations wasn't at least tailoff_gap_frac.

tailoff_obj_backsteps, tailoff_obj_frac – integer, double (2, .99). Same as above, only the ratio is taken with respect to the change in objective function values instead of the change in the gap.

ineff_cnt_to_delete – integer (0). Determines after how many iterations of being deemed ineffective a constraint is removed from the current relaxation.

eff_cnt_before_cutpool – integer (3). Determines after how many iterations of being deemed effective each cut will be sent to the global pool.

ineffective_constraints – integer (BASIC_SLACKS_ARE_INEFFECTIVE{2}). Determines under what condition a constraint is deemed ineffective in the current relaxation. Other possible values are NO_CONSTRAINT_IS_INEFFECTIVE{0}, NONZERO_SLACKS_ARE_INEFFECTIVE{1}, and ZERO_DUAL_VALUES_ARE_INEFFECTIVE{3}.

base_constraints_always_effective – boolean (TRUE). Determines whether the base constraints can ever be removed from the relaxation. In some case, removing the base constraints from the problem can be disastrous depending on the assumptions made by the cut generator.

branch_on_cuts – boolean (FALSE). This informs the framework whether the user plans on branching on cuts or not. If so, there is additional bookkeeping to be done, such as maintaining a pool of slack cuts to be used for branching. Therefore, the user should not set this flag unless he actually plans on using this feature.

discard_slack_cuts – integer (DISCARD_SLACKS_BEFORE_NEW_ITERATION{0}). Determines when the pool of slack cuts is discarded. The other option is DISCARD_SLACKS_WHEN_STARTING_NEW_NODE{1}.

first_lp_first_cut_time_out, first_lp_all_cuts_time_out, later_lp_first_cut_time_out, later_lp_all_cuts_time_out – double (0, 0, 5, 1). The next group of parameters determines when the LP should give up waiting for cuts from the cut generator and start to solve the relaxation in its current form or possibly branch if necessary. There are two factors that contribute to determining this timeout. First is whether this is the first LP in the search node of whether it is a later LP. Second is whether any cuts have been added already in this iteration. The four timeout parameters correspond to the four possible combinations of these two variables.

no_cut_timeout – This keyword does not have an associated value. If this keyword appears on a line by itself or with a value, this tells the framework not to time out while waiting for cuts. This is useful for debugging since it enables runs with a single LP module to be duplicated.

all_cut_timeout – double (no default). This keyword tells the framework to set all of the above timeout parameters to the value indicated.

max_cut_num_per_iter – integer (20). The maximum number of cuts that can be added to the LP in an iteration. The remaining cuts stay in the local pool to be added in subsequent iterations, if they are strong enough.

do_reduced_cost_fixing – **boolean** (FALSE). Whether or not to attempt to fix variables by reduced cost. This option is highly recommended

gap_as_ub_frac, gap_as_last_gap_frac – **double (.1, .7).** Determines when reduced cost fixing should be attempted. It is only done when the gap is within the fraction gap_as_ub_frac of the upper bound or when the gap has decreased by the fraction gap_as_last_gap_frac since the last time variables were fixed.

do_logical_fixing – **boolean** (FALSE). Determines whether the user's logical fixing routine should be used.

fixed_to_ub_before_logical_fixing, fixed_to_ub_frac_before_logical_fixing – **integer, double (1, .01)**. Determines when logical fixing should be attempted. It will be called only when a certain absolute number *and* a certain number of variables have been fixed to their upper bounds by reduced cost. This is because it is typically only after fixing variables to their upper bound that other variables can be logically fixed.

max_presolve_iter – **integer (10).** Number of simplex iterations to be performed in the presolve for strong branching.

strong_branching_cand_num_max, strong_branching_cand_num_min, strong_branching_red_ratio – **integer (25, 5, 1)**. These three parameters together determine the number of strong branching candidates to be used by default. In the root node, strong_branching_cand_num_max candidates are used. On each succeeding level, this number is reduced by the number strong_branching_red_ratio multiplied by the square of the level. This continues until the number of candidates is reduced to strong_branching_cand_num_min and then that number of candidates is used in all lower levels of the tree.

