A Framework for Fast Proto-typing of Meta-heuristics Hybridization

Hoong Chuin LAU[1] Wee Chong WAN[2] Steven HALIM[3] Kaiyang TOH[2]

[1]*School of Information Systems, Singapore Management University*
[2]*The Logistics Institute – Asia Pacific, National University of Singapore*
[3]*School of Computing, National University of Singapore*

Abstract: Hybrids of meta-heuristics have been shown to be more effective and adaptable than their parents in solving various combinatorial optimization problems. However, hybridized schemes are more tedious to implement due to their complexity. We address this problem by proposing the *Meta-heuristics Development Framework (MDF)*. In addition to being a framework that promotes reuse to reduce developmental effort, the key strength of MDF lies in its ability to model meta-heuristics using a "*Request, Sense and Response" (RSR) schema*, which decomposes algorithms into a set of well-defined modules that can be flexibly assembled through an intelligent central controller. Under this scheme, hybrid schemes become an event-based search that can adaptively trigger a desired parent's behavior in response to search events. MDF can hence be used to design and implement a wide spectrum of hybrids with various degrees of collaboration thereby offering the algorithm designer quick turnaround in designing and testing his meta-heuristics. This is illustrated in this paper through the construction of hybrid schemes using *Ants Colony Optimization (ACO)* and *Tabu Search (TS)*.

Keywords: Meta-heuristics, Hybridization, Software Framework, Reusability, Combinatorial Optimization, Ant Colony Optimization, Tabu Search.

1. INTRODUCTION

Meta-heuristics are an increasingly important in solving large-scale NP-hard combinatorial optimization problems. Other than computational efficiency, another key factor that leads to widespread adoption is their genericity – that meta-heuristics can be easily adapted to solve new problems (in contrast to specialized heuristics or exact methods which are often tailored to a specific problem). Yet another strong point of meta-heuristics is *hybridization*, in which a meta-heuristic is combined with other solution approaches (such as other meta-heuristics, constraint programming or exact methods). Hybridization has shown itself to be a promising field of research in recent years with encouraging results demonstrating their ability to outperform their parent heuristics (for example, [Bent and Hentenryck, 2004]; [Krasnogor and Smith, 2005]; [Resende and Riberio, 2003]; and [Vasquex and Vimont, 2005]), and the flexibility with which they can be adapted to solve new problems (for example, hyper-heuristics [Burke et al, 2003]).

The motivation behind hybrid approaches lies in the *No Free Lunch (NFL)* [Wolpert and Macready, 1997] theorem that no single meta-heuristic is superior to all other meta-heuristics in solving all problems. Thus, every meta-heuristic has their distinct advantages and disadvantages when applied to a given problem. As such, the goal of a hybrid meta-heuristic algorithm is to exploit the strengths of the techniques used in the hybrid to compensate for the weaknesses of others. For example, hybrid schemes may seek to strike a balance between *diversification* and *intensification* as meta-heuristics are strong in one aspect, but weak in the other.

However, a major stumbling block in the development of hybrid meta-heuristics approaches is the intensive development effort required as compared to conventional meta-heuristics, due to the need to develop multiple solution techniques and a means for the different techniques to work with each other. This issue is exacerbated by the fact that hybrid methods are often developed independently by different researchers with little or no code reuse, even for hybrid approaches sharing a common structure (e.g. memetic algorithms). Redeveloping and

validating the correctness of hybrids from scratch is often costly and time-consuming which discourages researchers from exploring new schemes.

One means of reducing development time is by using software frameworks, defined by [de Champeaux et al, 1993] as "medium-scale, multipurpose, reusable class hierarchies that depend only on the abstract interfaces of various components" and which have "been proven to be valuable tools for simplifying and accelerating further design". Recognizing this fact, the community has seen proposals for several frameworks for meta-heuristic development. For example, local search frameworks like *EASYLOCAL++* [Di Gaspero and Schaerf, 2003] and *Searcher* [Andreatta et al, 2002] allow users to rapidly implement their desired local search meta-heuristics by providing a generic structure which can be customized as required.

More advanced and generic frameworks like *HOTFRAME* [Fink and Voß, 2002], *HSF* [Dorne and Voudouris, 2004] and *HeuristicLab* [Wanger and Affenzeller, 2004] seek to allow end users to easily use meta-heuristics implemented by the framework to solve new problems and to develop new meta-heuristic approaches. In particular, these frameworks are object-oriented to allow users to easily customize framework components for their needs via inheritance. For example, *HOTFRAME* supports iterated local search, variants of SA, TS and evolutionary methods and some predefined hybrids. *HOTFRAME* also provides mechanisms for the end user to extend the framework to implement new search strategies, including hybrids.

While the above frameworks excel in reducing the developmental effort in implementing standard meta-heuristics models and their variations, they suffer from the common drawback that they were not designed to directly support the formulation and implementation of hybrid schemes. Thus, implementing hybrids with these frameworks would often require extensive modification of framework codes, which in turn requires users to have an in depth knowledge of the workings of the framework, consequently negating the black-box advantage. Hence to foster hybridization among meta-heuristics, there is a need for a framework that reduces the conceptual gap between the algorithmic formulation and actual implementation. Such a framework,

modeling the entire process of optimization, would form the link between disparate algorithms, enabling the easy creation of hybrids.

For this purpose, we propose the *Meta-heuristics Development Framework (MDF)*, a C++ software framework that bridges the gap of hybrid modeling and software implementation. One primary goal of MDF is to provide a methodology and platform that facilitates the rapid prototyping of meta-heuristic hybrid schemes. This methodology is particularly effective in algorithm prototyping of *new* problems when the designer needs to experiment with different algorithm strategies quickly so as to find one that performs best.

MDF achieves this goal by its modeling schema, which formulates the search algorithm as a *"Request-Sense-Response"* search (*RSR schema*). The schema works by decomposing the hybrid behavior into a set of standalone components known as the *Responses*. These components are then executed during the search when the framework *Senses* certain search events, which is dependent on the problem instance. The *Requests* are a list of rules designed by the user to match each response to an event, thus indirectly controlling the search process. In another words, MDF models hybrids as a search with events and handlers (or *responses*) that can be adaptively combined in accordance to the user's requests.

We like to remark early in the paper that hybrid schemes offer the advantage that they are not specific to a problem, and can hence be easily implemented and reused. The tradeoff however lies in their run-time efficiency and solution effectiveness when compared with specialized heuristics tailored to solve specific problems. For example, there are well-known heuristics such as the Iterative Local Search on Lin-Kernighan heuristic for TSP, which performs very well in terms of efficiency and effectiveness [Johnson and McGeoch, 1997]. Such specialized heuristics often require deep domain knowledge such as the structure of the problem that takes time to discover. In this paper, our intent is not to devise hybrid algorithms that beat the best-published results for specific problems, but rather, we want to provide a framework that allow algorithm designers to powerfully express hybrid schemes and perform rapid proto-typing (or debugging),

of his meta-heuristic algorithms.

This paper proceeds as follows. In Section 2, we will present an overview of the MDF architecture. We present our proposed *RSR schema* for the modeling of hybrids in Section 3. Section 4 will illustrate the use of MDF to implement a hybrid scheme of [Stuetzle and Dorigo, 1999] for solving the Traveling Salesman Problem (TSP). Section 5 explores the use of MDF to design and implement generic hybrid schemes, particularly those categorized under [Talbi, 2002]'s hybrid taxonomy. We will illustrate with TSP again, using various hybrid schemes of between TS and ACO. Section 6 will present the results of the experimentation on TSP, with particular emphasis on the cost required to implement the hybrid schemes vs. the improvement in performance attained. Finally, we present the conclusion to this paper in Section 7.

