TEACHER - AN AUTOMATED SYSTEM FOR LEARNING KNOWLEDGE-LEAN HEURISTICS

Arthur Ieumwananonthachai and Benjamin W. Wah
Center for Reliable and High-performance Computing
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1308 West Main Street
Urbana, Illinois 61801
{arthuri,wah}@manip.crhc.uiuc.edu

ABSTRACT

In this report, we present TEACHER, a system for learning knowledge-lean heuristics under resource constraints. TEACHER is an implementation of a genetics-based learning framework we have developed for improving the performance of heuristics in application problem solvers. Besides providing a flexible and modular framework for conducting experiments, TEACHER provides (a) a test-bed for experimenting with various resource scheduling, generalization, and heuristics-generation strategies, (b) an automated learning system that can be easily interfaced to new applications and can be customized based on user requirements and target environments. This report describes the application-independent (AppI) functions provided by TEACHER, and the application-dependent (AppD) functions for interfacing to new problem solvers. By adjusting various global parameters in TEACHER, users can control the numerous options and alternatives in TEACHER. To illustrate our design, we use CRIS, a genetics-based test-pattern generation system, as a running example throughout the report.

KEYWORDS:

Generalization, genetic programming, knowledge-lean heuristics, machine learning, prototype learning system, resource constraints, VLSI circuit testing.

Research partially supported by National Aeronautics and Space Administration under contract NCC 2-481, NAG 1-613, and NASA Graduate Fellowship NGT-50743 and by National Science Foundation under grants MIP 88-10584 and MIP 92-18715.

Technical Report CRHC-95-08.

1. INTRODUCTION

Problem solvers using ‘‘heuristics’’ are applied in many application domains when optimal solutions cannot be found within a reasonable amount of time. By definition, *heuristics* are common sense knowledge that can be used to solve a problem without any guarantee on the resulting performance. Consequently, performance of a problem solver can be affected by the choice of heuristics in the problem solver.

At present, most heuristics are designed manually based on past experience of their designers. Since the number of possible heuristics is very large for realistic applications of reasonable complexity, heuristics designed manually may not work well when applied in new problem instances. Further, there is no systematic method to evaluate the effectiveness of heuristics designed manually. For these reasons, an automated method for discovering the proper heuristics for a particular application is very desirable. This leads to the development of our system for automated learning of heuristics [10].

In most previous work in heuristics learning for a target application, domain knowledge about the application is required [7]. However, in many real-world application, such domain knowledge is not available or is extremely difficult to extract. In contrast, our approach for automated heuristics learning requires little domain knowledge and can be applied to a wide variety of applications that would not be possible under other circumstances. To operate in knowledge-lean domains, our system can only learn performance-related heuristics; *i.e.*, only the performance of the problem solver is affected by the choice of different heuristics.

This report describes the TEACHER learning system, an implementation of our learning strategy. This system can learn high-performance heuristics for its target application within given resource constraints, and can determine the scope of generality of the learned heuristics. There are three main objectives in the development of this system: (1) to be a test-bed for conducting experiments on the functionalities of various components of our learning strategy, (2) to be able to learn better heuristics for a wide variety of applications and problem solvers, and (3) to be able to learn customized heuristics for specific environments.

To accomplish these objectives, we have implemented many different strategies and options for components that are independent of target applications. In addition, new problem solvers can be interfaced to TEACHER easily by implementing a set of application-dependent (AppD) functions. Since little domain knowledge is required by our strategy, these functions are easy to implement. TEACHER allows new heuristics to be learned and customized for a specific environment and user requirements. All functionalities within TEACHER can be controlled and accessed through a set of global parameters adjustable by users.

In this report, we provide the details about TEACHER in order to allow readers to use this system for any of the objectives described above. As an example on how these objectives can be accomplished, we use CRIS, a genetics-based test-pattern generation system for circuit testing, as a running example.

This report is organized as follows. Section 2 briefly describes the learning strategy we have developed and the architecture of TEACHER. Section 3 describes the application-independent functionalities provided by TEACHER. Section 4 describes the application-dependent functionalities of TEACHER that must be supplied through a set of interface functions. Section 5 provides an example of how TEACHER can be applied to a new application (CRIS in this case). Section 6 describes the procedure for conducting a learning experiment using TEACHER. Section 7 describes the options in conducting experiments available through the global control parameters that can be adjusted by end users. Finally, Section 8 draws a conclusion.

This manual also includes 4 Appendices. Appendix A describes the application-independent (AppI) data structures and support functions provided by TEACHER. Appendix B lists the AppI control parameters that control the behavior of each learning experiment conducted using TEACHER. Appendix C lists the application-dependent (AppD) functions required by TEACHER from each target application, and shows a default definition for most functions that will work in most target application domains. Appendix D shows two examples of the implementation of AppD functions for CRIS: one without constraints on CPU time usages and one with constraints.

2. TEACHER: LEARNING STRATEGY AND ARCHITECTURE

In this section, we present a brief summary of the strategy we have developed for learning performance-related heuristics for knowledge-lean application domain [10]. We then present an overview of the implementation of TEACHER, our tool for learning new heuristics for real-world applications. In this paper, we use the term *heuristic method* (HM) to denote a collection of *heuristic decision elements (HDE)* or *heuristics decision rules* applied to solve a target problem [9].

2.1. Overview of Strategy Learning

In our research, we deal with application domains that have nondeterministic performance due to either (a) a large number of test cases, or (b) a small number of test cases but nondeterministic behavior in the problem solver. Consequently, a major issue in learning problem solving heuristics is the generality and performance of the resulting HM. There is a tradeoff between the performance level and the degree of generality that can be achieved by the set of HMs selected. With a large space of target problems, it may be impossible to find a single HM that performs well across the entire problem space. On the other hand, finding a good HM for each target problem is impractical due to the amount of efforts and memory space required. Ideally, we want to discover a small set of HMs that can cover the entire problem space with high performance level in each region.

To address this tradeoff issue, we require users to determine the minimum level of generality by partitioning the problem space into subdomains. A *problem subdomain* is a maximal partitioning of test cases in a subspace so that different HMs can be compared quantitatively based on their aggregate performance. Good HMs that perform consistently can then be learned for each subdomain. We also define a *problem subspace* as a user-defined partition of a problem space that is composed of one or more problem subdomains. This partitioning should be implemented in such a way that, based on the available domain knowledge, HMs that perform well over a single subspace can be discovered. This provides a higher level of generality above each problem subdomain. Based on this concept, we have developed a population-based learning and generalization method that finds high performance HMs for each subdomain, and determines the scope of generality of the learned HMs [10].

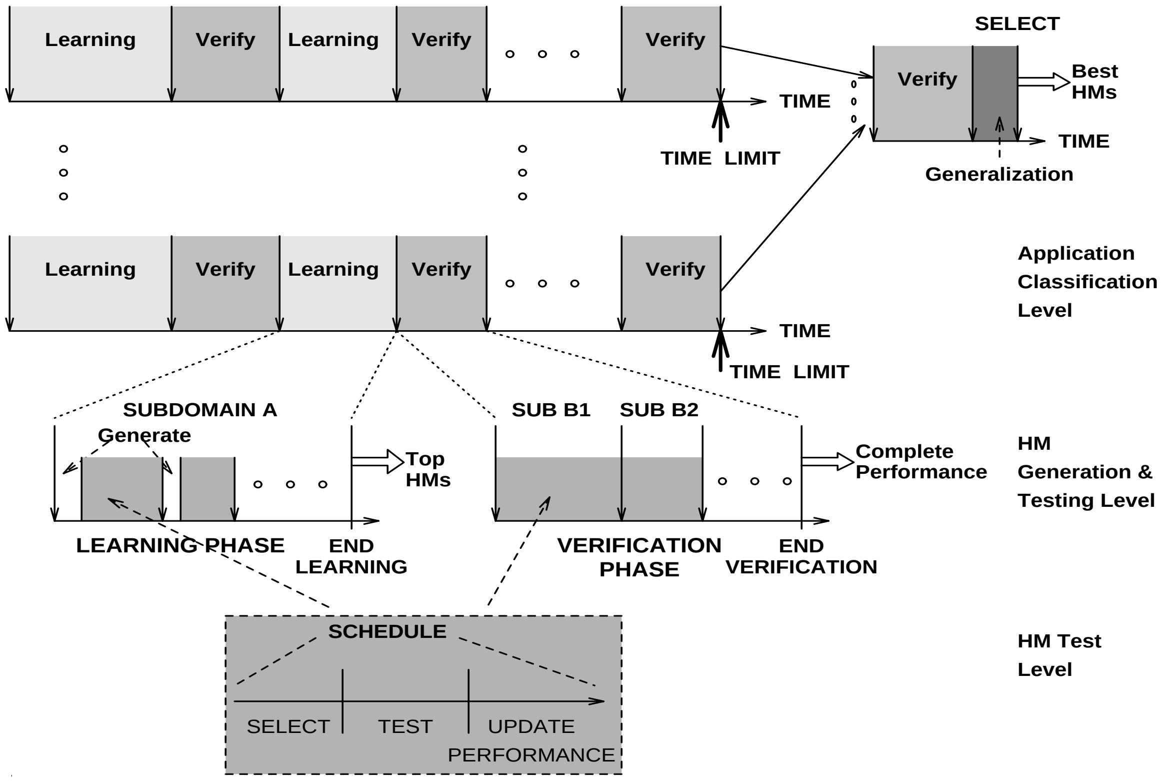

Figure 2.1. Overall Learning Process

The actions of our heuristic-learning strategy can be divided into three phases: (i) *learning phase*, (ii) *verification phase*, and (iii) *generalization phase*. Each learning experiment contains one or more segments of each of the first two phases and one generalization phase at the end of each experiment (see Figure 2.1). The objective of each learning phase is to find the best possible HMs for a target subdomain using only the given amount of resources. The objective of the verification phase is to obtain complete performance information for a selected set of HMs in each subdomain. The objective of the final generalization phase is to select the best set of HMs of a given size that can provide the best combined performance over all given subdomains.

Within each phase, a pool of multiple competing HMs are maintained. Each learning phase uses a generate-and-test approach based on genetic algorithms [1] to generate better HMs, and to determine their performance. Due to the limited resources, HMs cannot be tested as adequately as desired. Hence, a key component of the learning phase is the resource scheduler that determines the actions to be performed. At the end of each learning phase, a set of HMs with the best performance are extracted for further evaluation and, eventually, generalization. More details on resource-scheduling, verification, and generalization strategies are described in Section 3 and in reference [10].

2.2. Architecture of TEACHER

TEACHER is an implementation of our learning system specified in the previous section. One of our main goals in implementing this system is to simplify the task of applying TEACHER to new problem solvers and new application domains. To ensure this, we separate the functionalities of our learning system into functions that require domain knowledge, and functions that can be implemented in a general fashion independent of application domains. Application-independent (AppI) components are the core of our learning strategy and provides generalization and resource scheduling services. Application-dependent (AppD) components include the problem-solver interface, generator of new HMs, test-case manager, and evaluator of HM performance. Our system provides an implementation of AppI functions and defines an interface to AppD ones. To apply TEACHER to learn HMs for a new target application, the AppD functions must be implemented based on the specification of this interface. The overall architecture of TEACHER is shown in Figure 2.2.