is_feasible_default – **integer** (TEST_INTEGRALITY**{1}**). Determines the default test to be used to determine feasibility. This parameter is provided so that the user can change the default behavior without recompiling. The only other option is TEST_ZERO_ONE*{0}*.

send_feasible_solution_default – **integer** (SEND_NONZEROS**{0}**). Determines the form in which to send the feasible solution. This parameter is provided so that the user can change the default behavior without recompiling. This is currently the only option.

send_lp_solution_default – **integer** (SEND_NONZEROS**{0}**). Determines the default form in which to send the LP solution to the cut generator and cut pool. This parameter is provided so that the user can change the default behavior without recompiling. The other option is SEND_FRACTIONS*{1}*.

display_solution_default – **integer** (DISP_NOTHING**{0}**). Determines how to display the current LP solution if desired. See the description of user_display_solution() for other possible values. This parameter is provided so that the user can change the default behavior without recompiling.

shall_we_branch_default – **integer** (USER__BRANCH_IF_MUST**{2}**). Determines the default branching behavior. Other values are USER__DO_NOT_BRANCH*{0}* (not recommended as a default), USER__DO_BRANCH*{1}* (also not recommended as a default), and USER__BRANCH_IF_TAILOFF*{3}*. This parameter is provided so that the user can change the default behavior without recompiling.

`select_candidates_default` – **integer** (`USER__CLOSE_TO_HALF_AND_EXPENSIVE`**{11}**). Determines the default rule for selecting strong branching candidates. Other values are `USER__CLOSE_TO_HALF`*{10}* and `USER__CLOSE_TO_ONE_AND_CHEAP`*{12}*. This parameter is provided so that the user can change the default behavior without recompiling.

`compare_candidates_default` – **integer** (`LOWEST_LOW_OBJ`**{1}**). Determines the default rule for comparing candidates. See the description of `user_compare_candidates()` for other values. This parameter is provided so that the user can change the default behavior without recompiling.

`select_child_default` – **integer** (`PREFER_LOWER_OBJ_VALUE`**{0}**). Determines the default rule for selecting the child to be processed next. For other possible values, see the description `user_select_child()`. This parameter is provided so that the user can change the default behavior without recompiling.

`mc_find_supported_solutions` – **boolean** (`FALSE`). By default, `sym_mc_solve` routine will find all the non-dominated solutions if the problem to be solved is a bicriteria problem. However, if the user plans to find only the supported solutions, then, this parameter has to be set before calling `sym_mc_solve` routine.

`mc_rho` – **double** (`0.00001`). The value used in augmented Chebyshev norm during the bicriteria solution procedure.

`generate_cgl_cuts` – **boolean** (`TRUE`). Whether or not to generate cuts using COIN's cut generation library. Note that, to use CGL cuts, OSI interface has to be used and moreover the corresponding flags have to be set during installation. See the makefile for more details.

`generate_cgl_gomory_cuts` – **boolean** (`TRUE`). Whether or not to generate Gomory cuts using COIN's cut generation library.

`generate_cgl_knapsack_cuts` – **boolean** (`TRUE`). Whether or not to generate knapsack cover cuts using COIN's cut generation library.

`generate_cgl_oddhole_cuts` – **boolean** (`TRUE`). Whether or not to generate generalized odd hole cuts using COIN's cut generation library.

`generate_cgl_probing_cuts` – **boolean** (`TRUE`). Whether or not to generate probing cuts using COIN's cut generation library.

`generate_cgl_clique_cuts` – **boolean** (`TRUE`). Whether or not to generate clique cuts using COIN's cut generation library.

`generate_cgl_flow_and_cover_cuts` – **boolean** (`FALSE`). Whether or not to generate flow and cover cuts using COIN's cut generation library.

`generate_cgl_rounding_cuts` – **boolean** (`FALSE`). Whether or not to generate simple rounding cuts using COIN's cut generation library.

`generate_cgl_lift_and_project_cuts` – **boolean** (`FALSE`). Whether or not to generate lift-and-project cuts using COIN's cut generation library.

7.4.6 Cut Generator Parameters

CG_verbosity – **integer (0).** Verbosity level for the cut generator module.

