A distributed architecture for an instructable problem solver

Jacky Baltes and Bruce A. MacDonald
Computer Science Department
The University of Calgary
Calgary, Alberta, Canada T2N 1N4[1]

Abstract

Our research goal is to design systems that enable humans to teach tedious, repetitive, simple tasks to a computer. We propose here a learner/problem solver architecture for such a system. The problem solving module is able to combine diverse problem solving strategies on a single problem, by using a common representation for operators, and learning operators by analyzing solution traces. At the distributed processor level, the design provides a general dynamic load balancing system that has little domain knowledge. It is controlled from the next level by a tightly constrained planner. The distributed problem solver testbed enables us to design, experiment with, and evaluate our combined learning/problem solving system for automating users' repetitive tasks.

1 Introduction: end user task automation

Computer technology continues to promise convenient task automation to end users, but often fails to deliver it. System designers can not meet all the requirements, since different users have different preferences and must perform different, sometimes new tasks. Our research goal is to design systems that enable humans to teach computers tedious, repetitive, simple tasks [15, 16], in domains such as robot assembly [7, 8, 9, 14, 17], workstation "desktops," text editing [24], and operating system commands [2, 3] including shell scripts and makefiles.

Rather than bringing end users into the programming world, it may be more appropriate to design automated computer assistants that learn under conditions familiar to the human user; emulating situations like the interaction between an office manager and a human assistant. This means that the learner must accept the normal kinds of human instructions, take better advantage of humans' natural communication ability, and exploit the natural constraints in human instruction. Forms of human instruction include: specifications, goals, formal procedures, programs, abstract plans, partial plans, concrete plans, rules, advice, comments, general hints, manually guided examples, passively observable examples, gestures, commentary, and so on.

For example, a Unix expert might teach a new task, such as backing up a selection of files, by: calling over the beginning user; stating the goal of the task; listing all the files in the current directory, telling the user to consider a particular file name that is a representative name for the required set, copying it over to the new destination, then selecting another file, and so on. Another method would be to tell the beginner exactly what steps to do, and instruct him or her to suggest steps as the task proceeds, always checking with the expert before execution. Baltes shell clerk enables a simplified interaction of this kind between user and computer. This common type of human interaction is rich in information that is both easily provided by the expert and helpful to the beginner who must determine what is important in the demonstrated task steps. The expert's actions are examples of what we refer to as human instruction, and our aim is to enable a computer-based learner to observe and understand them.

Only minimal expertise should be required of users, ideally that they can (a) perform the task themselves, and (b) teach the task to other humans. A user should not be required to undergo formal training in programming, nor in education, but should be articulate enough to show a fellow worker how to do a task. For example, to show someone how to typeset a TEX document a knowledgeable user might suggest that a novice (human or computer) user observe while the expert: invokes the LATEX system on the document; points to the particular lines in the resulting log output, that suggest the previous step be repeated (e.g., to correct

[1]Email: {baltes,bruce}@cpsc.ucalgary.ca

cross–references); invokes LaTeX again; and prints the completed typeset result. The learner would be expected to automate the task. Another example task is to build an arch from a set of blocks. The teacher shows the robot the steps to build two columns and place a lintel on top of them. During the demonstration the teacher performs some strictly unnecessary actions, which focus the learner's attention on particular important aspects of the task. For example, in Lewis' [13, 14] system the teacher might use the lintel as a "measuring stick" to mark off the distance between the columns, showing that the distance constraint is determined by the lintel.

Generally there is an implicit promise made to a teacher when a learner accepts instruction. The teacher will expect the learner to be able to perform the newly taught task under a reasonable variety of different conditions, and also to put the task to use in other, larger tasks. The teacher does not expect to have to show the "same" task more than once. So systems must be able to put their knowledge, previous learning, and the human instructions together to perform tasks autonomously, while taking advantage of implicit biases in human teachers. For example, one LaTeX demo should suffice as above, but it would be reasonable to expect a further demonstration when bibliographic citations are to be entered, and the `bibtex` system is to be invoked, or when an index is to be created.

1.1 Performing tasks and understanding natural human instruction

One part of our research is the development of a model of human instruction. It should be a constructive model, so that we will be able to build systems that understand the instructions, taking advantage of humans' implicit biases. In this paper we present only the parts of the model that are relevant to the main point of the paper, which is the design of the underlying distributed, learning, problem solver.