The current version of TEACHER is implemented using Common-LISP (Allegro Common-LISP version 4.1 for Sun SparcStation running SunOS 4.1.3) to allow easy manipulation of HMs and partitioning of functionalities. This manual assumes that readers are familiar with this language. The system is built in a modular fashion to simplify future extensions and improvements of AppI components and allows the implementation of the AppD functionalities for new applications to be independent from the implementation of the AppI functionalities.

3. APPLICATION-INDEPENDENT (AppI) COMPONENTS

This section describes the AppI functionalities that have been implemented in TEACHER. The main AppI components of our learning strategy are the main control loop, the resource scheduling service, the verification service, and the generalization

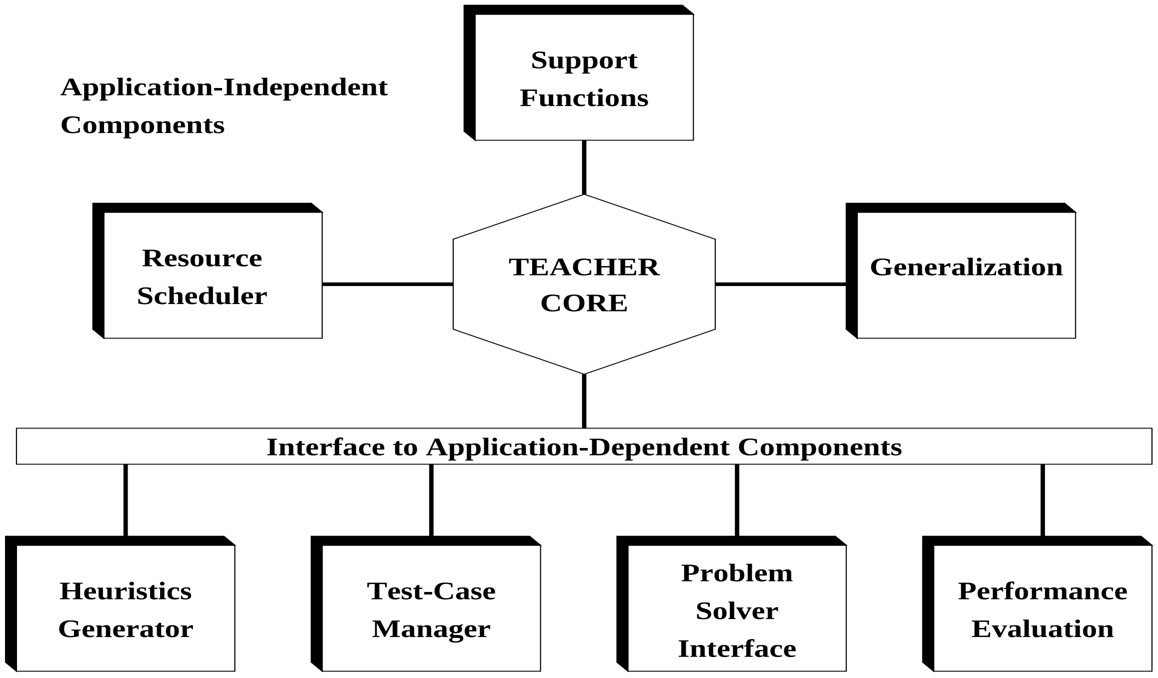

Figure 2.2. TEACHER System Architecture

---

service. In addition, our system also implements a set of commonly-used operations that allow simple and useful operations to be performed within both the AppI and AppD components with minimum duplications in coding.

In this section, we first describe our implementation of the core AppI components. These components are implemented with many options and alternatives to create a flexible and general learning system that will work with a wide range of applications and environments. These options are controllable through the setting of global control parameters described in Section 7. Next, we summarize the support functions we have implemented in TEACHER. Finally, we describe the definition of AppI data structures that are passed through the AppI-AppD interface. We also describe global variables that can be accessed from the AppD code. This information is necessary for implementing the AppD components for a new target application.

3.1. Main Control Loop

The main control loop is the main program of TEACHER that defines the behavior of the learning system . Learning is performed by controlling the available resources and the pool of active HMs. All AppI and AppD actions within each learning experiment are activated from this main loop. All AppD actions are called through a pre-defined interface described in Section 4.

The main control loop is represented in the Figure 3.1 as simplified pseudo-code.

```
Initialize() ;

retain-set = ;   /* empty set*/
For subdomain in LEARNING-SET                     /* Learn(subdomain) */
  pool = Initialize-heur(retain-set)
  While Resources-left do
    action = Resource-schedule(resources, pool) ;
    If action == "Generate"
    Then
      new-set = Generate-new-heuristics() ;
      pool = Remove-heuristics(pool) ;
      pool = Union(pool, new-set) ;
    Else
      heur = Select-heuristic(pool) ;
      test-case = Select-test-case(heur) ;
      result = Apply-problem-solver(heur, test-case) ;
      Update-performance(heur, test-case, result) ;
    EndIf
  EndWhile
                                                 /* Verify(subdomain) */
  top-set = Select-top-heuristics(pool, NUM-RETAIN) ;
  Full-evaluation(top-set, subdomain) ;
  retain-set = Union(retain-set, top-set) ;
EndFor

/* Final Verification Phase + Generalization Phase */
Full-evaluation(retain-set, Union(LEARNING-SET, VERIFY-SET) ) ;
Generalize-subdomain(retain-set, Union(LEARNING-SET, VERIFY-SET) ) ;
selectHMs = Select-top-heuristic-for-each-partition(retain-set) ;
```

Figure 3.1. TEACHER Main Control Loop

At present, timing of a learning experiment can be measured in logical time or in physical time. In logical time, each evaluation costs exactly one unit of time, whereas in

Table 3.1. Global parameters controlling resources of experiments

Parameter	Description	Range	Default
`*CPU-LIMIT*`	Number of CPUs in parallel learning	Integer ≥ 1	1
`*TIME-LIMIT*`	Number of time units per subdomain	Integer ≥ 1	1
`*ASYNCH-RUN-F*`	Simulation using logical or physical time	Boolean	`NIL`
`*NUM-LEARN-PHASES*`	Number of subdomains to learn	Integer ≥ 1	1

physical time, the actual CPU time is measured. Currently, it is up to AppD function `Evaluate-test-case` to update the actual amount of computational resources (CPU time) consumed in each evaluation. The mode of the current learning experiment is selected through global parameter `*ASYNCH-RUN-F*`.

Three global parameters control the amount of resources available to each learning experiment. The amount of time available in each learning phase for learning HMs for a given subdomain is specified by `*TIME-LIMIT*`. This is either in quantum for logical-time measurements or in CPU seconds for physical-time measurements. The system allows simulation of parallel learning through global parameter `*CPU-LIMIT*`. Finally, `*NUM-LEARN-PHASES*` specifies the number of learning phases, *i.e.*, the number of subdomains to be learned within a learning experiment. Any additional subdomains supplied by users will be tested only in the final verification phase.

Table 3.1 summarizes the global parameters for controlling resources available in TEACHER.

3.2. Resource Scheduling Component

The actions in a learning phase can be viewed as a sequence of generates-and-tests as shown in Figure 3.2. The interval between generating new HMs is called a *generation*. At the start of each generation, new HMs are generated and added to the active pool while old HMs are removed. Within each generation, the HMs within the active pool are evaluated through a sequence of tests.

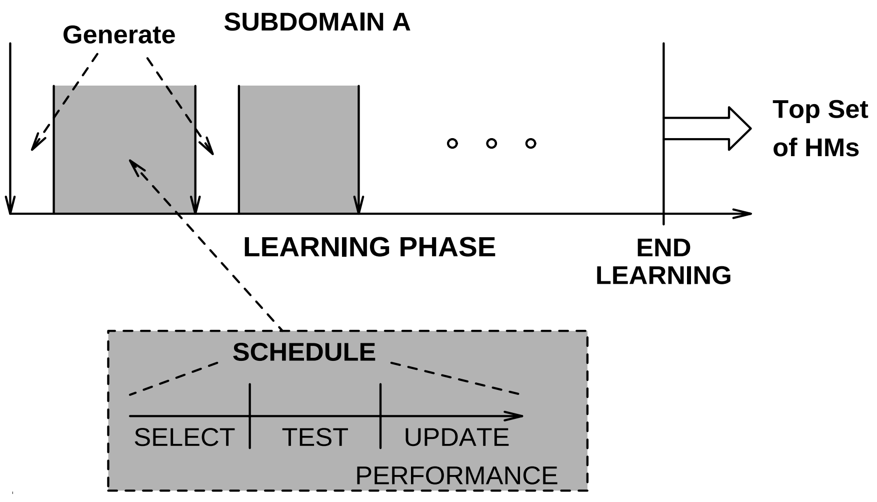

Figure 3.2. Resource Scheduling in a Learning Phase

The sequencing and timing of actions performed in each learning phase is controlled by the resource scheduler (resource scheduling component). The objective of the resource scheduler is to find the best possible HMs at the end of the learning phase. The decisions to be made by the resource scheduler can be divided into two types. First, within each generation, the resource scheduler must select the HMs to be tested from the active pool. This is known as the *sample-allocation problem*. Second, the resource scheduler must decide when to end the current generation. This is known as the *duration-scheduling problem*. When only a limited amount of resources is available, the choice of the resource scheduling strategy can dramatically affect the performance of the HM learned.

The decision making process of some resource scheduling strategies requires information about the performance distribution (usually variance) of each HM. Normally, this information cannot be obtained unless a minimum number of evaluation (usually 4) is performed on each HM. We have developed two methods to estimate the performance distribution in order to reduce the minimum number of test to be performed on each HM to one. These methods require some assumptions which may not be applicable to all applications (see Section 3.2.4). Users must make sure that these assumptions are met before applying these distribution estimation methods.

In the reset of this subsection, we first discuss the behavior of the learning system at the beginning of each generation and the parameters that can be controlled. Next, we discuss the resource scheduling strategies that have been implemented in TEACHER to solve the sample-allocation and duration-scheduling problems. We then summarize the two distribution-estimation methods we have implemented. Table 3.2 lists the global parameters that control the resource scheduling component.

3.2.1. Parameters for Controlling Behavior of Each Generation

At the end of each generation, a fraction of the HMs in the active pool are retained while the remaining HMs are replaced by new ones. The new HMs are generated by the AppD heuristics generator based on performance information obtained on the old HMs. In the special case in the first generation, the pool of HMs are provided either from an initial user-defined set (in the first learning phase) or from the set of the best HMs from previous learning phases. Additional new HMs are generated if there are not enough HMs in the first generation.

There are several resource scheduling parameters that control the behavior of this action. First, the total number of HMs that are maintained within the active pool are set based on global parameter `*NUM-CANDIDATES*`. The fraction of HMs that are retained within the active pool, *i.e.*, not removed, is set by global parameter `*PERCENT-RETAIN*`. The selection of the retained HMs can be either random or based on their performance (where the best ones are retained). This choice is controlled by global parameter `*RETAIN-STRATEGY*` which can be set to `'BEST` or `'RANDOM`.