7.4.7 Cut Pool Parameters

CP_verbosity – **integer (0).** Verbosity of the cut pool module.

cp_logging – **boolean (0).** Determines whether the logging option is enabled. In this case, the entire contents of the cut pool are written out periodically to disk (at the same interval as the tree manager log files are written). If this option is set, then the line following must start with the keyword cp_log_file_name and include the appropriate file name as the value.

cp_warm_start – **boolean (0).** Used to allow the cut pool to make a warm start by reading in a previously written log file. If this option is set, then the line following must start with the keyword cp_warm_start_file_name and include the appropriate file name as the value.

block_size – **integer (5000).** Indicates the size of the blocks to allocate when more space is needed in the cut list.

max_size – **integer (2000000).** Indicates the maximum size of the cut pool in bytes. This is the total memory taken up by the cut list, including all data structures and the array of pointers itself.

max_number_of_cuts – **integer (10000).** Indicates the maximum number of cuts allowed to be stored. When this max is reached, cuts are forcibly purged, starting with duplicates and then those indicated by the parameter delete_which (see below), until the list is below the allowable size.

min_to_delete – **integer (1000).** Indicates the number of cuts required to be deleted when the pool reaches it's maximum size.

touches_until_deletion – **integer (10).** When using the number of touches a cut has as a measure of its quality, this parameter indicates the number of touches a cut can have before being deleted from the pool. The number of touches is the number of times in a row that a cut has been checked without being found to be violated. It is a measure of a cut's relevance or effectiveness.

delete_which – **integer (DELETE_BY_TOUCHES{2}).** Indicates which cuts to delete when purging the pool. DELETE_BY_TOUCHES indicates that cuts whose number of touches is above the threshold (see touches_until_deletion above) should be purged if the pool gets too large. DELETE_BY_QUALITY{1} indicates that a user-defined measure of quality should be used (see the function user_check_cuts in Section7.3.4).

check_which – **integer (CHECK_ALL_CUTS{0}).** Indicates which cuts should be checked for violation. The choices are to check all cuts (CHECK_ALL_CUTS{0}); only those that have number of touches below the threshold (CHECK_TOUCHES{2}); only those that were generated at a level higher in the tree than the current one (CHECK_LEVEL{1}); or both (CHECK_LEVEL_AND_TOUCHES{3}). Note that with CHECK_ALL_CUTS set, SYMPHONY will still only check the first cuts_to_check cuts in the list ordered by quality (see the function user_check_cut).

`cuts_to_check` – **integer (1000).** Indicates how many cuts in the pool to actually check. The list is ordered by quality and the first `cuts_to_check` cuts are checked for violation.

7.4.8 C++ Interface/OSI Parameters

As the implementation of the whole interface, there exists a matching C interface parameter to each of the C++ Interface/OSI parameter and the parameter setting functions are designed to set the corresponding C interface parameter. Thus, we will just give a table of the parameter names, their C interface complements and the values they can be set to, rather than their detailed descriptions. For each parameter, the user can see the C interface complement for further explanation.

C++ Interface	**C Interface**	**Value**
OsiSymVerbosity	verbosity	-user defined-
OsiSymWarmStart	warm_start	-boolean-
OsiSymNodeLimit OsiMaxNumIteration OsiMaxNumIterationHotStart	node_limit	-user defined-
OsiSymFindFirstFeasible	find_first_feasible	-boolean-
OsiSymSearchStrategy	node_selection_rule	LOWEST_LP_FIRST HIGHEST_LP_FIRST BREADTH_FIRST_SEARCH DEPTH_FIRST_SEARCH
OsiSymUsePermanentCutPools	use_permanent_cut_pools	-boolean-
OsiSymGenerateCglGomoryCuts	generate_cgl_gomory_cuts	-boolean-
OsiSymGenerateCglKnapsackCuts	generate_cgl_knapsack_cuts	-boolean-
OsiSymGenerateCglOddHoleCuts	generate_cgl_oddhole_cuts	-boolean-
OsiSymGenerateCglProbingCuts	generate_cgl_probing_cuts	-boolean-
OsiSymGenerateCglCliqueCuts	generate_cgl_clique_cuts	-boolean-
OsiSymGenerateCglFlowAndCoverCuts	generate_cgl_flow_and_cover_cuts	-boolean-
OsiSymGenerateCglRoundingCuts	generate_cgl_rounding_cuts	-boolean-
OsiSymGenerateCglLiftAndProjectCuts	generate_cgl_lift_and_project_cuts	-boolean-
OsiSymKeepWarmStart	keep_warm_start	-boolean-
OsiSymTrimWarmTree	trim_warm_tree * -boolean-	
OsiSymDoReducedCostFixing	do_reduced_cost_fixing	-boolean-
OsiSymMCFindSupportedSolutions	mc_find_supported_solutions	-boolean-
OsiSymSensitivityAnalysis	sensitivity_analysis	-boolean-
OsiSymRandomSeed	random_seed	-user defined-
OsiSymDivingStrategy	diving_strategy	BEST_ESTIMATE COMP_BEST_K COMP_BEST_K_GAP
OsiSymDivingK	diving_k	-user defined-
OsiSymDivingThreshold	diving_threshold	-user defined-
OsiSymGranularity	granularity	-user defined-
OsiSymTimeLimit	time_limit	-user defined-
OsiSymGapLimit	gap_limit	-user defined-
OsiObjOffset	-	-user defined-
OsiProbName	problem_name	-user defined-