Lewis [13, 14] has viewed instruction as a series of communicative acts, and presents an initial model for interpreting these instructions. The human teacher's actions have two objectives: (a) to change the state of the world in order to perform a task demonstration, and also (b) to update the learner's emerging representation of the new task. For example a robot engineer might grab a robot's hand and physically move it through steps to assemble an electric motor. This creates a series of world state changes culminating in the task goal. The engineer may also intersperse actions intended to help the learner, but whose worldly effects are not required for the task goals; actions such as lining up the components in order of assembly, saying the word "red" when connecting the red wire, picking up and replacing the housing in the assembly jig to show it needs to be there, and so on. Instruction is seen as a kind of discourse, and understanding instruction as recognizing a constrained plan. To meet the general requirements our problem solver must be able to incrementally remember the teacher's instructions for later reuse in problem solving. It must be able to understand the communicative instructions a teacher does, and also understand how to solve a task that is demonstrated.

In addition, the system must not place undue cognitive load on the human user, and the learning problem solver must work within this constraint. We have begun to examine aspects of the cognitive load involved [15], and the practical constraints on the problem solver.

From a technical viewpoint, the learner is faced with a teacher who is able to do somewhat more than supply examples of the task, but somewhat less than evaluate a program if the learner were to propose one. The teacher is able to give demonstrations, and annotate these with hints, advice, and so on. Our problem solver will have a module that understands such instructions, the design of this is an important research problem, but not the topic of this paper.

The learner must do more than remember tasks and execute them on request. It must also handle a wide variety of conditions, modifying its performance of the task to match differing situations. Also, a teacher will expect the learner to remember previously taught tasks, and use them as required for subtasks in later "new" tasks. These two "implicitly promised" abilities mean that the learner must be able to carry out problem solving when it is performing tasks.

1.2 Distributed problem–solver

In our design the problem solving module smoothly combines diverse problem solving strategies on a single problem, by using a common representation for operators [4]. The system learns operators that correspond to different planning biases in the domain (e.g., macro–operators, abstraction hierarchies and case–based planning), by analyzing a trace.

The basic problem solver is already implemented on a distributed system, currently a network of transputers; in the future it may be moved to a network of UNIX workstations.

The design involves many research questions, and we have separated out the underlying general dis-

tributed planner, so it can be used as a research testbed. An object oriented paradigm mitigates the problems of engineering such a large parallel software system. At the distributed processor level, the design provides a general dynamic load balancing system that has little domain knowledge. It is controlled from the next level by a tightly constrained planner, which uses its knowledge of planning to drive the load balancer efficiently. The planner is in turn controlled by the problem solving architecture, which includes learning.

The load balancer is demand driven; objects are swapped from one processor to another only if work is requested by a neighboring processor. Estimates must be available for the expected work and for object–object communication. *A priori* estimates are in general difficult. However since our problem solver emphasizes the use of learning to improve performance on similar problems, suitable estimates can be extracted from previous problem solving episodes.

The objects at the planner level are states, operators, and partial plans. A *split–work* method distributes the work over the processors; partial plans are created by extracting nodes from the solution stack, generating new subproblem objects [20, 21].

The distributed problem solver testbed enables us to design, experiment with, and evaluate our combined learning/problem solving system for automating users' repetitive tasks.

1.3 Organization of the paper

This section has motivated our general research goal, and introduced the distributed design that this leads to. Section 2 introduces our general architecture for an instructable problem solver. Remaining sections give details of the learning/planning engine. Section 3 describes the planner design, and explains its fine grained multi–strategy design. Section 3.1 develops a uniform operator representation for multiple strategies. Section 3.2 explains and illustrates the algorithm that controls the search. Stronger biases are examined before weaker ones and operators are indexed as in a case based system. Section 3.3 discusses how operators are learned, to implement different kinds of planning biases. One emphasis of our research is that the learner explicitly takes the current problem solver state into consideration when suggesting new general operators. Section 4 describes the distributed architecture.

2 Designing a learning problem solver

Before we introduce the design of the learning problem solver, it is important to recognize the general technical problems; are the proposed learning and problem solving requirements tractable?