3.2.2. Sample-Allocation Strategies

There are four different sample-allocation strategies in our current implementation: round-robin, greedy, minimum-risk [5], and non-parametric minimum-risk [2, 10]. A round-robin strategy is the simplest static strategy that does not use dynamic performance information to guide the selection of HMs. It is used in most existing genetics-based machine-learning systems. A greedy strategy is a simple dynamic sample-allocation strategy that always selects the HM with the best performance for further testing. Minimum-risk and non-parametric minimum-risk strategies are two dynamic sample-

Table 3.2. Global parameters controlling resource scheduling strategy

Parameter	Description	Range	Default
`*NUM-CANDIDATES*`	Number of active HMs	Integer ≥ 1	30
`*PERCENT-RETAIN*`	% of HMs retained between generations	Float 0.0-0.5	0.33
`*RETAIN-STRATEGY*`	Selection method to retain HMs	Name	`'BEST`
`*STRATEGY-TYPE*`	Sample-allocation strategy	Name	`'Min-Risk-1`
`*SCHED-GOAL*`	Optimization goal	Name	`'SELECTION`
`*ALLOC-MIN-CT*`	Minimum tests required per HMs	Integer ≥ 1	4
`*GENERATION-LIMIT*`	# generations per learning phase	Integer ≥ 1	10
`*DMDS-SCHED-F*`	Select DMDS strategy	Boolean	`NIL`
`*DIST-TYPE*`	Distribution of HMs performance	Name	`'UNKNOWN`
`*DYNAMIC-UPDATE-F*`	Select dynamic-update estimation	Boolean	`NIL`
`*UPDATE-THRESH*`	# new HMs between estimation update	Integer ≥ 1	20
`*PRE-SAMPLE-COUNT*`	# quanta to perform for each HM	Integer ≥ 1	4
`*PRE-SAMPLE-FRAC*`	Min. fraction of time for pre-sampling	Float 0.0-0.5	0.1
`*PRE-SAMPLE-MIN-CAND*`	Min. # of HMs to be pre-sampled	Integer ≥ 4	4
`*TEST-SCHED-F*`	Select ignoring pre-sampling time	Boolean	`NIL`

allocation strategies we have developed. They use past performance information to guide the selection of HMs for future testing. The minimum-risk strategy requires performance of each HM to be normally distributed, whereas the non-parametric minimum-risk strategy makes no assumption about performance distribution.

For both minimum-risk and non-parametric minimum-risk strategies, the performance variance of each HM must be reasonably approximated before they can be applied. To obtain this information, a minimum number of tests must be performed on each HM. The global parameters `*ALLOC-MIN-CT*` controls the minimum number of tests to be performed before one of these two strategies can be used. The default value for this parameter is 4. When the minimum number of tests is below this threshold, a round-robin strategy is used instead. In Section 3.2.4, we present some methods that reduce the minimum number of tests required to 1.

The sample-allocation strategy is selected based on the values of global parameters `*STRATEGY-TYPE*` and `*SCHED-GOAL*`. `*STRATEGY-TYPE*` can be either `'Round-robin`, `'Greedy`, or `'Min-risk`. The selection between the minimum-risk and the non-parametric minimum-risk strategies is controlled by `*SCHED-GOAL*` which can be either `'TOP-ONE` for minimum-risk, or `'SELECTION` for non-parametric minimum-risk.

3.2.3. Duration-Scheduling Strategies

We have implemented two duration-scheduling strategies: fixed-duration and dynamic multi-objective duration scheduling (DMDS). The flag `*DMDS-SCHED-F*` indicates when DMDS will be used. In fixed-duration scheduling, equal and pre-determined amount of time is allocated to each generation. This duration is controlled by the amount of time allocated to each learning phase (`*TIME-LIMIT*`) and the number of generations set by `*GENERATION-LIMIT*`.

For DMDS, the generation duration is determined by dynamic performance information within the learning system. In this case, `*GENERATION-LIMIT*` must be set to 1. DMDS strategy is created to deal specifically with multi-objective problems where constraints must be imposed on some performance measures. At present only one constraint can be set in TEACHER using global parameter `*GLOBAL-CONSTRAINT*`. DMDS tries to find feasible HMs under relaxed constraints before attempting to address constraints that are more difficult to achieve. This strategy uses global variable `*RT-CONSTRAINT*` to store the current constraint level as well as parameters of DMDS control. Details of DMDS strategy can be found in reference [10]. DMDS also requires additional AppD supports to manipulate constraints. These functions are discussed in Sections 4.5.1 and 5.3.

3.2.4. Distribution Estimation

There are two methods available in TEACHER for estimating the performance distributions of HMs in the active pool: pre-sampling and dynamic update. Both methods assume that the variances of each HM's performance are identical. In pre-sampling, tests are performed on a subset of untested HMs within the active pool at the beginning of each generation. Each HM selected is tested a minimum amount of times that is controlled by global parameter `*PRE-SAMPLE-COUNT*` (usually 4). This testing is performed until a minimum number of HMs (specified by `*PRE-SAMPLE-MIN-CAND*`) are tested and the amount of time allocated for pre-sampling is exhausted. The fraction of time in each generation to be used for pre-sampling is specified by global parameter `*PRE-SAMPLE-FRAC*`. After pre-sampling is finished, the performance information of the tested HMs is used to estimate the mean and variance of the mean performance of HMs in the pool, and the (common) variance of each HM's performance. Unfortunately, the overhead for pre-sampling is quite significant.

A dynamic-update method is created to reduce the amount of overhead in estimating distributions. With this method, pre-sampling is performed only in the first generation. In subsequent generations, only performance data from the first test-case solved by each new HM is required. This information is used for updating the estimated distribution. The main shortcoming of this method is that the common variance of each HM's performance cannot be updated. The global parameter `*UPDATE-THRESH*` specifies the minimum number of new HMs' data that must be collected before estimating and updating the distribution parameters.

Currently, our distribution-estimation methods can be enabled only when either the minimum-risk or the non-parametric minimum-risk sample-allocation strategy is used. Dynamic update is performed when the global parameter `*DYNAMIC-UPDATE-F*` is set to `T`. Pre-sampling is performed only when this is not true and the global parameter `*DIST-TYPE*` is set to `'NORMAL`. This means that the performance of each HM is normally distributed with common variance, and that the mean value of each HM is also normally distributed.

Since the overhead for pre-sampling during the first generation is still high, the time spent in pre-sampling in the first generation can be ignored by setting global parameter `*TEST-SCHED-F*` to `T`.

Table 3.3. Global parameters controlling verification and generalization

Parameter	Description	Range	Default
`*VERIFY-SELECT*`	# of top HMs to verify in each subdomain	Integer ≥ 1	20
`*GEN-HM-DESIRED*`	# of generalized HMs to produce	Integer ≥ 1	1
`*GEN-MAXDEV-DESIRED*`	Desired closeness to top HM	Float 0.0-1.0	1.0

3.3. Verification Component

At the end of each learning phase, a set of HMs that have the best observed performance are selected for further verification. The number of HMs selected is controlled by global parameter `*VERIFY-SELECT*`. The verification component of TEACHER controls the behavior and actions within each verification phase. Table 3.3 summarizes parameters used in the verification and generalization phases.

The goal of verification is to obtain complete performance information of the selected set of HMs by performing full evaluation over the entire test-case database. The best set of HMs for a target application can then be selected more accurately in the generalization phase. This is necessary because the performance of each HM during the learning phase is usually based on incomplete data over a subset of the database of test-cases. This incomplete information is not enough for selecting the best set of HMs during the generalization phase.

Actions within each verification phase is similar to those within the learning phase (see Figure 2.1). The main differences are (1) that the set of HMs are fixed with no pruning or generation in the verification phase, and (2) that test-cases can come from more than one problem subdomains. Our current implementation simply uses facilities developed for the learning phase. The code iterates over each problem subdomain and evaluates each HM completely over each problem subdomain using the round-robin resource scheduling strategy.

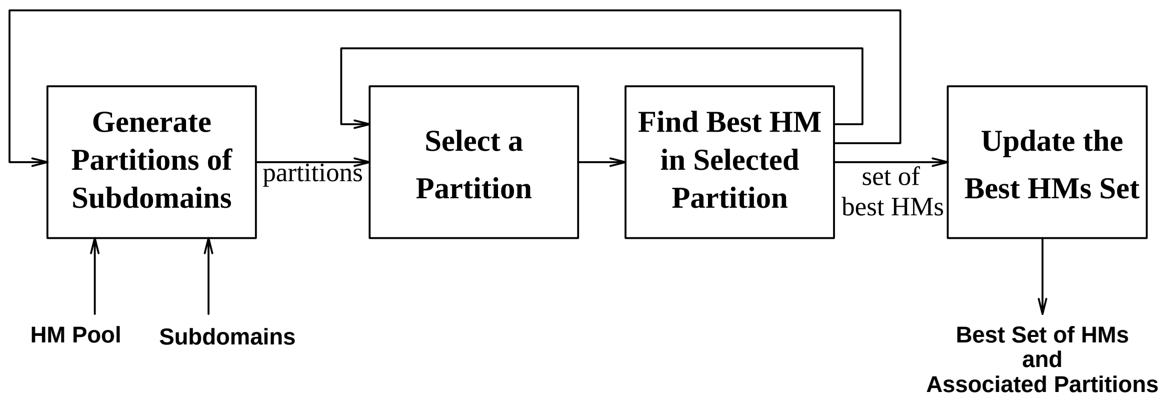
Generate Partitions of Subdomains
partitions
Select a Partition
Find Best HM in Selected Partition
set of best HMs
Update the Best HMs Set
HM Pool
Subdomains
Best Set of HMs and Associated Partitions

Figure 3.3. Subdomain Partitioning and Generalization

3.4. Generalization Component

After one or more learning and verification phases, TEACHER must then find the set of HMs that can best generalize and cover the given subdomains from among those HMs that have been learned and verified. The number of HMs desired is controlled by global parameter `*GEN-HM-DESIRED*`. This section describes our implementation of this generalization and selection process.

The objective here is to find a set of HMs of size `*GEN-HM-DESIRED*` that provide the best combined performance over the given set of subdomains. Each HM within the selected set can be viewed as the best HM over a partition (or portion) of the given subdomains. Instead of enumerating over all possible set of HMs, we enumerate over all possible partitioning of the given set of subdomains into `*GEN-HM-DESIRED*` number of partitions. This is done because the number of HMs is usually much larger than the number of subdomains. Basicly, we find a set of HMs for each set of partitions by selecting the best single HM for each partition (inner loop in Figure 3.3). The best set of HMs is then found by enumerating over all possible partitioning (outer loop in Figure 3.3) and by selecting the set of HMs that give the best combined performance. The overall process of selecting the best partitioning of subdomains is shown in Figure 3.3.

In generalization, we need to formulate a range-independent and distribution-independent performance measure for comparing performance of HMs within each subdomain. Otherwise, performance across multiple subdomains cannot be compared or combined due to differences in range or distribution. We have developed one such performance measure called *probability-of-win* (P_{win}) [10].

We select the best HM for a partition based on the average of probability-of-win values across all subdomains in the partition. HMs picked this way usually wins with a high probability across most of the subdomains but occasionally may not perform well in a few subdomains. We have enhanced this selection method by enforcing that P_{win} of the selected HM must not deviate from the maximum P_{win} of each subdomain within the given partition by the value specified by global parameter `*GEN-MAXDEV-DESIRED*`. This constraint is relaxed if no partitioning can provide a set of HMs that can satisfy this constraint. In that case, the best HM for each partition is selected based on the minimum deviation from the maximum P_{win} of each subdomain. This enhancement not only forces the performance of the selected set of HMs to be closer to the maximum for each subdomain, but also reduces the number of HMs that must be considered in each partition.