However, as it is seen, only some of the C interface parameters have their matches. If the other parameters are required to be modified, the user can always set them directly by their C interface names, using the overlapping functions: `setSymParam(string, int), setSymParam(string, double) and setSymParam(string,string)`. For instance, the `verbosity` parameter can be set, let's say, to 2 either by setSymParam(OsiSymVerbosity, 2) or by setSymParam("verbosity", 2). Note that, this flexibility is also supported for parameter querying functions.

Bibliography

[1] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. CONCORDE TSP solver. http://www.tsp.gatech.edu/concorde.html.

[2] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. On the solution of traveling salesman problems. *Documenta Mathematica*, Extra Volume Proceedings ICM III (1998):645–656, 1998.

[3] E. Balas, S. Ceria, and G. Cornuéjols. Mixed 0-1 programming by lift-and-project in a branch-and-cut framework. *Management Science*, 42:1229–1246, 1996.

[4] C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh, and P. H. Vance. Branch and price: Column generation for solving huge integer programs. *Operations Research*, 46:316–329, 1998.

[5] M. Benchouche, V.-D. Cung, S. Dowaji, B. Le Cun, T. Mautor, and C. Roucairol. Building a parallel branch and bound library. In *in Solving Combinatorial Optimization Problems in Parallel, Lecture Notes in Computer Science* **1054**, page 201. Springer, Berlin, 1996.

[6] V. J. Bowman. On the relationship of the Tchebycheff norm and the efficient frontier of multiple-criteria objectives. In H. Thieriez, editor, *Multiple Criteria Decision Making*, pages 248–258. Springer, Berlin, 1976.

[7] Q. Chen, M. C. Ferris, and J. T. Linderoth. Fatcop 2.0: Advanced features in an opportunistic mixed integer programming solver. *Annals of Operations Research*, 103:17–32, 2001.

[8] J. Climaco, C. Ferreira, and M.E. Captivo. Multicriteria integer programming: an overview of different algorithmic approaches. In J. Climaco, editor, *Multicriteria Analysis*, pages 248–258. Springer, Berlin, 1997.

[9] C. Cordier, H. Marchand, R. Laundy, and L.A. Wolsey. bc-opt: A branch-and-cut code for mixed integer programs. *Mathematical Programming*, 86:335–353, 1999.

[10] ILOG CPLEX Division. `http://www.cplex.com`.

[11] J. Eckstein, C.A. Phillips, and W.E. Hart. PICO: An object-oriented framework for parallel branch and bound. Technical Report RRR 40-2000, Rutgers University, 2000.

[12] M. Ehrgott and X. Gandibleux. A survey and annotated bibliography of multiobjective combinatorial optimization. *OR Spektrum*, 22:425–460, 2000.

[13] M. Ehrgott and X. Gandibleux. Multiobjective combinatorial optimization—theory, methodology and applications. In M. Ehrgott and X. Gandibleux, editors, *Multiple Criteria Optimization—State of the Art Annotated Bibliographic Surveys*, pages 369–444. Kluwer Academic Publishers, Boston, MA, 2002.