2. MDF ARCHITECTURAL OVERVIEW

The design of MDF began with the work of Tabu Search Framework (TSF) [Lau et al, 2003a], which uses abstraction and inheritance as the primary mechanism to build adaptable components and interfaces. In [Lau et al, 2004a], the idea of a Meta-Heuristic Framework (MDF) was proposed that supports *ACO, TS, SA* and *GA*. Architecturally, MDF is composed of 4 major components, *General Interfaces*, *Proprietary Interfaces, Engines* and *Control Mechanism*, as shown in Figure 2.1. The first three components are the essential building blocks for a standard meta-heuristic application, while the special consideration to the design of Control Mechanism is given to support hybridization. In this section, we will briefly explain the first three components. Interested readers may refer to [Lau et al, 2003a] for details. For the Control Mechanism, [Lau et al, 2004a] presented a rudimentary design, and this paper presents a refined version in Section 3.

2.1 General Interfaces

General Interfaces represent concepts (e.g. solution representation, move, etc.) which are common to all meta-heuristic methods as abstract interfaces. These interfaces are not tied to any

specific problem, thus allowing users freedom to implement their preferred representation. For example in TSP, the solution can be represented as an array of integers denoting the sequence of cities in the tour or as an adjacency matrix that traces the path taken by the salesman. More complicated solution representation such as a splay tree can also be extended from the *Solution Interface* without affecting the framework. By using the principle of abstraction, the framework is able to modify the solution object with other inherited objects without any assumption on the solution structure. Similarly a user can implement different *Move* strategies and apply them to the same *Solution* object. More importantly, by having a common solution interface between various meta-heuristics, it facilitates the notion of hybridization in which various parent algorithms can modify the same solution structure. Thus *General Interfaces* promote code reuse without sacrificing flexibility.

2.2 Proprietary Interfaces

Proprietary Interfaces allow for the customization of the unique behaviors exhibited by each meta-heuristic. For instance, ACO has two such interfaces: *local heuristic* and *pheromone trail* and TS also has two such interfaces: *tabu list* and *aspiration criteria* and so forth for the other meta-heuristic engines in MDF. Unlike the *General Interfaces*, *Proprietary Interfaces* are exclusive to the meta-heuristic they belong to and hence need not be implemented unless the user wishes to implement that meta-heuristic. An exception to this is during hybridization, in which a proprietary behavior is crossed into the routine of another meta-heuristic.

2.3 Resource Container & Engines

The *Engine* classes make use of the General and Proprietary Interfaces to implement specific standard meta-heuristic search algorithms. For example in the *TS Engine*, the *Neighborhood Generator* object is invoked to generate the neighbors based on the *Move* object(s) around the current *Solution* object. It then apply the *Objective Function* and *Penalty Function*

objects to evaluate each neighbor and then select the best neighbor based on the *Tabu List* and *Aspiration Criteria* objects. The new solution is updated into the current *Solution* object and the iterations continue till a terminating condition (such as 1000 iterations) is reached. In short, the Engine can be seen as the "worker" that arrange the interfaces into execution blocks in which the sequence modeled the behavior of the meta-heuristic.

In addition, the parameters of the search (such as tabu tenure) are stored in a data structure class called the *Setting* instead of the engine itself. Multiple *Settings* from different engines are stored collectively in a "*Resource Container*". This centralization design allows fast access and easy modification on the parameters, either manually or through the *Control Mechanism*.

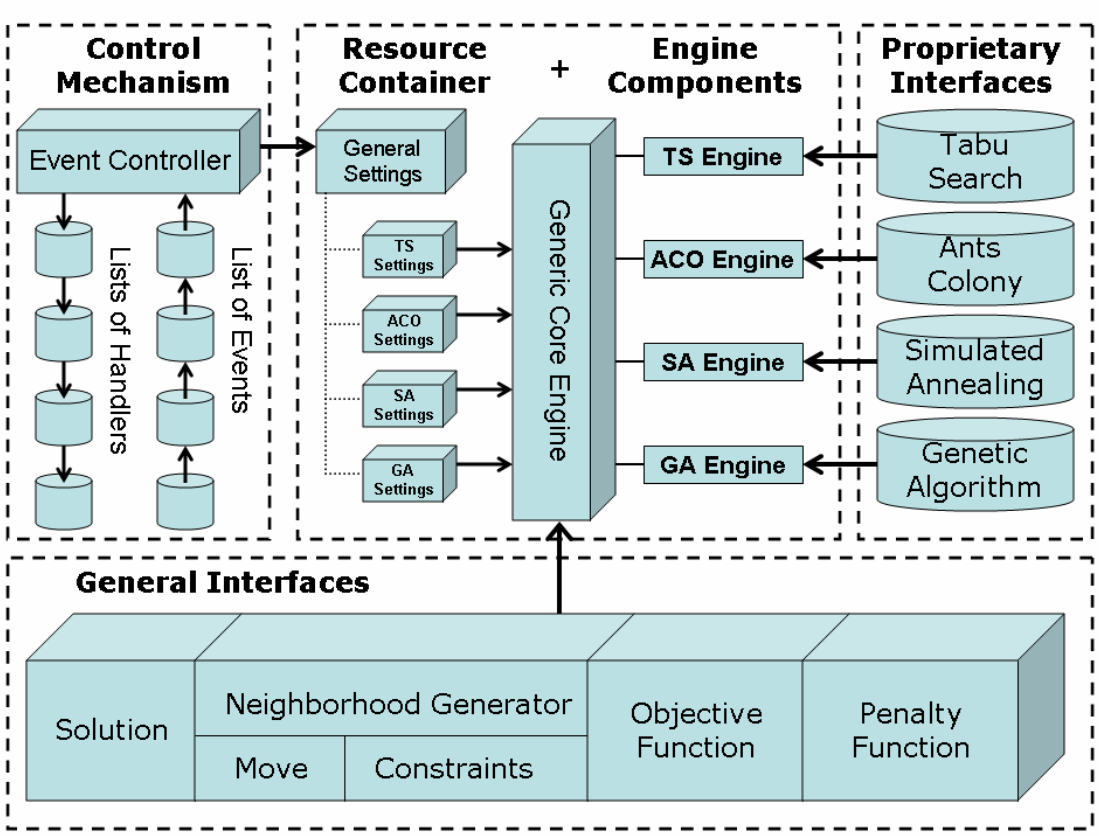

Figure 2.1: Architecture of Meta-heuristics Development Framework

The first three components are adequate in modeling and implementing standard meta-

heuristics. To facilitate hybridization, MDF is designed with a *Control Mechanism* that uses the *RSR schema* to formulate hybrid schemes. The next section will provide a detailed description of the design of the Control Mechanism for hybridization.

3. RSR MODELING OF HYBRIDIZATION

An observation we make is that a meta-heuristic is often hybridized with another algorithm, such as another meta-heuristic, to improve its intensifying or diversifying capability (e.g.: [Burke et al, 2001], [Talbi, 2002], [Bent and Hentenryck, 2004]**).** While intensification and diversification may work differently, they are similar in that they are both applied to adjust the search trajectory when specific search situations (or events) are detected during the search. As such, we term any adjustment to search (such as any modification to the current solution or the change of search parameters) as a *Response* due to a *Sensed* Event. A *Request* is then defined as the rule that matches a particular sensed event to a response. It is not difficult to see that while the responses are static, their actual executions are dependent on the occurring search events and thus dynamic to each problem instance. We view this *Request, Sense and Response (RSR)* scheme as an event-driven schema, which is an event loop where a *response* is triggered when it *senses* the occurrence of an event based on *requests* defined by the user (as illustrated in Figure 3.1).