3.5. General Support Routines

To simplify the implementation of TEACHER, we have created a set of support functions implementing several general and commonly-used operations. These functions make these basic operations easier to use and reduce the amount of duplicate codes. They can be called from both the AppI and AppD components. The services provided by these functions can be categorized into three main groups as shown below.

(1) *Statistics Functions*. These functions provide computation of several simple statistical measures (mean, variance, etc.), and the cdf and inverse-cdf information for three basic distributions (normal, Student-*t*, and F).

(2) *Randomization Functions*. These functions introduces randomness into learning experiments in a controlled fashion. They allow a random seed to be set and provide random manipulation of elements in a list.

(3) *Data-Manipulation Functions*. These functions provide commonly used list and number manipulations. They include `Max-of`, `Min-of`, `Sort-descending`, `Subset`, `Sqr`, `Vector-length`, `Normalize-vector`, and `Dot-product`.

The definitions of all available functions are shown in Appendix A.

3.6. Global Variables and Data-Structures Definitions

In order for the AppD functions to be written, the definition of each input and output to each function must be known to the programmer. Several AppI data structures are used by variables that are passed in or out of the AppD functions. In this section, we define all the AppI data structures used. There are also some global variables that can be accessed by the AppD functions. We describe these variables as well as their data structures here. The exact definitions of all the data structures discussed here are shown in Appendix A.

The most important data structure is the representation of each HM (or candidate). This is defined as data structure `candidate-S`. There are two other structures defining the performance information of each HM associated with `candidate-S`: `performance-history-S` and `performance-instance-S`. All three data structures are passed between the AppD and AppI components of TEACHER. For some applications, special information about the HM or its performance must be added to the usual information. To accomplish this, a super-set of these data structures must be declared in the AppD components in such a way that the AppI components can still access these data structures without modification. Section 4.1 shows how this can be accomplished.

Two other AppI data structures are passed as inputs to the AppD functions. The first data structure, `candidate-list-S`, represents the set of HMs that exist within the current learning experiment. This data structure is used by global variable `*candidate-pool*` that is passed to the AppD functions. This global variable represents the current state of the learning system in term of HMs. The second data structure, `cpu-status-S`, represents the current status of the processor that will execute the target problem solver. Another data structure, `quantum-status-S`, is used within one field of the `cpu-status-S` data structure (`cpu-status-S-quantum-status`).

There are three global variables that can be accessed by an AppD function: `*candidate-pool*, *learning-constraints*, and *learning-params*.` These global variables should **never** be modified by AppD functions. They should be used only as sources of information. First, `*candidate-pool*` is a global variable we have mentioned earlier. Second, `*learning-constraints*` represents the amount of resources available in each learning phase of the current learning experiment. It is defined using the `resource-constraints-S` data structure. Its value is set based on global parameters `*TIME-LIMIT*` and `*CPU-LIMIT*` (see Sections 3.2 and 7). Third, `*learning-params*` represents the parameters of learning algorithm. Currently, it only stores the definition of a quantum within the `learning-params-S` data structure, and its value is set by AppD function `Reset-quantum-def`.

4. APPLICATION-DEPENDENT (AppD) COMPONENTS

In the previous section, we have described the various functionalities provided by TEACHER that are independent of the target application domain. In this section, we describe the AppD functionalities that must be implemented for each new application domain so that TEACHER can tune its HMs. These functionalities include the implementation of four main components of learning that require application-specific information: test-case manager, problem-solver interface, performance-data manager, and heuristics generator. In addition, TEACHER also requires a set of initialization and clean-up functions to set up application-specific, environment-specific, and user-specific conditions. Finally, there are a set of support functions to report application-dependent information about the state of the learning system to users.

In this section, we discuss the importance and functions of each component mentioned above along with any potential difficulties in implementing them. For a complete listing of the AppD functions required along with their requirements and a default implementation, see Appendix C. But first, we describe a method for customizing the AppI data structures for application-specific requirements without requiring changes in the existing AppI code.

4.1. Data-Structure Definitions

As mention in Section 3.6, the data structures representing each HM and its performance are declared within the AppI components of TEACHER. However, some target applications may require additional information about each HM or its performance that are not available with the AppI definitions of these data structures. In that case, an AppD declaration of these data structures is required. However, this cannot be done arbitrarily without breaking existing AppI codes that are dependent on accessing specific information at specific location within each data structure. Here, we show a method for declaring a new customized data structure that is a superset of the pre-defined data structure and works with the existing AppI code. An example of adding a new application-specific field into several AppI data structures is shown in Figure 4.1.

```
(defstruct ( my-candidate-S (:include candidate-S)
                            (:constructor make-candidate-S)
                            (:conc-name candidate-S-) )

"The data structure for a potential candidate
 - declare domain-dependent fields of candidate structure."

  (new-field      NIL)        ; an example application-specific field

) ;; end of my-candidate-S


(defstruct ( my-performance-history-S
             (:include performance-history-S)
             (:constructor make-performance-history-S)
             (:conc-name perf-hist-S-) )

"The data structure for a performance-history record
 - declare domain-dependent fields of the performance history."

  (new-field      NIL)        ; an example application-specific field

) ;; end of my-performance-history-S


(defstruct ( my-performance-instance-S
             (:include performance-instance-S)
             (:constructor make-performance-instance-S)
             (:conc-name perf-inst-S-) )

"The data structure for a performance record over one test instance
 - declare domain-dependent fields of instance performance."

  (new-field      NIL)        ; an example application-specific field

) ;; end of my-performance-instance-S
```

Figure 4.1. Declaring additional AppD fields for AppI data structures

4.2. Initialization and Clean-up

This section describes the set of functions that put TEACHER into a state necessary for a learning experiment to be conducted for the target application and environment. Most of these functions try to initialize the system into a proper condition based on the requirements of the target application, target environment, and users. These functions are applied to initialize the set of subdomains with their associated test cases (`Reset-test-database`), and the initial pool of HMs (`Reset-candidate-pool`) along with existing information on their performance (`Reset-performance-database`). There is also a function to define the proper definition of quantum for each target application (`Reset-quantum-def`). Function `Initialize-domain` allows any additional AppD initialization of the learning system to be performed.

`Clean-up-domain` is a function for cleaning up any application-specific residue produced during the learning process. Another function, `Load-up-domain`, allows the AppD components to be divided into multiple files. Only a single file, ‘‘domain.cl’’, containing this function must be loaded manually. This function can then be used to load all additional files if necessary.

In most circumstances, these functions do not have to perform any action, except for `Reset-test-database` which must perform initialization based on both application- and environment-specific information. We defer discussion about `Reset-test-database` until the next section in the context of test-case management.

4.3. Test-Case Management

The main issue in the management of test cases is in deciding what is a test case and what constitutes a subdomain for the target application. In addition, the necessary information for the problem solver to identify the target test case must be determined. These decisions must be made separately for each application. Next, the test-case manager must decide on the sequence of test cases to be applied during a learning experiment. Function `Reset-test-database` allows these decisions to be made in an application-dependent fashion. This function returns the set of subdomains to be used within the current learning experiment and the specification of test cases (in a proper testing sequence) for each subdomain. In a flexible implementation, this function should allows users to customize the return value for each experiment. The result of this function should be in the form of:

```
( ( subdomain-name ( -- list of test cases -- ) )
  ( subdomain-name ( -- list of test cases -- ) )
     ...
)
```

Function `Get-test-case` uses the specification of test cases for the current subdomain as defined by `Reset-test-database` to select the next test case for testing the current HM.

Some applications may require certain actions to be performed before test cases from a new subdomain can be used. TEACHER provides a solution to this problem by calling `Initialize-test-subdomain` function before using a test case from a new subdomain. In most instances, this function is not needed.

4.4. Problem-Solver Interface

This component is application specific. To implement this component, the programmer must know the method for activating the target problem solver, the method for passing a specific HM and a specific test-case to the problem solver, and the method for accessing the results of the problem solving process. With this information, the problem solver can be used to solve a specific test-case using a specific HM, and the result can be recorded within TEACHER. There are several methods for passing a HM and a test-case to the problem solver; these include passing as command-line parameters, passing as input file(s), and/or passing as source code that must be compiled and linked to other portions of the problem solver. An example implementation of the interface to the CRIS problem solver (function `Evaluate-test-case`) is presented in Section 5 and Appendix D.

Another support function, `Finish-quantum?`, is required to determine the number of times to call the problem solver within a single quantum. In most normal circumstances, only a single test should be performed within each quantum, and this function can be implemented in a trivial fashion.

4.5. Performance-Data Manager

The purpose of the performance-data manager is to maintain performance information of each HM on each test case that have been collected so far. This information can then be used to judge the "goodness" or the level of performance of each HM. Currently, performance information is maintained actively only for the current subdomain.

Function `Update-performance-database` is used to update existing performance information using new information on testing a new test case. The `Candidate-performance` and `Compute-fitness` functions then use the new information to estimate the overall level of performance of each HM. The results of these functions are used for selecting HMs for reproduction in the generation process as well as for scheduling future actions.

Currently, the computed performance should be based on either the average or the maximum performance over all test cases seen so far. The fitness value should also be modified by the lowest performance value in the current active pool. This modified fitness value provides a higher chance for selecting good HMs in the heuristics-generation process. If HMs can be modified in a point-based fashion (see Section 4.6), then a different aggregate measure, such as geometric mean, may be required. Currently, TEACHER has an undocumented support for this feature so that dynamic sample-allocation strategies can still work with the geometric mean.

The performance information stored can also be used for determining when poor HMs should be pruned. These HMs can be determined based on the result of function `Unacceptable-performance`. This is used only when there are multiple objectives, and constraints must be imposed on one or more performance measures. In that case, this function detects when a HM will not likely satisfy the given constraints.

4.5.1. Support for Multi-objective Constraints Manipulation

In addition to the above functions, there are several additional functions that must be implemented in order for multi-objective learning and/or DMDS scheduling to perform properly. These functions use performance data of the constrained performance measure(s) to determine the likelihood that a HM can satisfy a given level of constraint. When there is no constrained performance measure (the single-objective case), the default version of these functions (as shown in Appendix C) can be used. An example of how these functions can be implemented properly for CRIS as the target application is presented in Section 5 and Appendix D.

The main function is `Constraint-probability` that determines the probability that a HM can satisfy a given constraint level. A companion but less useful function is `Inverse-constraint-level` which determines the level of constraint that a HM can satisfy at a certain confidence level. These two functions are used both in DMDS scheduling and in the selection of HMs for verification. Function `Constraint-probability` should also be used in modifying fitness values (in `Compute-fitness`), and in selecting HMs to be removed (in `Unacceptable-performance`).

Function `Constraint-average` returns the average value of a constrained performance measure achieved by a given list of HMs. It is used by DMDS to control future constraint levels. Function `Constraint-distribution` is called when a distribution-estimation method (see Section 3.2.4) is used to reduce the minimum number of tests. This function estimates the average variance of the constrained performance measure(s). The estimated value can then be used by other functions when there are not enough test results to compute a proper variance.