[14] M. Ehrgott and M.M. Wiecek. Multiobjective programming. In M. Ehrgott, J. Figueira, and S. Greco, editors, *State of the Art of Multiple Criteria Decision Analysis*, Boston, MA, 2004. Kluwer Academic Publishers.

[15] P. K. Eswaran, A. Ravindran, and H. Moskowitz. Algorithms for nonlinear integer bicriterion problems. *Journal of Optimization Theory and Applications*, 63(2):261–279, 1989.

[16] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine*. The MIT Press, Cambridge, MA, 1994.

[17] B. Gendron and T. G. Crainic. Parallel branch and bound algorithms: Survey and synthesis. *Operations Research*, 42:1042–1066, 1994.

[18] A. M. Geoffrion. Proper efficiency and the theory of vector maximization. *Journal of Mathematical Analysis and Applications*, 22:618–630, 1968.

[19] A.Y. Grama and V. Kumar. State of the art in parallel search techniques for discrete optimization problems. *IEEE Transactions on Knowledge and Data Engineering*, 11:1–8, 1999.

[20] M. Grötschel, M. Jünger, and G. Reinelt. A cutting plane algorithm for the linear ordering problem. *Operations Research*, 32(6):1195–1220, 1984.

[21] K. Hoffman and M. Padberg. LP-based combinatorial problem solving. *Annals of Operations Research*, 4:145–194, 1985.

[22] T.H. Hultberg. FlopC++, 2004. Available from http://www.mat.ua.pt/thh/flopc/.

[23] M. Jünger and S. Thienel. The ABACUS system for branch and cut and price algorithms in integer programming and combinatorial optimization. *Software Practice and Experience*, 30:1325–1352, 2001.

[24] V. Kumar and V. N. Rao. Parallel depth-first search, part II: Analysis. *International Journal of Parallel Programming*, 16:501–519, 1987.

[25] J. Linderoth. *Topics in Parallel Integer Optimization*. PhD thesis, School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, GA, 1998.

[26] R. Lougee-Heimer. The Common Optimization INterface for Operations Research. *IBM Journal of Research and Development*, 47:57–66, 2003.

[27] A. Makhorin. Introduction to GLPK, 2004. Available from http://www.gnu.org/software/glpk/glpk.html.

[28] A. Martin. Integer programs with block structure. Habilitation Thesis, Technical University of Berlin, Berlin, Germany, 1998.

[29] G.L. Nemhauser, M.W.P. Savelsbergh, and G.S. Sigismondi. MINTO, a Mixed INTeger Optimizer. *Operations Research Letters*, 15:47–58, 1994.

[30] G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization*. Wiley, New York, 1988.

[31] M. W. Padberg and G. Rinaldi. A branch and cut algorithm for the solution of large scale traveling salesman problems. *SIAM Review*, 33:60–100, 1991.

[32] T.K. Ralphs and L. Ladányi. *SYMPHONY Version 4.0 User's Manual*, 2004. http://www.brandandcut.org.

[33] T.K. Ralphs, M.J. Saltzman, and M.M. Wiecek. An improved algorithm for biobjective integer programming and its application to network routing problems. To appear in Annals of Operations Research, 2004.

[34] V. N. Rao and V. Kumar. Parallel depth-first search, part I: Implementation. *International Journal of Parallel Programming*, 16:479–499, 1987.

[35] Y. Shinano, K. Harada, and R. Hirabayashi. A generalized utility for parallel branch and bound algorithms. In *Proceedings of the 1995 Seventh Symposium on Parallel and Distributed Processing*, pages 392–401, Los Alamitos, CA, 1995. IEEE Computer Society Press.

[36] R. Solanki. Generating the noninferior set in mixed integer biobjective linear programs: an application to a location problem. *Computers and Operations Research*, 18:1–15, 1991.

[37] S. Tschoke and T. Polzer. *Portable Parallel Branch and Bound Library User Manual: Library Version 2.0*. Department of Computer Science, University of Paderborn.

[38] Y. Xu, T.K. Ralphs, L. Ladányi, and M.J. Saltzman. ALPS: A framework for implementing parallel search algorithms. to appear in the Proceedings of the Ninth Conference of the INFORMS Computing Society, 2004.