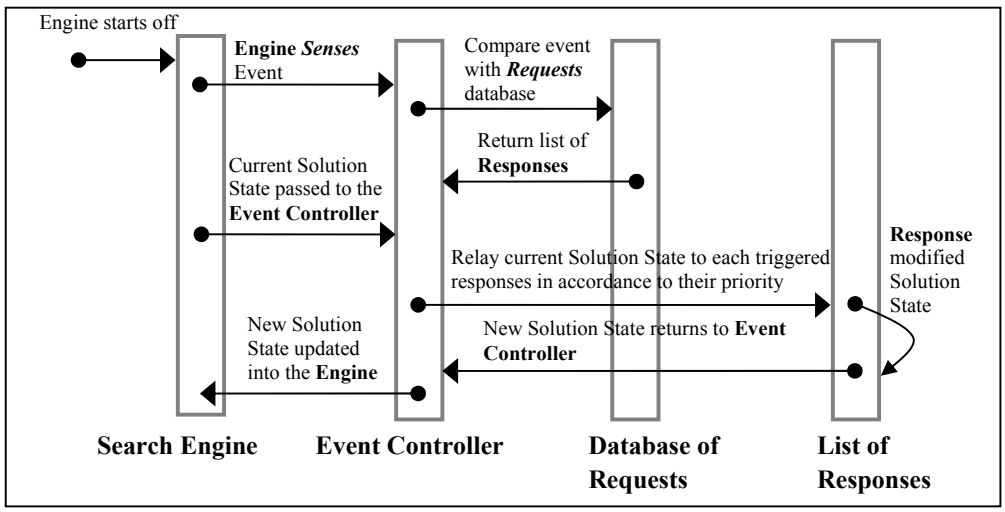

Figure 3.1: Sequence diagram illustrating the *RSR schema*.

More formally, a hybrid scheme can be represented by its present *search state* (S_a), a set of *events* $X=\{\chi_1, \chi_2, \ldots \chi_m\}$, a set of *handlers* $\Psi=\{\psi_1, \psi_2, \ldots \psi_n\}$ and a set of *requests* $R=\{r_1, r_2, \ldots r_q\}$. A search state ($S$) defines the state at a time instant of the search process and consists of the tuple (*Current Solution*, *Best found Solution*, *Search parameter*).

An event (χ) represents a decision point, where the occurrence of a given condition (e.g. new best solution found) in the search process indicates that some responses (carried out by the handlers) may be undertaken. Events can be categorized into 3 types: problem specific, algorithm specific and general. Problem specific events are specific to the problem being considered, e.g. a specific path being chosen for a TSP solution. Algorithm specific events occur at specific points during the execution of a meta-heuristic algorithm. An example of such an event is the completion of the crossover step in a GA. General events are those that do not fall under the other 2 categories, and are usually applicable to all problems and algorithms (e.g. New best solution found or a series of 10 consecutive non-improving moves made). MDF provides support for all 3 types of events and we will illustrate an example in Section 4.

A handler (ψ) describes an action, algorithm or strategy which is to be carried out in response to an event. It can be viewed as a function that modifies the present search state S_a to obtain the next search state S_{a+1}. Handlers can implement user defined strategies (e.g. problem specific heuristics) or make use of MDF's built-in engines to respond to events. Each handler has a priority level (from highest to lowest priority: instant, normal, low) which determines the order of execution when multiple handlers are awaiting execution.

A request (r) is defined by a pair (χ, ψ), which states that the handler ψ is the response to be carried out when the engine senses the occurrence of event, χ. Note that a "multiple-to-multiple" relationship where multiple events can be triggered simultaneously and multiple handlers can be executed in response to these triggered events is allowed. Requests have to be defined by end users in prior to the search.

A hybrid search hence begins with a set of requests R, an initial state S_0 and a main meta-

heuristic ψ_0 (usually the primary meta-heuristic of the hybrid). Execution proceeds by calling the *Main_Algorithm* with a recursive function *Execute_Handler* (see Figure 3.2).

```
Main_Algorithm(R, S0)     {
    Sa = S0;
    Ψ.Initialize (ψ0);     // Start off the main meta-heuristic
    while (Termination_Condition_Not_Met) {
        Execute_Handler (Ψ, Sa);
    }
}

Execute_Handler (Ψ, Sa) {
    while (Handler_Termination_Condition_Not_Met) {
        for each (ψ ∈ Ψ)     {
            Sa = ψ(Sa);
            if (X = Sense_Event(Sa))   {
                for each (χ ∈ X) {
                    Ψ = Event_Controller.Match_Requests(χ, R);
                    Sa = Execute_Handler (Ψ, Sa);
                }
            }
        }
    }
    return Sa;
}
```

Figure 3.2: A pseudo code of the *RSR schema*.

The `Handler_Termination_Condition_Not_Met` is specific for each handler and can range from instant termination after carrying out an operation (e.g. a "kick" operation handler) to running for a few iterations (e.g. running TS as a sub meta-heuristic of GA for 10 iterations) to running until the next event occurs (e.g. switching between meta-heuristic approaches).

The function `Event_Controller.Match_Requests(`χ`,R)` is a matching function which determines the handlers to be executed given the event χ and the list of predefined requests. More formally, for a given input search state S_a, a set of events X that correspond to S_a is first determined. For each matching event $\chi \in X$, the function returns the ordered set of responses Ψ, where for each $\psi \in \Psi$ and $\chi \in X$, there is a corresponding tuple (χ, ψ) in R. The responses in Ψ are ordered by their priority level. If two or more responses have the same priority

level, they will be arbitrarily ordered as it is assumed that the order of execution of handlers in the same priority level is unimportant. The function also ensures that any duplicate handlers (due to two events occurring triggering the same handler) are removed.

The motivation for the adaptation of such a scheme is that it supports hierarchical chains of requests. A chain occurs if a handler calls upon another handler (to assist it in performing optimization), thus forming a hierarchical level of control handling. With this design, multiple meta-heuristics can be deployed in a single search to form highly complex hybrid schemes. This design also allows for sequential processing of handlers avoiding the complexities involved with multi-threaded/distributed architectures. Different handlers are able to communicate with each other via storage of search parameters in the search state. In MDF, there are stored in *Settings* in the *Resource Containers*.

4. ILLUSTRATION

We now illustrate how the RSR schema can be implemented with MDF. Particularly, we consider the hybrid scheme proposed by [Stuetzle and Dorigo, 1999]. In this scheme, *ACO* and a simple *Local Search (LS)* were hybridized to solve TSP. The approach was to apply LS to the iteration-best solution before the ants update it into the pheromone trails. To implement this strategy, we first define an algorithm-specific event to detect the time point in which the iteration-best ant has found a solution. A suitable time point for this is the moment before the pheromone update. Figure 4.1 shows the sample codes for this event. The class `BEFORE_PHEROMONE_UPDATE_Event` inherits the interface *Event* defined in MDF. Under this interface, the derived class must implement the function `bool TriggerResponse (int MessageID, Solution* Current_Solution)`, The parameter `MessageID` reflects the event detected at the current time point, such as `BEST_NEW_FOUND_SOLUTION`, `NON_IMPROVING_SOLUTION`, `NON_FEASIBLE_SOLUTION`, `BEFORE_TABU_LIST_UPDATE`,

BEFORE_PHEROMONE_UPDATE. The second parameter is the solution at the current time point and mainly used by the user to implement problem-specific events (such as objective value above a certain threshold). In our example, we are watching for the *MessageID* to be "*BEFORE_PHEROMONE_UPDATE"*. We return *true* or *false* indicating whether a positive match has been made.