Under single-objective optimization condition, the minimum-risk and non-parametric minimum-risk sample-allocation strategies (see Section 3.2.2) use the variance (*i.e.*, uncertainty) of the single objective measure as the measure of uncertainty for each HM during selection of HMs for testing. Under multi-objective conditions, however, a HM with small variance on the unconstrained performance measure but high uncertainty level on one or more constrained measures should also be evaluated further to ensure that this HM satisfies the given constraints. In this case, the measure of uncertainty for each HM should be the maximum between the uncertainty of the unconstrained measure (variance of optimized measure) and the uncertainty of the constrained measures. Function `Constraint-uncertainty` is designed to return this level of uncertainty for the constrained measures.

4.6. Heuristics Generation

This is probably the most application-specific component of TEACHER. It represents the core component of genetic algorithms within genetics-based machine learning. The objective of this component is to generate a new set of HMs for the next testing duration (generation) based on the selection and recombination process. Existing HMs are selected for reproduction based on their past performance. New HMs are generated by combining or by modifying selected HMs. For more information on details about this process, references within the genetic-algorithm and genetic-programming community should be consulted [1, 3, 4, 6]. For an example implementation of this operation, refer to the implementation of `Generate-new-candidates` for CRIS in Section 5 and Appendix D.

An alternative method for finding better HMs is to use performance feedback to modify each HM. This is known as point-based learning and can be applied in addition to the population-based genetics-based learning presented here [9]. Usually, domain knowledge is necessary for point-based modifications to be meaningful. The function `Apply-point-base` provides a hook for implementing such learning in TEACHER.

One important issue in point-based learning is the performance evaluation. Traditionally, equal weight is given to past performance data from each test case. When HMs are improved through point-based modification, performance from newer version of HMs may be more important (accurate) than earlier information. In this case, a different method for aggregating performance may be necessary (see Section 4.5), and dynamic scheduling strategies may not work with this new method. For this reason, this facility should be used only by experienced users who understand the issues involved.

4.7. Debugging and Support Functions

There are some application-specific information about the state of the learning system that are useful for debugging and monitoring learning experiments. To provide this information to users, two AppD functions are called from the AppI components of TEACHER. Function `Output-candidate-list` is called to output information about the state of HMs in a given list. The programmers can choose to output any information that is relevant to their applications.

Another function, `Candidate-objective-value`, is used for indicating the effectiveness of the learning system. It is usually used in experiments in which the actual performance of each HM is known ahead of time. This condition is usually indicated by setting the global parameter `*SIM-F*` to `T`. Under this circumstance, the actual objective values obtained from this function can be used in conjunction with the current goodness (average performance) value to show the status of the learning experiment. In other circumstances, this function should just return a prediction of the actual performance, *i.e.*, the current goodness value.

5. EXAMPLE APPLICATION-DEPENDENT IMPLEMENTATION

In this section, we present an example of how to interface TEACHER to a target application. First, we present an overview of the chosen target application, CRIS. Second, we discuss some of the key points in implementing the AppD functions for CRIS. We then discuss some of the key issues in extending these AppD functions to deal with multiple objectives in the application.

5.1. Target Application

CRIS [8] is a genetic-search package for generating test patterns for VLSI circuits. It was developed by experts in the area who have identified seven parameters to learn. Each of these parameters has its range specified in Table 5.1. A HM in this case is a set of these seven parameters.

These parameters are control parameters used in a genetic search for finding the best test pattern. The performance of a test pattern obtained is characterized by its fault coverage. The relation between control parameters and fault coverage is unknown.

A test-case for CRIS is a circuit to be tested. This circuit is represented by a file containing the circuit description. In addition, there is a random seed used within CRIS that affects the performance results. A set of parameters that represents a HM for CRIS is passed to CRIS as an input file along with the name of the file containing the description of the target circuit and a random seed. The performance of CRIS, represented by the fault coverage, is output to a file that can then be extracted by TEACHER. Our current implementation also stores the amount of CPU time used by CRIS.

Table 5.1. Parameters of CRIS use as components of HM

Parameter	Parameter Type	Parameter Range	Definition
x_1	integer-valued	[1,10]	survival rate
x_2	integer-valued	[1,20]	FF stages number
x_3	integer-valued	[1,40]	test vectors number
x_4	real-valued	[0.1,10.0]	sensitivity
x_5	integer-valued	[50,800]	trials failure number
x_6	integer-valued	[1,20]	generations number
x_7	real-valued	[0.1,1.0]	genes splicing

5.2. Implementation Details

Based on the information described above, an implementation of the AppD functions for CRIS is shown in Appendix D. In this section, we discuss some of the salient points related to developing the AppD code for CRIS.

5.2.1. General Information

As mentioned previously, a HM is represented by a list of seven numbers. Each HM is identified by an integer starting from 1. The AppD data structures used are declared in ‘‘struct.cl’’ while all AppD variables (`*base-heur-cand*` and `*heur-boundary-vectors*`) and parameters are declared in ‘‘var.cl’’. One important control parameter is `*PERFORMANCE-METRIC*` that controls the method for computing the performance of HMs. It can be set to use average (`'AVG-METRICS`) or maximum (`'MAX-METRICS`) performance value.

All initialization functions are in ‘‘domain.cl’’. For CRIS, we can optionally define a baseline HM as a basis for performance comparison. This baseline HM, if defined by users through domain-specific parameter `*BASE-HEUR-FORM*`, is initialized in `Reset-candidate-pool`. A set of previously created HMs can also be read from file ‘‘Exec/cris/heuristic-base’’ by this function, if such a file exists. In addition, function `Initialize-domain` makes sure that old result and input files from previous experiments are removed before the start of a new experiment.

5.2.2. Test-Case Management

For this application, a test-case is represented by (1) name of the circuit description file, (2) number of inputs to the circuit, and (3) random seed used by CRIS. The first two parameters are always the same for the same circuit. Note that the performance of the same HM on the same circuit with different random seeds can be different. A subdomain can be composed of a single test circuit with different random seeds or multiple circuits with one or more random seeds.

The desired test-case database is read in from file ‘‘test.db’’ by function `Reset-test-database`. The sequence of test-cases within each subdomain is as specified in file ‘‘test.db’’. However, it is possible for the sequence to be randomized within the initialization function. The function `Get-test-case` then gets the first untested test-case from the test database. This allows different HMs with the same number of tests to be evaluated on the same set of test cases.

5.2.3. Problem-Solver Interface

There are three main components in implementing function `Evaluate-test-case` for CRIS. The first and most important part is to invoke CRIS to solve a circuit with a given HM and a given random seed. This is implemented within function `Interface-solver` called by `Evaluate-test-case`. This function first outputs the given HM parameters along with the random seed of the current test case to a file. It then uses this file as input to CRIS along with the filename of the given circuit. This function also redirects CRIS’s result to a file and then extracts the fault coverage and CPU-time usage from this file. Finally, this fault-coverage value is used as the performance of the given HM on the given test-case.

The second component of `Evaluate-test-case` deals with maintaining a database of HMs and their performance outside TEACHER. A set of functions (`PM-Scan-rdb-directory`, `PM-Scan-perfm-rdb`, and `PM-Write-perfm-rdb`) defined in ‘‘resilient.cl’’ is used to accomplish this functionality. This set of functions allow users to find the definition of any HM used in a learning experiment. In addition, each learning experiment can be duplicated without having to call CRIS again for performance information that has been recorded previously. One negative aspect of this scheme is the large overhead in searching through a performance file for each performance result when there are many performance results for each HM.

The last component of `Evaluate-test-case` makes sure that there exists performance values of the baseline HM (see Section 5.2.1) for the current test case if the baseline HM is defined (i.e. `*BASE-HEUR-FORM*` is not `NIL`). Without this information, the resulting performance values cannot be normalized for the `'AVG-METRICS` performance objective. This is accomplished by calling function `PS-Enforce-baseline-test`.

5.2.4. Performance-Data Manager

Function `Update-performance-database` first computes the normalized performance, *i.e.*, ratio of improvement, that is used as the performance of a HM when `'AVG-METRICS` is the performance objective and a baseline HM is defined. Otherwise, the raw fault coverage is used as the HM performance. Normalized performance is computed by comparing the resulting fault coverage with the fault coverage of the baseline HM on the same circuit. These actions are performed by functions `PM-Equal-weight-norm` and `PM-Get-inst-perfm-value`, respectively.

The setting of `*PERFORMANCE-METRIC*` is also used for determining the result of function `Candidate-performance`. The maximum value so far is returned when this control parameter is set to `'MAX-METRICS`. Otherwise (*i.e.*, `'AVG-METRICS`), this function returns the average performance so far.

The fitness value is computed by `Compute-fitness`; this computation based on the goodness value which is the same as the result of `Candidate-performance`. The worst goodness value among all HMs in the active pool is applied as a bias to increase the spread of the ratios between an individual fitness value to the total fitness value within the active pool. This allows selective pressure in the heuristics-generation process as better HMs are found.

5.2.5. Heuristics Generation

Our current implementation of `Generate-new-candidates` for CRIS uses four methods for generating new HMs: random, cross-over, mutation, and greedy mutation. The number of HMs generated by each method is based on the AppD control parameters defined in ‘‘var.cl’’. In HM generation, HMs are selected based on their fitness values (`Fitness-select`). This algorithm makes sure that all components of a new HM are within the specified range of value. The algorithm then adds the new HM to the active pool.

One special set of random HMs are generated at the beginning of a learning experiment by function `Initialize-domain`. This set of HMs represent the boundary of the heuristics space and are generated by function `GEN-Initialize-boundary-vectors`. This set of HMs are added to make sure that all extreme conditions are represented in the HM pool.

5.3. Example of Multi-objective Heuristics Learning

To illustrate the issue involved in implementing AppD functions under multi-objective conditions, we have created an example implementation for CRIS under this condition. We select the CPU time used by each test case as a constrained performance measure. The differences between this condition and the one discussed in the last section are shown in the file ‘‘multiobj.cl’’. These new functions include all multi-objective support functions (Section 4.5.1), `Compute-fitness`, and `Unacceptable-performance`.

To understand this code, we must first explain the format of global variable `*RT-CONSTRAINT*`. This global variable is a list with two parameters. The first parameter deals with the current performance-objective level that does not have to be considered by the AppD components. The second parameter is a list of three elements that control the level of constraints during a learning experiment. The first element specifies the current target constraint level; the second element, the minimum degree of certainty on satisfying the constraints before the HM can be accepted completely; the third element, the minimum degree of certainty that must be achieved (otherwise, the HM would be eliminated).

The main function that deals with uncertainties in constraint satisfaction is `Constraint-probability`. Our current implementation uses the mean and variance of a constrained measure to compute this probability. Since variances are to be found, a minimum of four tests are required before the actual probability can be computed.

Most other functions use this constraint probability to decide on their results. Function `Compute-fitness` uses this constraint probability to reduce the fitness value when the amount of certainty is lower than that desired. Function `Unacceptable-performance` selects for elimination HMs with constraint probabilities below the minimum threshold.

6. CONDUCTING AN EXPERIMENT

In this section, we show how a learning experiment can be conducted using TEACHER. To accomplish this, an implementation of the AppD functionalities described in Section 4 must exist for the desired target application. Here, we use CRIS as our target application with its AppD components implemented as described in Section 5.2.

6.1. Overview

First, the code for the AppD components must be compiled and optimized. This is accomplished by entering Common-LISP, loading the object code of the AppI components and the source code of the AppD components, and compiling the AppD components. The following segment is used for compiling the AppD learning components of CRIS: ‘‘`cl < compile-all`’’ with the contents of the file ‘‘compile-all’’ as follows.

```
;; Load the object code of the AppI component
(load "teacher")

;; Load all source code of the AppD component
(Load-dependent)    ;; equivalent to (load "domain") (Load-up-domain)

;; Compile each AppD source file
(compile-file "./domain/Src/struct.cl"
 :output-file "./domain/Obj/struct.fasl" )

(compile-file "./domain/Src/var.cl"
 :output-file "./domain/Obj/var.fasl" )

(compile-file "./domain/Src/domain.cl"
 :output-file "./domain/Obj/domain.fasl" )

   ...
```

Next, we are ready to run an experiment to learn new HMs for CRIS. In each experiment, the various AppI and AppD parameters must be set appropriately. An input file is used to control the action within the Common-LISP environment. First, the object code for both the AppI and AppD components must be loaded in the appropriate order (AppI before AppD). Then, the various global parameters must be set to their appropriate values. The list of AppI global parameters are shown along with their default values in Appendix B. Users should also read Section 3, 6.2, and 7 for more details on these global parameters. Finally, the main learning code is called from the input script. Before the start of each experiment, the seed for the random number generator should be set to a known value so that results can be reproduced in a deterministic fashion. This is accomplished using the `Seed-random` function in TEACHER.

Here, we show an example script for learning HMs for CRIS using 1200 logical time units for each one of the five learning subdomains. The scheduling strategy used is the non-parametric minimum-risk sample-allocation strategy with fixed duration (120 time unit/generation). 30 HMs are active at any point in time with 20 HMs replaced at the end of each generation. 20 HMs are then selected for verification at the end of each one of the five learning phases. Finally, the generalization component selects the best six HMs to cover the given set of subdomains. The performance objective is to optimize the average ratio of improvement in fault coverages over the baseline HM.

```
;; Load all object codes
(load "teacher")
(Load-dependent)    ;; equivalent to (load "domain") (Load-up-domain)

;; Set global parameters
(setq *TIME-LIMIT*       1200)
(setq *GENERATION-LIMIT*   10)
(setq *ASYNCH-RUN-F*      NIL)
(setq *DMDS-SCHED-F*      NIL)
(setq *STRATEGY-TYPE*   ’Min-risk)
(setq *NUM-CANDIDATES*     30)
(setq *PERCENT-RETAIN*   0.33)
(setq *VERIFY-SELECT*      20)
(setq *NUM-LEARN-PHASES*    5)

(setq *GEN-HM-DESIRED*      6)

;; Set output level
(setq *DEBUG-LEVEL*         1)
(setq *PROGRESS-LEVEL*      1)
```

```
(setq *PERF-RECFILE*   "cris-result")

;; Set AppD parameters
(setq *PERFORMANCE-METRIC*  'AVG-METRICS)
(setq *BASE-HEUR-FORM*      (list 3 20 20 3.85 442 1 0.1) )

;; Start learning experiment
(Seed-random 1413)
(main)
```

Before starting this learning experiment, it is necessary for users to make sure that all the application-specific conditions are met. For example, directory ‘‘Exec/cris/’’ with executable ‘‘cris’’ must exist before conducting the above experiment. In addition, file ‘‘test.db’’ must exist in the current directory and contains at least five subdomains. In this experiment, ‘‘test.db’’ actually contains 15 subdomains, each representing a circuit. Also, data files describing each circuit within ‘‘test.db’’ must exist in directory ‘‘Exec/cris/data/’’.

```
;; list of classes
(  15  ) ; number of classes

;; list of test instances per class
(1 ; class
   ( ;; ( CircuitName  NumOfInputs  RandomSeed )
     (s386 7 61801)
     (s386 7 98052)
     (s386 7 15213)
     ...
   )
)

(2 ; class
   (
     (s344 9 61801)
     (s344 9 98052)
     ...
   )
)
   ...
```

6.2. Performance Results

In this section, we describe the application-independent results that will be produced by TEACHER for each learning experiment. We also describe various parameters for controlling the information output by TEACHER. See Appendix B and Table 6.1 for a complete listing and a brief description of all control parameters.

There are three different types of information output by TEACHER during each experiment: (1) result of generalization and selection, (2) progress report during the experiment, and (3) debugging information and error messages. Each type of information is discussed briefly below.

(1) *Final result of generalization and selection*. This information shows the result of each learning experiment. Specifically, it shows the partitioning of tested subdomains based on the performance found, and the HM selected for each partition. This information is generated at the end of each verification/generalization phase and is output/appended to a file specified by global parameter `*PERF-RECFILE*`. An example from the experiment described in Section 6.1 is shown below. The number within the first parenthesis represents the subdomain name(s), while the two entries in the second parenthesis represent the ID of the selected HM and its average probability-of-win within the current partition.

```
PARTITION=(12): (101 0.9405976)
PARTITION=(1 5): (108 0.8962431)
PARTITION=(4 11): (104 0.8917674)
PARTITION=(2 10 13): (158 0.9211097)
PARTITION=(8 14 15): (77 0.9009816)
PARTITION=(3 6 7 9): (535 0.7815812)
PARTITION="default heuristics": (188 0.7073738)
```

Also generated is a file ‘‘`*PERF-RECFILE*`-probe’’. In the current example, ‘‘cris-result-probe’’ provides a performance summary for each HM verified. Each list of five elements has the HM’s ID, its performance value for the current subdomain, its performance standard deviation, number of test cases, and probability of constraint violation.

```
;;(setq *random-state* #S(random-state :seed 31396254056065))
;;(setq *Random-element-state* #S(random-state :seed 164068919845701))
;;(setq *Random-float-state* #S(random-state :seed 220967091996383))
;; Generation Limit = 10
;; Learning Strategy Type = min-risk
;; Time Limit = 1200
;; Number of Stage I Candidates = 30
;; Number of Stage II Candidates = 20
;; Percentage of Retaining =  0.330
(setq *v-LEARNED-SUBDOMAINS* '(1 2 3 4 5))
(setq *v-TESTED-SUBDOMAINS* '(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15))
(setq *v-PERFORMANCE-ON-EACH-SUBDOMAIN*  '(
    (
        (571    0.99632    0.01341  10    1.00000)
        (418    0.99599    0.01318  10    1.00000)
        (585    0.92676    0.03682  10    1.00000)
        (72    1.03471    0.01017  10    1.00000)
        ...
    )
    (
        (571    0.18379    0.02301  10    1.00000)
        (418    0.94938    0.07393  10    1.00000)
        (585    0.18329    0.05763  10    1.00000)
        (72    0.92284    0.06467  10    1.00000)
        ...
    )

    ...

    (
        (571    0.98222    0.00888  10    1.00000)
        (418    0.99244    0.00707  10    1.00000)
        (585    0.97794    0.02057  10    1.00000)
        (72    0.97920    0.00683  10    1.00000)
        ...
    )
))
```

(2) *Progress report*. This shows the status of the learning system as learning progresses. This information is also output to the file specified by `*PERF-RECFILE*`. There are several levels of status information that is controlled through the setting of global parameter `*PROGRESS-LEVEL*`. The amount of information increases as the value of this global parameter increases.

In Level 0, only the start and the end of each learning and verification phase is reported. In Level 1, the end of each generation and the end of each segment for DMDS are added. In Level 2, the status of the learning system at the end of each generation and each segment is added. This information is presented in a form that could be used if learning were to stop at that time. In Level 3, the list of HMs that are active at the end of each generation is also reported. In Level 4, performance information for each HM listed in Levels 2 and 3 is provided. In addition, in all levels above 1, the list of candidate HMs and their performance at end of each learning phase is output to the standard output.

An example of a progress status report in level 1 is shown below.

```
End of generation 1     at 120   quanta 120      seconds
End of generation 2     at 240   quanta 240      seconds
End of generation 3     at 360   quanta 360      seconds
   ...
End of generation 10    at 1200  quanta 1200     seconds
End of learning iteration 0 for subdomain 1 at 1200  quanta 1200     seconds
End of generation 1     at 120   quanta 120      seconds
   ...
End of generation 10    at 1200  quanta 1200     seconds
End of learning iteration 1 for subdomain 2 at 2514  quanta 2514     seconds
   ...
```

Unless the program is likely to crash before it finishes, we recommend using Level 0 or 1 is used. Otherwise, it is necessary to get as much information as possible so that the learning experiment can be restarted at the point closest to where the original program was stopped.

(3) *Debugging information and error report*. This type of information is output to the standard output that can be redirected to a file. The amount of information provided is dependent on the setting of global parameter `*DEBUG-LEVEL*` as well as the setting of various debugging flags described in Section 7.

The amount of information for each type of debugging increases as the value of `*DEBUG-LEVEL*` increases. In Level 0, no debugging information is provided, and only ERROR messages are reported. In Level 1, information about actions that users want to know about (through activation of certain debugging flags) is reported. In Level 2, the system reports its status at the point when actions were performed. In Level 3, all information related to actions described in Level 2 is provided.

Unless detailed debugging is needed, Level 1 debugging should be enough for most circumstances. In higher levels, there is usually too much information to be understood by users.

Table 6.1. Global parameters for controlling observable information output

Parameter	Description	Range	Default
`*PERF-RECFILE*`	Name of output file	Filename	"perfdata.record"
`*PROGRESS-LEVEL*`	Amount of progress information	Integer ≥ 0	1
`*DEBUG-LEVEL*`	Amount of debug information	Integer ≥ 0	1

7. CONTROLLING THE BEHAVIOR OF LEARNING EXPERIMENTS

From Sections 3, 4, and 6, we observe that there are many different functionalities built into TEACHER. The users of TEACHER must be able to select the functionalities they want from among the available alternatives. In the current implementation, users can control and select the proper functionalities by setting the various global parameters to their proper values before starting each learning experiment. In this section, we summarize the purpose of existing global parameters. We divide these parameters into several groups based on their functions. A complete listing of all global parameters along with their default values are shown in Appendix B.

(1) **Resource Specification**. These four parameters (`*CPU-LIMIT*, *TIME-LIMIT*, *ASYNCH-RUN-F*, *NUM-LEARN-PHASES*`) specify the amount of resources available within a learning experiment (see Section 3.1). These parameters, especially `*TIME-LIMIT*` and `*NUM-LEARN-PHASES*`, must be set in each learning experiment.

(2) **Resource Scheduling Control**. This category of global parameters control the behavior of the resource scheduling component of TEACHER (see Section 3.2). They can be divided further into 4 subcategories.

(2-A) *Generation-Control Parameters*. This set of parameters (`*NUM-CANDIDATES*, *PERCENT-RETAIN*, *RETAIN-STRATEGY*`) determine the number of HMs in the active pool and the number of new HMs to be generated in each generation (Section 3.2.1).

(2-B) *Sample-Allocation Control*. This set of parameters (`*STRATEGY-TYPE*, *SCHED-GOAL*, *ALLOC-MIN-CT*`) determine the sample-allocation strategy to be used (Section 3.2.2).

(2-C) *Duration-Scheduling Control*. This set of parameters (`*GENERATION-LIMIT*`, `*DMDS-SCHED-F*`) determine the duration scheduling strategy and the actual duration of each generation under a fixed-duration strategy (Section 3.2.3).