The next step is to implement the handler, which allows the LS to enhance the solution found by the ACO's best ants. Figure 4.2 shows the sample code for this handler. We see that the LS procedure is embedded in a derived class *LS_Handler*. Under the interface *Handler*, the derived class must implement *void Execute(Engine* Main_Engine)*. This function will be called by the Event Controller if the respective event ("*BEFORE_PHEROMONE_UPDATE"*) is detected. The parameter in this function is a pointer to the main engine (which is `ACO_Engine` in our example), and through this pointer, we can get a reference to the `solution` object. In addition to keeping a reference to the `solution` object, other objects such as the `pheromone trail` and `penalty function` are also accessible via the engine pointer, to cater for responses that required modification these objects (such as ACO's search parameters).

Finally we need to define a *Request* which matches our defined event to the implemented handler as shown in Figure 4.3. With the two implemented classes *BEFORE_PHEROMONE_UPDATE_Event* and *LS_Handler*, and the control codes for the event controller (*Request*), the implementation of the hybrid in MDF is complete.

```
class BEFORE_PHEROMONE_UPDATE_Event : Event     {
   bool TriggerResponse (int MessageID, Solution*
   Current_Solution)     {
         If (MessageID == BEFORE_PHEROMONE_UPDATE)
               return true;
         return false;
   }
}
```

Figure 4.1: An illustration of a MDF event.

```
class LS_Handler : Handler    {
   void Execute (Engine* main_engine)    {
      ACOEngine* ACO = (ACOEngine) main_engine;
       Solution* current_solution = ACO->GetCurrentSolution();
      LS_PROC (current_solution);
      ACO->SetCurrentSolution (current_solution);
   }
   // Local Function
   void LS_PROC (current_solution)  {
     // LS procedure optimizing on the current_solution
         …
   }
}
```

Figure 4.2: An illustration of a MDF handler.

```
#include "BEFORE_PHEROMONE_UPDATE_Event.h"
#include "LS_Handler.h"

class HYBRID_EventController : EventController  {
   void Initialize_Request () {
     BEFORE_PHEROMONE_UPDATE_Event evt = new
         BEFORE_PHEROMONE_UPDATE_Event ();
      LS_Handler hnd = new LS_Handler ();
      ADD_REQUEST (evt, hnd, NORMAL);   // Normal priority
   }
}
```

Figure 4.3: An illustration of a Request code in MDF Event Controller.

5. IMPLEMENTATION OF GENERIC HYBRID SCHEMES

This section demonstrates how generic hybrid schemes can be implemented using MDF. Generic hybrid schemes are problem-independent, and they can be readily customized to solve a given optimization problem when the corresponding General Interfaces have been implemented. The advantage of generic hybrid schemes is that they often require little or no problem domain knowledge from the users and can be easily implemented, in contrast with specialized approaches that capitalize on the structure of the problem, which takes time and expertise to discover [Burke et al, 2003]. Arguably, the tradeoffs of these schemes are slower run-time and reduced effectiveness. These are not our concerns since the goal of MDF is to facilitate fast proto-typing and not to devise new tailored approaches. In this section, we focus our attention on how generic

schemes can be readily designed and implemented in MDF.

One approach in forming generic schemes is based on the taxonomy proposed in [Talbi, 2002]. In this work, meta-heuristic hybrids were classified into four categories as illustrated in Figure 4.1. The four categories are *High Level Relay* **[HLR]**, *High Level Teamwork* **[HLT]**, *Low Level Relay* **[LLR]** and *Low Level Teamwork* **[LLT]**, and they formed the fundamental in which generic hybrid schemes can be derived. In [Lau et al, 2003b], the authors followed the same taxonomy to derive the *Hybrid Ants System Tabu Search* (*HASTS*) schemes that represent different hybrid schemes of ACO with TS. Four hybrid models were proposed, namely *Empowered Ants (HASTS-EA)* **[LLT]**, *Intensification Exploitation (HASTS-IE)* **[LLT]**, *Enhanced Diversification (HASTS-ED)* **[LLR]** and *Collaborative Coalition (HASTS-CC)* **[HLR].** These schemes were used to solve the *Inventory Routing Problem with Time Windows*.

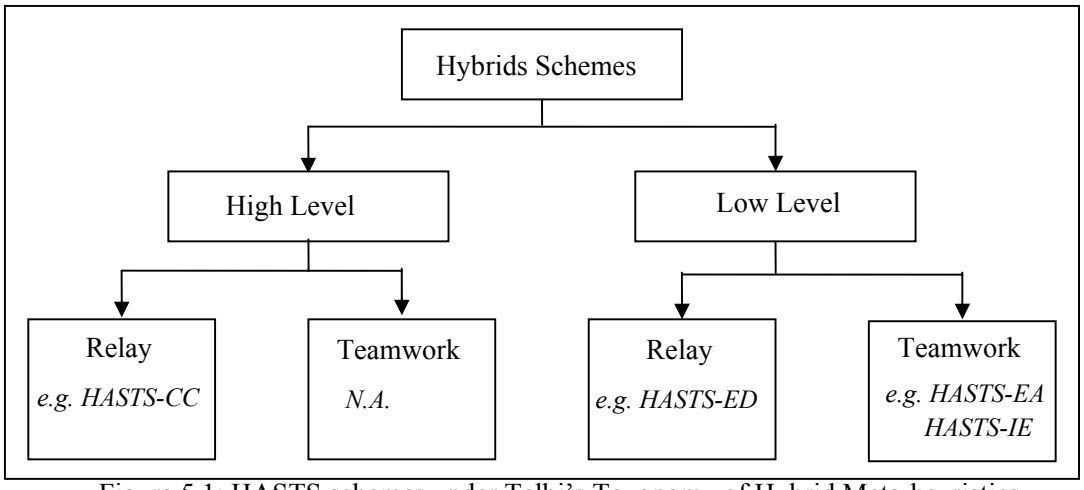

Figure 5.1: HASTS schemes under Talbi's Taxonomy of Hybrid Meta-heuristics.

In the following, we will use these generic schemes to solve TSP. Arguably, many benchmark instances of TSP have been solved to optimality and it is unlikely for these generic schemes to achieve better results. Our purpose is not to achieve better results, but to demonstrate how MDF is a ready platform for implementing HASTS. What we like to measure is the cost-efficiency of creating hybrids under the assumption that their parent algorithms have been implemented on MDF. We argue that this measure enables us to predict the expected gain from exploiting the RSR scheme to implement hybrid schemes on a different problem. In addition, we

also illustrate the ease of creating *hyper-hybrids* (i.e. hybridizing hybrids). We found that these hyper-hybrids improve the quality of solutions for most cases and the additional development cost is relatively insignificant. Hence, under MDF, complex hyper-hybrids can be generated rapidly, offering the algorithm designer quick turnaround to test the effectiveness of his approach.

Strict Tabu Search (Pure TS)

We restrict our implementation of TS to the *strict TS* procedure described in [Glover and Laguna, 1997]. The *Move* used in TS is a simple swap-edge operator that exchanges two arbitrary edges in the tour. Unlike the "fast" two-opt operation used by many local search approaches [Johnson and McGeoch, 1997], this operator does not restrict the generated neighbors to only those that are improving. Rather, to facilitate the notion of accepting a bad neighbor (as described by Glover), $O(n^2)$ neighbors are generated in each iteration. While this reduces the efficiency of the algorithm, ironically this simple algorithm is better suited to our experiment in observing the effect of combining naïve approaches. Two proprietary interfaces are required to elicit the essence of TS: *Tabu List* and *Aspiration Criteria*. In our case, we tabu the swapped edges and apply a static tenure that is equal to the instance size. The best aspiration criterion is used to over-ride the tabu status of a solution if a better objective value is found.