(2-D) *Distribution-Estimation Control*. This set of parameters (`*DIST-TYPE*`, `*DYNAMIC-UPDATE-F*`, `*UPDATE-THRESH*`, `*PRE-SAMPLE-COUNT*`, `*PRE-SAMPLE-FRAC*`, `*PRE-SAMPLE-MIN-CAND*`, `*TEST-SCHED-F*`) control the distribution-estimation process (if any is to be used). Section 3.2.4 contains details about their functions.

(3) **Verification Control**. `*VERIFY-SELECT*` specifies the number of HMs to be selected for verification from each learning phase (see Section 3.3).

(4) **Generalization Control**. `*GEN-HM-DESIRED*` controls the number of HMs to be selected at the end of an experiment, and `*GEN-MAXDEV-DESIRED*` specifies how close should the selected HMs be to the top HM in each subdomain.

(5) **Multi-objective Control**. The two control parameters that control the level of constraints for multi-objective learning are `*GLOBAL-CONSTRAINT*` and `*RT-CONSTRAINT*`. `*RT-CONSTRAINT*` also specifies the parameters of the DMDS scheduling strategy (see Sections 3.2.3 and 5.3).

(6) **Learning-System-Result Specification**. This set of parameters (`*PERF-RECFILE*` that specifies the name of the file to record result, and `*PROGRESS-LEVEL*` that specifies the amount of status information to be reported to users) specify the amount of information that users want to obtain during and after a learning experiment (see Section 6.2).

(7) **Debugging Control**. This set of parameters control the amount of debugging information to be reported by each component of the learning system. The global flag `*DEBUG-F*` turns on all debugging information. If this flag is off, it is up to the individual component's debugging flags (`*DEBUG-LOAD-F*`, `*DEBUG-GENERATE-F*`, `*DEBUG-EXECUTE-F*`, `*DEBUG-GUIDANCE-F*`, `*DEBUG-SCHEDULER-F*`, `*DEBUG-HEUR-MANAGER-F*`, `*DEBUG-PRE-SAMPLE-F*`) to indicate whether any debugging information should be reported. Once TEACHER performs properly for a target application, it is recommended that only debugging information about the generation of new HMs be reported (`*DEBUG-GENERATE-F*`). The amount of information

provided is dependent on the value of the global parameter `*DEBUG-LEVEL*` (see Section 6.2).

Another debugging flag is `*SIM-F*` which indicates that the system knows the actual performance level of each HM within the system (see Section 4.7). This should be used only during testing of learning-system functionalities.

8. CONCLUDING REMARKS

TEACHER is an evolving software system for supporting our research on designing resource-constrained automated learning for knowledge-lean application domains. It is designed to support a wide spectrum of learning strategies and target applications. Its goal is to separate application-independent (AppI) components from application-dependent (AppD) ones. With this separation, AppI functionalities of TEACHER can be enhanced and improved upon without additional efforts on the implementation of AppD functionalities. In addition, TEACHER can be easily applied to new applications through the implementation of AppD functionalities. Finally, numerous options in TEACHER can be controlled by the users through the setting of global control parameters. This allows users to customize TEACHER to suit their requirements.

ACKNOWLEDGEMENT

We would like to acknowledge individuals who have contributed to the implementation of TEACHER we see today. Akiko Aizawa and Steven Schwartz (for helping in laying foundations of TEACHER), Lon-Chan Chu (for contributing to the development of the generalization process), Pankaj Mehra, Yong-Cheng Li, and Lon-Chan Chu (for providing feedback on applying TEACHER to new target applications). Our special thanks go to Steven Schwartz who wrote the first albeit primitive version of TEACHER and whose influence can still be seen in how the code is structured and commented.

REFERENCES

[1] L. B. Booker, D. E. Goldberg, and J. H. Holland, ‘‘Classifier Systems and Genetic Algorithms,’’ in *Machine Learning: Paradigm and Methods*, ed. J. Carbonell, MIT press, 1990.

[2] L.-C. Chu, *Algorithms for Combinatorial Optimization in Real Time and their Automated Refinement by Genetic Programming*, Ph.D. Thesis, Dept. of Electrical and Computer Engineering, Univ. of Illinois, Urbana, IL, May 1994.

[3] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley Pub. Co., 1989.

[4] J. J. Grefenstette, C. L. Ramsey, and A. C. Schultz, ‘‘Learning Sequential Decision Rules using Simulation Models and Competition,’’ *Machine Learning*, vol. 5, pp. 355-381, Kluwer Academic Pub., Boston, MA, 1990.

[5] A. Ieumwananonthachai, A. Aizawa, S. R. Schwartz, B. W. Wah, and J. C. Yan, ‘‘Intelligent Process Mapping Through Systematic Improvement of Heuristics,’’ *J. of Parallel and Distributed Computing*, vol. 15, pp. 118-142, Academic Press, June 1992.

[6] J. R. Koza, *Genetic Programming*, The MIT Press, Cambridge, MA, 1992.

[7] R. S. Michalski, ‘‘Understanding the Nature of Learning: Issues and Research Directions,’’ in *Machine Learning: An Artificial Intelligence Approach*, ed. R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, vol. II, pp. 3-25, Morgan Kaufmann, Los Altos, CA, 1986.

[8] D. G. Saab, Y. G. Saab, and J. A. Abraham, ‘‘CRIS: A Test Cultivation Program for Sequential VLSI Circuits,’’ *Proc. of Int’l Conf. on Computer Aided Design*, pp. 216-219, IEEE, Santa Clara, CA, Nov. 8-12, 1992.

[9] B. W. Wah, ‘‘Population-Based Learning: A New Method for Learning from Examples under Resource Constraints,’’ *Trans. on Knowledge and Data Engineering*, vol. 4, no. 5, pp. 454-474, IEEE, Oct. 1992.

[10] B. W. Wah, A. Ieumwananonthachai, L. C. Chu, and A. Aizawa, ‘‘Rational Scheduling of Experiments and Generalization in Genetics-Based Learning of New Heuristics,’’ *IEEE Trans. on Knowledge and Data Engineering*, IEEE, (accepted to appear) 1995.

APPENDIX A - APPLICATION INDEPENDENT INFORMATION

A.1 Application Independent Data Structure Definition

```
(defstruct ( candidate-list-S (:conc-name cand-list-S-) )

"List of candidates considered by the learning framework - includes:
  - list of active candidates being considered
  - list of candidates that have been tested over the entire subdomain
  - list of acceptable candidates not currently active
  - list of candidates pruned due to low performance
  - list of candidates discarded due to violation of constraints
  - list of candidates selected for verification phase
  - number of new candidate generated so far"

  (active-list       nil )         ; list of active candidates
  (finished-list     nil )         ; list of finished candidates
  (retired-list      nil )         ; list of retired candidates
  (pruned-list       nil )         ; list of pruned candidates
  (discard-list      nil )         ; list of discarded candidates
  (verify-list       nil )         ; list of candidates for verification
  (num-gen             0 )         ; number of new candidates generated

) ;; end of candidate-list-S

(defstruct ( resource-constraints-S )

"Specific amount of resources available to the system for learning"

  (time-limit        (Hours 1) )   ; time-limit in seconds/virtual time
  (num-processors           1  )   ; number of processors available

) ;; end of resource-constraints-S

(defstruct ( learning-params-S )

"Parameters specifying the learning algorithm - includes:
  - limit on amount of time to spend on one candidate"

  (quantum           nil )         ; duration of a quantum (in seconds)

) ;; end of learning-params-S

(defstruct ( cpu-status-S )
"Show current status of a CPU - includes:
  - CPU id
  - time used so far by current CPU in current learning phase or overall
  - current candidate being evaluated
  - test case being evaluated on
  - result of evaluation
  - status of current quantum
  - flag indicate done with current candidate"

  (cpu-id              1 )         ; id of CPU
  (time-used           0 )         ; time used by this CPU
  (curr-candidate    nil )         ; candidate being evaluated
  (test-case         nil )         ; current test case for evaluation
  (eval-result       nil )         ; result of current evaluation
  (quantum-status    nil )         ; status of current quantum so far
  (finish            nil )         ; flag indicate quantum done

) ;; end of cpu-status-S

(defstruct ( quantum-status-S )

"The data structure for the status of quantum for learning."

  (time                    0  )    ; time used by current quantum
  (test                    0 )     ; # tests done in current quantum

) ;; end of quantum-status-S

(defstruct ( candidate-S )

"The data structure for a potential candidate.
This currently consists of:
 - candidate identity string
 - The heuristics set specification
 - Performance history
 - Status flag
 - Distribution info
 - Test case type information"

  (identity             NIL )      ; identity string
  (heuristic-spec       NIL )      ; a set of heuristics
  (performance-history  NIL )      ; totals, and a list of performance
                                   ;   entries
  (objective-value        0 )      ; for SIMULATION/DEBUG (*SIM-F*)
  (goodness-value         0 )      ; current goodness value
  (guidance-value         0 )      ; current guidance value
  (status               "A" )      ; status string, "A" = active,
                                   ; "*" = finished, "+" = retired,
                                   ; "=" = pruned, "-" = discarded,
                                   ; "I" = incumbent
  (inuse                  0 )      ; flag indicate if being used
  (scheduler-info       NIL )      ; scheduler dependent data field
  (test-type              1 )      ; type of next test case
  (multitype-history    NIL )      ; for multitype of test cases
  (generation-history   NIL )      ; for generation of new candidate (OPT)

) ;; end of candidate-S

(defstruct ( performance-history-S (:conc-name perf-hist-S-) )

"The data structure for a performance history record."

  (quantum-count           0 )     ; number of quantum used
  (instance-count          0 )     ; number of test instances tested
  (norm-inst-count         0 )     ; norm instance count for point-based
  (time-tested           0.0 )     ; time tested (in seconds)
  (sum-perf              0.0 )     ; sum of performance values
  (sum-perf-sqr          0.0 )     ; sum of squares of performance values
  (performance-instances NIL )     ; a list of performance instances

  (test-count              0 )     ; number of tests run (OPTIONAL)

) ;; end of performance-history-S

(defstruct ( performance-instance-S (:conc-name perf-inst-S-) )

"The data structure for a performance record over one test instance."

  (performance           0.0 )     ; performance for this instance

) ;; end of performance-instance-S
```

A.2 Statistical Support Routines

`(Mean number-list)`	- return mean, moment($E[X^k]$), sample standard
`(Moment moment number-list)`	deviation, standard deviation, and the margin of
`(Sample-std-dev number-list)`	confidence (assume Student's T-distribution) of the
`(Std-dev number-list)`	given list of number
`(T-margin conf num-list)`	

`(F-cdf z-val df1 df2)`
`(Gauss-cdf z-val)`
`(Student-T-cdf z-val df1)`
	- find cdf value of the given z-val for the given distribution and degree of freedom(s)

`(Inverse-F-cdf prob df1 df2)`
`(Inverse-gauss-cdf prob)`
`(Inverse-T-cdf prob df1)`
	- find inverse function to c.d.f. of given distribution with specified degree of freedom(s)

A.3 Randomization Routines

`(Random-element alist)`	- return a random element from the given list
`(Random-subset alist size)`	- return a random subset of given size from list
`(Random-float num-a num-b)`	- return a random number between num-a and num-b
`(Seed-random seed)`	- set the seed values for all random numbers