Ants Colony Optimization (Pure AC)

The *ACO* algorithm is implemented with the settings of $m = 25$ ants, $\rho = 0.2$, $\alpha = 1.0$, $\beta = 2.0$, and candidate list of size 20 as proposed in [Stuetzle and Dorigo, 1999]. While this is not within the context of our discussion, it is interesting to mention that the actual development time for ACO is much less than expected as it benefits from the code reuse in the TS implementation. *Solution* and *Objective Function* for example, has the same formulation in both implementations, and hence can be reused. In addition, as both applications (ACO and strict TS) use the same *Solution* interface, hybridization can be easily performed without further alteration. This illustrates the strengths in MDF; fast proto-typing and flexibility in hybridization. Other

proprietary interfaces include the *Local Heuristic*, which computes the inverse of edge length and the *Pheromone Trail* is a two dimensional table that records the history of the best ants.

Empowered Ants (HASTS-EA)

To reduce the possibility of over-convergence in ACO, this hybrid scheme empowers the ants with memory (*tabu list*) so as to reduce possibility of reconstructing similar solution by ants in the same iteration. The procedure of *HASTS-EA* is described in Figure 5.2. The tabu list is added to record some random edges traveled by the first ant. These edges are then "tabu-ed" to prevent *d* subsequent ants in the iteration to move along them. At the end of the iteration, the tabu list is reset. The tabu list also eliminates the need for local pheromone decay, reducing one of the original ACO's parameters. To implement HASTS_EA, the *Neighborhood Generator* object in the ACO is modified to include a tabu list. For each ant in the iteration, when the *Neighborhood Generator* generates the list of unvisited edges, the "tabu-ed" edges are removed from the list. The aspiration criterion for the tabu list is when no other unvisited edges can be found.

HASTS-EA procedure

1) (Initialization)

Initialize Pheromone Density Table τ and Local Greedy Heuristic η

2) (Construction)

For each ant *k*, **do**

Choose probabilistically a new unvisited path *j* based on τ and η

For each new path *j*, **do**

Consult Tabu List and Aspiration Criterion on *j*

Until a non-tabu-ed *j* is found

Until ant *k* completed its solution

Update Tabu List with *p* random edges in solution found by ant *k*

3) (Trail Update)

Update paths in best ant solution into τ

4) (Terminating Conditions)

If not (end_conditions**), goto *step 2***

Figure 5.2: Pseudo code of HASTS-EA.

Improved Exploitation (HASTS-IE)

In this scheme, the conventional ACO is hybridized with TS to improve the solution found by each iteration best ant. The improved solution from TS is then returned to ACO to be updated into the pheromone density table. To implement this hybrid scheme using MDF, we embed TS into a handler object. An event is then set to detect the time when a solution is found by *Iteration Best Ant*. When this event triggers, the *Event Controller* will pass the solution found by the iteration best ant into the handler, in which the embedded TS will optimize the solution. The terminating condition for the embedded TS is for it to reach 100 non-improving moves and the best result found by TS will be returned to the ACO Engine via the *Event Controller*. This new solution will replace the original solution found by the iteration best ant, and updated into the pheromone table (illustrated in Figure 5.3).

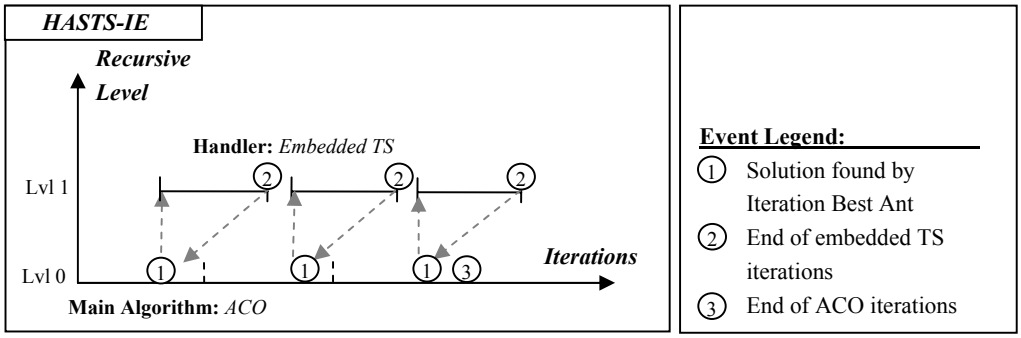

Figure 5.3: Timeline illustration of HASTS-IE.

Enhanced Diversification (HASTS-ED)

This hybrid scheme uses ACO as a diversification strategy for TS. When TS encounters a series of non-improving moves, a random connected portion of the constructed tour are injected into the implemented ACO for reconstruction. In a similar fashion with HASTS-IE, this hybrid schemes embeds ACO into a handler object. An event is then setup to count 100 non-improving moves encountered during the TS search. When the counter reaches a pre-defined value, the event will be triggered. The triggering of this event will result in the *Event Controller* passing the best found solution by TS into the handler. The handler class then randomly extracts a connected

portion of the tour and relocates the portion into the embedded ACO. The portion of tour is then re-optimized by ACO with an additional constraint that ensures that the terminating ends in the extracted portion will remain as the terminals during the reconstruction. After 100 iterations, the reconstructed portion of the tour is recombined back to the original tour and the newly diversified solution is returned back to the TS Engine, via the *Event Controller*, to replace the current solution in the TS Engine (illustrated in Figure 5.4).

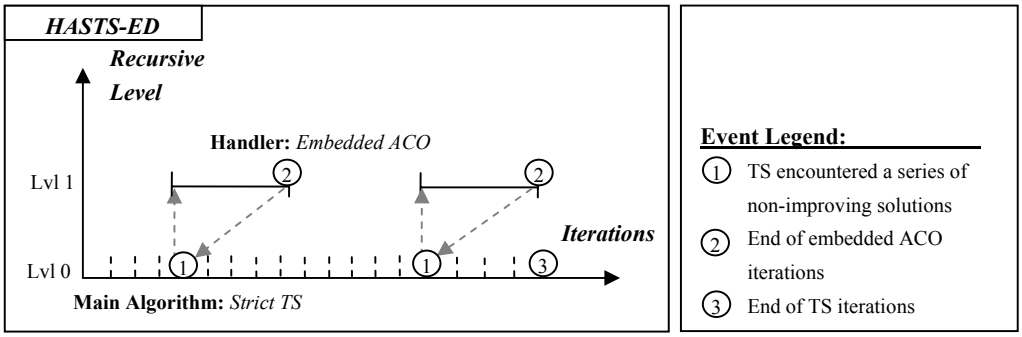

Figure 5.4: Timeline illustration of HASTS-ED.

Collaborative Coalition (HASTS-CC)

This hybrid scheme follows a 2-phase approach between ACO and TS, in which the ACO acts as constructing phase while the TS fit into the optimizing phase. To implement this scheme, we embedded the TS Engine into a handler object. We then set an event that detects the end of all iteration in ACO. When this event triggers, it indicates that ACO has completed all its iterations. The *Event Controller* will then pass the best found solution by the ACO Engine into the handler, which in turn places this solution as the initial solution for the embedded TS. Subsequently, the handler runs the TS Engine for some iterations, and returns to the *Event Controller* the optimized solution (illustrated in Figure 5.5).

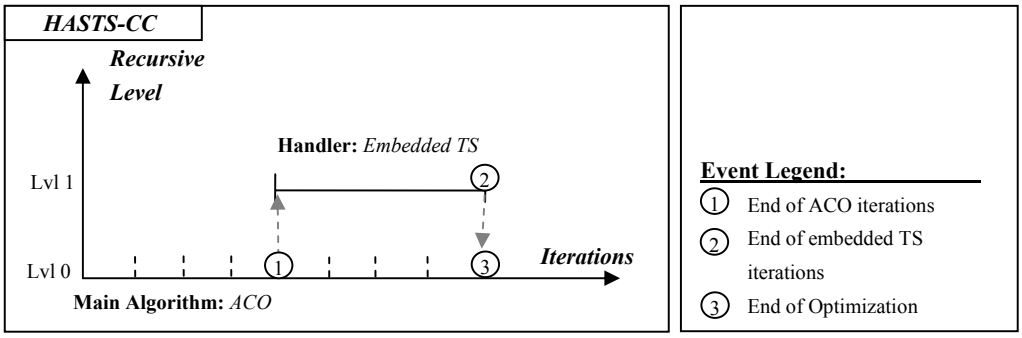

Figure 5.5: Timeline illustration of HASTS-CC.

Two Hyper-Hybrid Models

In addition, the four hybrid schemes can be combined easily to form hyper-hybrids. We continue to introduce two hyper-hybrid schemes, HASTS-CCED and HASTS-IEEA. In HASTS-CCED (illustrated in Figure 5.6), we replace the embedded TS in the handler with the implementation of HASTS-ED. This aims to enhance the optimizing phase. For HASTS-IEEA (illustrated in Figure 5.7), the event and the TS embedded handler of HASTS-IE is added into the implementation of HASTS-EA, thus allowing the overall design to develop a more aggressive diversifying capability. These hyper-hybrids have illustrated to us on the ease of forming complex hybrids from previously constructed applications. Initial experimentation of these hyper-hybrids has shown promising results with low additional development cost.

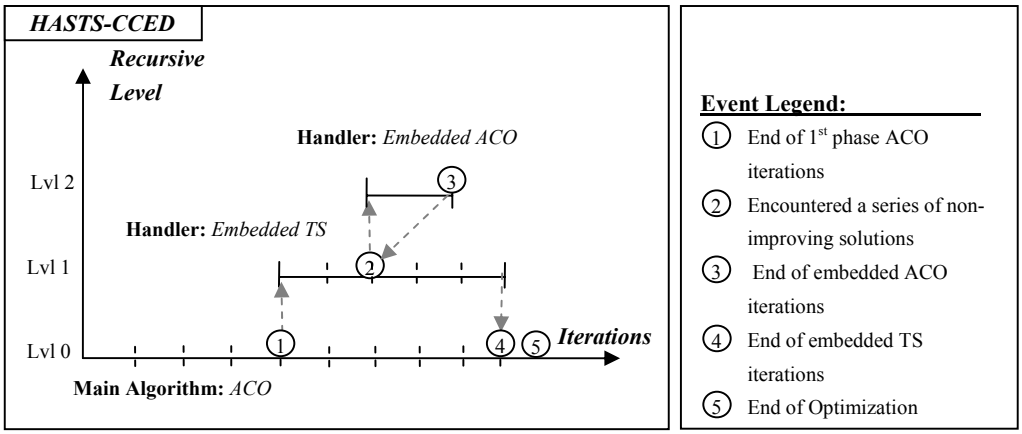

Figure 5.6: Timeline illustration of HASTS-CCED.

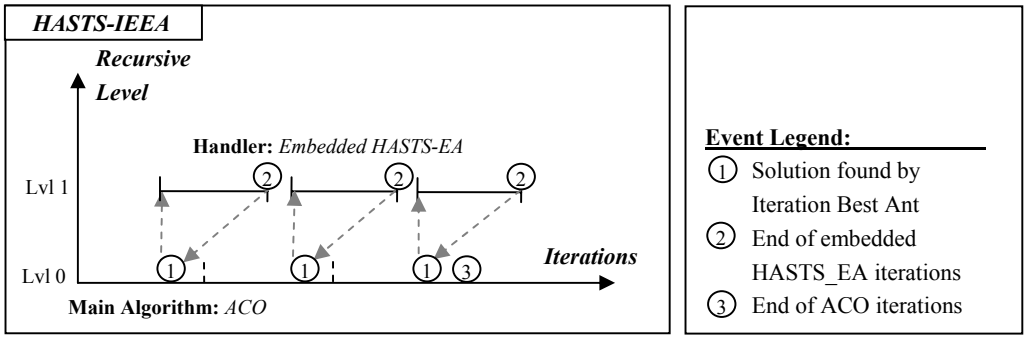

Figure 5.7: Timeline illustration of HASTS-IEEA.

6. EXPERIMENTAL RESULTS

We now present experimentally the cost-effectiveness of MDF in designing and implementing hybrids for TSP. The test problems are obtained from TSPLIB [Reinelt, 1991].

6.1 Development Cost

The most obvious incentive for using a framework is the reduction in development cost. It is however difficult to measure development cost directly due to a myriad of factors. We noted the work of [Park, 1992], in which the *Source Lines of Code* (SLOC) metric was proposed as an estimator for approximating the required effort. While the SLOC metric may not be adequate in predicting the factors such as differing programmer efficiencies and a lack of consideration for the debugging validation costs, it nevertheless provides a good estimation of the overall development cost. SLOC has been widely used in the software development industry (such as *COnstructive COst MOdel* (COCOMO) [USC-CSE, 2006] and *System Evaluation and Estimation of Resources - Software Estimating Model* (SEER-SEM) [Galorath, 2006]). As such, we adopt this metric in our experimentation as an approximation on the development efforts required to implement each of the hybrid schemes described in Section 5. The SLOC count for each approach was obtained by using the freely available *SLOCCount* [Wheeler, 2006] tool.

For the pure TS and AC approaches, SLOC counts the lines of code written to customize

the TS and AC engines in MDF for solving TSP. For the hybrid approaches, SLOC counts the lines of codes needed to implement the additional events and handlers required for that hybrid, under the condition that both pure TS and pure AC have already been implemented. For the hyper-hybrid HASTS-CCED, the 2-shade bar signifies that HASTS-CCED uses the same events and handlers as its parents (HASTS-CC and HASTS-ED) and can be implemented without writing any additional code. This is similar for the case of HASTS-IEEA.

From Figure 6.1, we can infer that most of the development time was spent developing the parent meta-heuristics (pure TS and AC). While MDF also offers some savings in development effort for these pure schemes as compared to developing from scratch, the real benefits of MDF can be seen in the effort required to implement the hybrids. The simple hybrids (HASTS-EA, HASTS-IE, HASTS-ED and HASTS-CC) can each be implemented in under 50 lines of code (almost an order of magnitude less than the development codes of their parent schemes). Remarkably, hybrid of hybrids (HASTS-CCED and HASTS-IEEA) can be implemented without writing *any* additional code.