A.4 Miscellaneous Manipulation Routines

`(Max-of list metric)`	- return max element in list based on given metric
`(Min-of list metric)`	- return min element in list based on given metric
`(Sort-descending list rankf)`	- sort list in descending order using the given ranking function
`(Sqr number)`	- return the square of the given number
`(Subset list test)`	- return a subset of list that pass given test
`(Vector-length num-list)`	- return vector length (2-norm) of given vector
`(Normalize-vector vector)`	- return vector scaled to unit length
`(Dot-product vec-1 vec-2)`	- return inner product/dot product of two vectors

APPENDIX B - CONTROL PARAMETERS FOR TEACHER SYSTEM

B.1 Resources Parameters (Table 3.1)

TIME-LIMIT	1000	Number of quantums per subdomain (>= 1)
CPU-LIMIT	1	Number of CPUs to simulate (>= 1 and integer)
NUM-LEARN-PHASES	1	Number of learning phases/subdomains (>= 1 and integer)
ASYNCH-RUN-F	NIL	if T then use actual CPU usage

B.2 Resource Scheduling Control (Table 3.2)

Heuristics Generation

NUM-CANDIDATES	30	Number of active candidates(>= 1 and integer)
NUM-CANDIDATES-2	20	(>= 1 and <= *NUM-CANDIDATES* and integer)
PERCENT-RETAIN	0.33	Percent candidates retained for more testing (>= 0 and <= 0.33)
RETAIN-STRATEGY	'BEST	Method for select retained candidates ('RANDOM)

Sample Allocation

STRATEGY-TYPE	'Min-risk-1	('Min-risk, 'Round-Robin, 'Round-robin-1, 'Greedy) Name of scheduling strategy to use
SCHED-GOAL	'SELECTION	Goal to optimize during scheduling ('TOP-ONE)
ALLOC-MIN-CT	4	Min #-of-tests/candidate before dynamic scheduling

Duration Scheduling

GENERATION-LIMIT	10	Number of generation per subdomain(>= 1 and integer)
DMDS-SCHED-F	NIL	if T then use DMDS duration scheduling

Distribution Estimation

DIST-TYPE	'UNKNOWN	Distribution of performance within each candidate('NORMAL)
DYNAMIC-UPDATE-F	NIL	if T then use simple dist. data update each generation
UPDATE-THRESH	20	# new candidates before update dist. data(> 0)

PRE-SAMPLE-COUNT	4	Number of quantums to evaluate each candidate
PRE-SAMPLE-FRAC	0.1	Min. fraction of generation duration used
PRE-SAMPLE-MIN-CAND	4	Min. # candidates to pre-sampling
TEST-SCHED-F	NIL	if T then discount pre-sampling time

B.3 Verification Control Parameters (Table 3.3)

VERIFY-SELECT	20	# candidates to verify per learning phase (>= 1 and integer)

B.4 Generalization Control Parameters (Table 3.3)

GEN-HM-DESIRED	1	# candidates to select for future use (>= 1 and integer)
GEN-MAXDEV-DESIRED	1.0	Max desired different in P_{win} from best value (0 <= x <= 1)

B.5 Debugging Flag

`*DEBUG-LEVEL*`	`1`	Control amount of debugging info to output
`*SIM-F*`	`NIL`	Run learning in simulation mode
`*DEBUG-F*`	`NIL`	Control all debugging code on/off
`*DEBUG-EXECUTE-F*`	`NIL`	Each of this control individual component debugging code on/off
`*DEBUG-GENERATE-F*`	`T`	
`*DEBUG-HEUR-MANAGER-F*`	`NIL`	
`*DEBUG-LOAD-F*`	`NIL`	
`*DEBUG-RULEBASE-F*`	`NIL`	
`*DEBUG-SCHEDULER-F*`	`NIL`	
`*DEBUG-GUIDANCE-F*`	`NIL`	
`*DEBUG-PRE-SAMPLE-F*`	`NIL`	

B.6 Multi-objective Constraint Control

`*RT-CONSTRAINT*`	`(-∞ 0.75 0.25) (∞ 0.6 0.35) )`	Current/initial level of constraint in use
`*GLOBAL-CONSTRAINT*`	`∞`	Final constraint level desired

B.7 Output Control (Table 6.1)

`*PERF-RECFILE*`	`"perfdata.record"`	Name of file to store learning result and progress
`*PROGRESS-LEVEL*`	`1`	Control amount of progress to report

APPENDIX C - INTERFACE TO APPLICATION DEPENDENT FUNCTIONS

C.1 Initialization and Clean-up Routines

These routines are called at the start of the learning system or after the learning process have finished.

`(Load-up-domain)`

Description: Load up all domain dependent files other than "domain.cl"

`(Initialize-domain)`

Description: Initialize domain-dependent variables

Default: No action

`(Clean-up-domain)`

Description: Clean-up domain dependent stuffs after the system complete

Default: No action

`(Reset-quantum-def)`

Output: Amount of tests to perform within 1 quantum

Description: Return definition of amount of tests to perform within 1 quantum

Default: `1` - one test per quantum

`(Reset-candidate-pool)`

Output: A list of candidate as starting point

Description: Reset the candidate pool to original state

Default: `NIL` - empty list, i.e. start with random set

`(Reset-performance-database)`

Output: A structure for recording candidates performance

Description: Reset the performance database maintained by the learning system

Default: `NIL` - record performance within `candidate-S` structure instead

`(Reset-test-database)`

Output: A data structure represent the test database

```
( ( test-type ( -- list of test cases -- ) )
  ( test-type ( -- list of test cases -- ) )
     ...
)
```

Description: Initialize the test database for current application

Default: `( (1 test-cases-list) )` - one subdomain

C.2 Test-Case Management Routines

`(Get-test-case candidate test-database perf-database)`

Input: A candidate to be tested
  Current subdomain test database
  Database of past performances

Output: A data structure represent the desired test-case

Description: Find a test-case for further evaluation of candidate

Example: The first untested test-case in the given test database

```
(nth (perf-hist-S-instance-count
       (candidate-S-performance-history candidate) )
     test-database)
```

`(Initialize-test-subdomain test-database type-index)`

Input: Test database
  Current subdomain index

Description: Domain dependent initialization so that new subdomain can be tested

Default: Perform no action

C.3 Problem Solver Interface Routines

`(Evaluate-test-case cpu-status)`

Input: Current CPU status (see Section 3.1)

Output: The result of apply problem solver using candidate on the test case

Description: Call problem solver to use given candidate on given test case

Example: Too complicated. See Section 6 and Appendix D.

`(Finish-quantum? cpu-status test-database)`

Input: Current CPU status (see Section 3.1)
  Current test-database

Output: `NIL` if not finish with this quantum

Description: Check if current quantum is finished

Default: `T` - all quantum is composed of a single test

C.4 Performance Evaluation Routines

`(Update-performance-database perf-db candidate test-case result)`

Input: Current performance database structure (default `NIL`)

Candidate with new result

Test case associate with the new result

New result

Description: Update performance database based on new results from evaluating a candidate on a test case

Example: Too complicated (see Appendix D for example)

`(Candidate-performance candidate)`

Input: Candidate to be evaluated

Output: A performance value

Description: Calculate the known performance of the given candidate

Example: Mean value of past performance

```
(/ (perf-hist-S-sum-perf
     (candidate-S-performance-history candidate) )
   (perf-hist-S-instance-count
     (candidate-S-performance-history candidate) ) )
```

`(Compute-fitness candidate-list)`

Input: A list of candidates

Output: A list of fitness value

Description: Calculate the fitness of the candidate in the list

Example: `(mapcar #'Candidate-performance candidate-list)`

`(Unacceptable-performance candidate)`

Input: A candidate

Output: `T` if performance is unacceptable

Description: Check if the performance of the candidate is acceptable

Default: `NIL` - performance always acceptable

C.4.1 DMDS Constraints Handler Routines

`(Constraint-average candidate-list)`

Input: A list of candidates

Output: Average values of the constrained measure

Description: Compute the average constrained measure of given candidates

Default: `most-negative-single-float` - no constrained measure

`(Constraint-distribution candidate-list)`

Input: A list of candidates

Description: Compute distribution information about uncertainty
in constrained measure based on performance of give candidates

Default: Perform no action when there are no constrained measure

`(Constraint-probability candidate value)`

Input: A candidate
Constraint value to check against

Output: Probability that constraint value is satisfied

Description: Compute probability that candidate satisfy constraint
using T-distribution

Default: `1.0` when there are no constrained measure

`(Constraint-uncertainty candidate)`

Input: A candidate

Output: Uncertainty value (std. dev. of constrained measure)

Description: Compute uncertainty in constraint satisfaction of candidate

Default: `0` when there are no constrained measure

`(Inverse-constraint-level candidate level)`

Input: A candidate
Level of confidence desired

Output: Constraint value that can be satisfied at given level

Description: Compute the constraint which given candidate achieve at
the given level of confidence

Default: `most-negative-single-float` when there are no constrained measure

C.5 Heuristic Generation Routines

`(Generate-new-candidates candidate-pool base-list number perf-db)`

Input: Pool of candidates

A list of candidates to be used for generating new ones

Number of new candidates to be generated

Performance database of existing candidates

Output: A list of new candidates

Description: Generate a set of new candidates using past information

Example: Too complicated to be shown. See Appendix D for an example.

`(Apply-point-base candidate result perf-db)`

Input: Candidate to be modified

Latest performance result from this candidate

Performance database for existing candidates

Effect: Modified definition of the given HM

Description: Modify to improve a candidate based on past performance

Default: No action - This function should be ignored at this point

C.6 Debugging Routines

`(Output-candidate-list stream candidate-list)`

Input: Output stream

A list of candidate

Description: Print information about given candidates to output stream

`(Candidate-objective-value candidate)`

Input: A candidate

Output: (Estimated) objective value

Description: Find objective value of the given candidate

Default: `candidate-S-goodness-value` as approximated objective value

APPENDIX D - EXAMPLE IMPLEMENTATION

D.1 Implementation of AppD Functions for CRIS

Data Structure and Global Variables (var.cl, struct.cl)

Initialization and Clean-up Routines (domain.cl)

```
Load-up-domain
Reset-quantum-def
Reset-candidate-pool
Initialize-domain
Reset-performance-database
Reset-test-database
Clean-up-domain
```

Performance Evaluation Routines (perfm.cl)

```
Update-performance-database
Candidate-performance
Compute-fitness
Unacceptable-performance
```

Heuristic Generation Routines (gen.cl, unuse.cl)

```
Generate-new-candidates
Apply-point-base
```

Test-Case Management Routines (test.cl)

```
Get-test-case
Initialize-test-subdomain
```

Problem Solver Interface Routines (solver.cl, resilient.cl)

```
Evaluate-test-case
Finish-quantum?
   PM-Scan-rdb-directory, PM-Scan-perfm-rdb, PM-Write-perfm-rdb
```

Constraints Handler Routines (unuse.cl) - use default

```
Constraint-average
Constraint-distribution
Constraint-probability
Constraint-uncertainty
Inverse-constraint-level
```

Debugging Routines

```
Output-candidate-list     (monitor.cl)
Candidate-objective-value (perfm.cl)
```

Optional Routines

```
Update-candidate-list     (override.cl)
```

D.2 Extension of AppD Functions to Support Multi-Objective