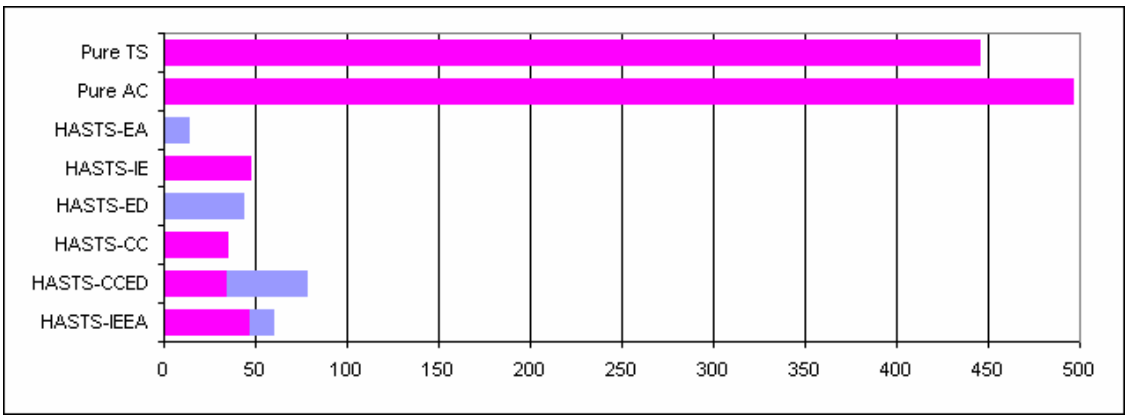

Figure 6.1: Approximation of development cost by SLOC

The results demonstrate the effectiveness of MDF as a framework for rapid prototyping of hybrid meta-heuristics, due to the effectiveness of maximizing code reuse. What is noteworthy is that for the hybrids implemented, the higher the level of hybridization required, the less actual implementation work (in terms of code to be written) needs to be done. Simple hybrids require far

less lines of additional code than pure approaches and hybrid of hybrids require less effort than simple hybrids to implement. Consequently, this strongly encourages developers to perform hybridization even on existing MDF-based applications, as the amount of effort will be greatly reduced, hence promoting the developers to experiment on their devised hybrid schemes.

6.2 Comparison of Effectiveness

All TSP test cases are run on different hybrids using an Athlon XP 3200+ processor with 512MB of memory, and the results are taken after 90 seconds regardless of the instance size. For each scheme, a greedy heuristic based on the nearest neighbor is used to construct the initial solution.

The results of 21 test cases from TSPLIB are recorded in Tables 6.1 and 6.2. For both tables, we measure the improvement in performance of each hybrid scheme with respect to their parent approaches – Table 6.1 with ACO and Table 6.2 with TS. A positive result indicates improvement while a negative result shows that the hybrid is actually performing poorer than its parent. We also include the comparison of the expected effectiveness of the hybrids per line of code (which we term as *Effective Gain*) under the tested benchmarks.

Except for the HASTS-EA scheme, the rest of the hybrids performed better than their parent, with most significant effective gain with the HASTS-IE scheme, thus showing the effectiveness of this scheme. However, if we assume that HASTS-EA and HASTS-IE have been constructed, then the additional effort to construct HASTS-IEEA will be negligible (instead of the recorded 61 lines, which is the sum of HASTS-EA (14 lines) and HASTS-IE (47 lines)). Under this condition, HASTS-IEEA will emerge as the hybrid scheme with the best effective gain. One noticeable trend is that approaches like HASTS-CC and HASTS-EA that use pure AC as the primary meta-heuristic do not do perform well for larger test cases like LIN318. This could be due to the fact that these schemes are limited by pure AC which does not perform well for larger TSP problems [Stuetzle and Dorigo, 1999].

Table 6.1: Comparison of hybrid schemes improvement with respect to ACO

No	Instance Name	EA	IE	ED	CC	CCED	IEEA
1	att48	-0.1198	2.0190	2.0190	1.7885	2.0190	2.0190
2	eil51	0.0000	0.6977	0.9302	0.9302	0.9302	0.9302
3	eil76	0.0000	2.0036	2.0036	2.0036	2.0036	2.0036
4	pr76	-6.2557	3.4243	3.4243	3.4243	3.4243	3.4243
5	kroA100	-3.2794	1.2848	1.2848	1.2848	1.2385	1.2848
6	kroB100	-0.2549	4.0959	4.0397	4.0830	3.7762	4.3379
7	kroC100	-1.7765	3.2545	3.2545	3.2545	3.2545	3.2545
8	kroD100	0.8679	6.1763	6.1763	4.9840	6.1763	6.6585
9	kroE100	-2.4864	2.0355	2.7929	3.0949	2.9898	3.2306
10	eil101	0.9245	3.0817	3.0817	3.0817	3.0817	3.0817
11	chl30	2.2643	5.6685	5.6069	5.2680	5.8380	5.8380
12	kroA150	-6.8817	4.0893	3.3054	3.4499	3.3235	4.1796
13	kroB150	-1.8274	5.7687	6.2955	3.7586	5.4391	6.3743
14	d198	1.7244	9.2947	8.7429	9.2027	9.1855	9.2890
15	kroA200	-1.0268	13.3629	12.9639	13.4949	13.1546	13.5184
16	kroB200	0.0000	19.3807	18.5479	18.5479	18.5479	20.1109
17	a280	0.0000	17.7067	15.8061	15.4577	15.9328	18.3085
18	lin318	-2.4714	17.9826	17.6739	17.3192	17.3959	18.1973
19	pcb442	-0.0081	17.4962	16.3265	16.0603	16.1732	16.5798
20	att532	0.0000	19.7122	19.2026	19.4504	19.3575	19.6531
21	rat783	3.9117	20.2625	19.5584	19.5584	19.5584	20.6276
Expected Improvement		-0.7950	8.5142	8.2399	8.0713	8.2286	8.7096
Cost (SLOC)		14	47	44	35	79	61
Expected Gain		-0.0568	0.1812	0.1873	0.2306	0.1042	0.1428

Table 6.2: Comparison of hybrid schemes improvement with respect to TS

No	Instance Name	EA	IE	ED	CC	CCED	IEEA
1	att48	-0.9763	1.1808	1.1808	0.9484	1.1808	1.1808
2	eil51	-0.7026	0.0000	0.2342	0.2342	0.2342	0.2342
3	eil76	-1.8553	0.1855	0.1855	0.1855	0.1855	0.1855
4	pr76	-8.9883	0.9406	0.9406	0.9406	0.9406	0.9406
5	kroA100	-4.5548	0.0657	0.0657	0.0657	0.0188	0.0657
6	kroB100	-4.3580	0.1709	0.1124	0.1574	-0.1619	0.4228
7	kroC100	-3.6615	1.4627	1.4627	1.4627	1.4627	1.4627
8	kroD100	-4.3175	1.2685	1.2685	0.0138	1.2685	1.7759
9	kroE100	-4.8784	-0.2509	0.5241	0.8332	0.7257	0.9721
10	eil101	-2.2258	0.0000	0.0000	0.0000	0.0000	0.0000
11	chl30	-2.4048	1.1620	1.0975	0.7424	1.3396	1.3396
12	kroA150	-9.0765	2.1198	1.3198	1.4673	1.3382	2.2120
13	kroB150	-8.5606	-0.4622	0.0993	-2.6052	-0.8137	0.1834
14	d198	-7.4675	0.8109	0.2074	0.7103	0.6914	0.8046
15	kroA200	-16.7871	-0.1526	-0.6138	0.0000	-0.3934	0.0271
16	kroB200	-22.7715	1.0225	0.0000	0.0000	0.0000	1.9189
17	a280	-18.2840	2.6602	0.4121	0.0000	0.5620	3.3720
18	lin318	-23.9362	0.8024	0.4290	0.0000	0.0928	1.0621
19	pcb442	-19.1427	1.7107	0.3172	0.0000	0.1346	0.6189
20	att532	-24.1471	0.3251	-0.3076	0.0000	-0.1154	0.2517
21	rat783	-19.4510	0.8753	0.0000	0.0000	0.0000	1.3292
Expected Improvement		-9.9308	0.7570	0.4255	0.2455	0.4139	0.9695
Cost (SLOC)		14	47	44	35	79	61
Expected Gain		-0.7093	0.0161	0.0097	0.0070	0.0052	0.0159

One point which we wish to demonstrate is the difficultly of predicting the performance of hybrid schemes with any amount of accuracy without performing empirical testing. This point

is borne out by the example of HASTS-IEEA which gives good performance despite the poor performance of its HASTS-EA parent. This is in contrast to HASTS-CCED which is worse than HASTS-ED. This task of testing multiple hybrid approaches is however prohibitively time consuming if each hybrid needs to be developed from scratch. This is where the key benefit of MDF is evident, for it allows developers to rapidly prototype and test the effectiveness of different hybrid approaches. While this admittedly comes at the cost of run-time efficiency, the cost of CPU time is significantly cheaper (and decreasing at a faster rate) than the cost of the developer's time, making the use of MDF a cost effective proposition.

7. CONCLUSION

MDF provides a platform for the rapid prototyping of hybrid meta-heuristic schemes. The well-structured architecture of MDF offers a strong basis for software reuse (Section 6.1), in which each interface clearly defines its role without losing the generic aspect of the framework (Sections 2 and 4). The savings in development time through recycling of codes can be channeled into development and testing of more hybrid schemes, which allows more schemes to be compared so as to choose the most appropriate one. This is especially relevant when considering that the effectiveness of a hybrid can usually only be determined empirically (Section 6.2).

MDF has been successfully used in several applications [Lau et al, 2004b; Lau et al, 2005]. The current design of MDF relies on the users in matching an event to an appropriate handler (with embedded actions). This procedure while good in giving absolute control, it demands certain experiences and expertise from the users. To make MDF user-friendly, we are currently developing a more intelligent control mechanism which makes use of adaptive learning to match events to handlers. The improved controller will monitor the past performance of a handler in dealing with a given event and then favors the execution of handlers which have shown good prior performance.

REFERENCES:

Andreatta, A. A., Carvalho, SER., Ribeiro, C. C., 2002. In Optimization Software Class Libraries, *K*luwer *pp. 59-80.*

Bent, R., van Hentenryck., P., 2004. A Two-Stage Hybrid Local Search for the Vehicle Routing Problem with Time Windows. *Transportation Science,* 38(4): 515-530.

Burke, E., Cowling, P., Landa, Silva J.D. 2001. *Hybrid Population-Based Metaheuristic Approaches for the Space Allocation Problem. Proceedings of the 2001 Congress on Evolutionary Computation (CEC 2001), IEEE Press, pp. 232-239.*

Burke, E., Hart, E., Kendall, G., Newall, J., Ross P., Schulenburg S, 2003. *Hyper-Heuristics: An Emerging Direction in Modern Search Technology*. In Glover F. (ed), Handbook of Meta-Heuristics, pp 457 – 474, Kluwer.

de Champeaux, D., Lea, D, Faure, P., 1993. Object-Oriented System Development, Addison Wesley.

Di Gaspero, L., Schaerf, A., 2003. EASYLOCAL++: An Object-Oriented Framework for the Flexible Design of Local Search Algorithms. *Software - Practice & Experience* 33(8):733-765.

Dorne, R., Voudouris, C., 2004. HSF: The iOpt's Framework to Easily Design Metaheuristic Methods. In Metaheuristics: Computer Decision Making, Kluwer Academic Publishers, pp 237-256.

Fink, A., Voß, S., 2002. HotFrame: A Heuristic Optimization Framework. In: Voß, S., Woodruff, D.L. (Eds.), Optimization Software Class Libraries, pp. 81-154, Kluwer Acad Publishers.

Glover, F., Laguna, M., 1997. Tabu Search. Readings. Kluwer Acad Publishers.

Galorath Incorporated, 2006. SEER-SEM Homepage. http://www.galorath.com/tools_sem.html.

Johnson, D.S., McGeoch, L.A., 1997. The Traveling Salesman Problem: A Case Study in Local Optimization. In Local Search in Combinatorial Optimization, John Wiley and Sons, Ltd, pp 215-310.

Krasnogor, N., Smith, J.E., 2005. A Tutorial for Competent Memetic Algorithms: Model, Taxonomy and Design Issues. *IEEE Transactions on Evolutionary Computation* 9:474-488.

Lau, H. C., Wan, W. C., Jia, X., 2003a. A Generic Object-Oriented Tabu Search Framework. In the Proceedings of the 5th Metaheuristics International Conference. In T. Ibaraki, K. Nonobe and M. Yagiura, (Eds.) *Metaheuristics: Progress as Real Problem Solvers, Springer*.

Lau, H. C., Lim, M.K. Wan, W. C., Wang, H., Wu, X., 2003b. Solving Multi-Objective Multi-Constrained Optimization Problems using Hybrid Ants System and Tabu Search. In the Proceedings of the 5th Metaheuristics International Conference.

Lau, H. C., Lim, M.K., Wan, W. C., Halim, S., 2004a. A Development Framework for Rapid Meta-heuristics Hybridization. In the Proceedings of the 28th Annual International Computer Software and Applications Conference. pp 362-367.

Lau, H.C., Ng, K.M., Wu, X., 2004b. Transport Logistics Planning with Service-Level Constraints. In Proceedings of 19th National Conference on Artificial Intelligence. pp 519-524.

Lau, H.C., Wan, W.C., Halim, S., 2005. Tuning Tabu Search Strategies via Visual Diagnosis. In Proceedings of the 6th Metaheuristics International Conference. pp 630-636.

Park, R.E., 1992. Software Size Measurement: A Framework for Counting Source Statements. Technical Report CMU/SEI-92-TR-020. Carnegie Mellon University-Software Engineering Institute.

Reinelt, G., 1991. TSPLIB - A Traveling Salesman Problem Library. *ORSA Journal on Computing* 3:376-384.

Resende, M.G.C., Riberiro, C.C., 2003. GRASP and Path-Relinking: Recent Advances and Applications. Proceedings of the 5th Metaheuristics International Conference. In T. Ibaraki, K. Nonobe and M. Yagiura, (Eds.) *Metaheuristics: Progress as Real Problem Solvers, Springer*.

Stuetzle, T., Dorigo, M., 1999. ACO Algorithms for the Traveling Salesman Problem. In Evolutionary Algorithms in Engineering and Computer Science, Wiley. pp 163-183.

Talbi, E., 2002. A Taxonomy of Hybrid Metaheuristics. *Journal of Heuristics* 8:541-564.

USC-CSE Center for Software Engineering, 2006. COCOMO II Homepage. http://sunset.usc.edu/research/COCOMOII/.

Vasquex, M., Vimont, Y., 2005. Improved Results on the 0-1 Multidimensional Knapsack Problem. *European Journal of Operational Research* 165: 70-81.

Wagner, S., Affenzeller, M., 2004. The HeuristicLab Optimization Environment. *Technical Report*, Institute of Formal Models and Verification, Johannes Kepler University Linz, Austria.

Wheeler, W.A., 2006. SLOCCount Homepage. http://www.dwheeler.com/sloccount/.

Wolpert, D.H., Macready, W.G.., 1997. No Free Lunch Theorems for Optimization. *IEEE Transactions on Evolutionary Computation* 1(1): 67-82.