Briefly the learning problem can be seen as one of inducing a procedure graph from instantiated graph paths, at the same time extracting subgraphs for later re-combination. We see simple sequential procedures represented as some kind of graph (eg the AND–OR variety). In general this problem is intractable, and a good model of human instruction is essential in constraining the search for a suitable graph. Task performance is a planning problem; putting together previous tasks and primitive actions to achieve a new task. Planning new tasks is generally intractable in a rich environment, and we see the learning component as important in controlling the complexity of this. The system will be not be expected to solve new problems on its own, but will be expected to (a) solve problems that are similar to previously learned ones, and (b) solve problems that can be composed from previously learned ones.

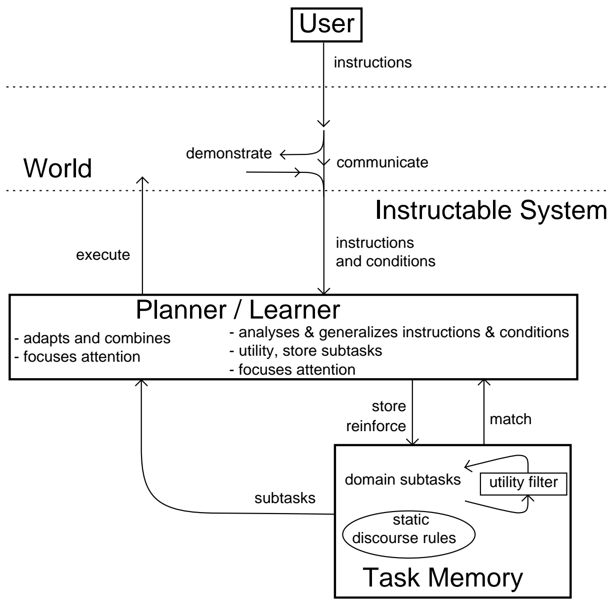
That is (cf. [**?**]) the complexity of learning will be alleviated by a helpful presentation of examples, hints, etc, plus background knowledge of human instruction. The problem solving complexity will be alleviated by caching previous learning, in a similar way to case–based techniques.

2.1 Learning problem solver architecture

During instruction the system interprets the human user's input, extracts what it considers to be useful components of the task, including possibly the complete task, and stores these for later reuse. When a new goal is given, similar tasks will be retrieved from the remembered past experience, and put together to solve the current problem in the current situation.

The general architecture we have developed is shown in Figure 1. It represents our current intentions for a learning problem solver, given our emphasis on understanding human instruction for automating repetitive tasks. The teacher demonstrates the task by performing it manually, while the learning system watches and records (a) the task actions, (b) the changing environmental conditions as each action is executed, and (c) the instructions the teacher gives to augment the actions for performing the task. The learner uses its existing knowledge of tasks and its model of teacher interaction (the static discourse model [13, 14]) to interpret this information and store

Figure 1: The general architecture of our learning problem solver



seemingly useful tasks and subtasks (general operators) in its memory. When a task must be executed the specified goal evokes similar tasks from the learner's memory, and these are adapted and combined to attempt to solve the problem. This performance is also analyzed and any seemingly useful new subtasks stored. As subtasks are repeatedly used, the memory module attempts to filter out ones that do not seem useful. At each stage of learning and performance, focusing rules[2] limit the objects that are considered, to help manage the complexity of the learning and problem solving.

3 A Multi–strategy Learner/Problem Solver

This section presents our design for a learning problem solver, including the operator representation, the search control algorithm, and the acquisition of new operators for various planning biases. As mentioned previously, problem solving is intractable in rich domains. Different methods have been proposed to limit the search space, such as means–ends analysis [19], non–linear planning [23, 25], abstraction hierarchies [11, 22], case–based planning [6], reactive planning [1], and many more. These methods correspond to *planning biases*, that is assumptions about some aspect of the problem, for example the domain, the distribution of problems, or the structure of plans. These assumptions allow the problem solver to reduce the search space. For example, non–linear planning assumes that permutations of operators in a sequence result in equivalent states and that the cost of avoiding commitment to a given ordering of operators is less than having to backtrack, should the planner choose an incorrect order. Only if these assumptions are met in a domain, may the planner be successful, otherwise it will fail or perform poorly.

[2]Rosanna Heise is working on the focusing system.

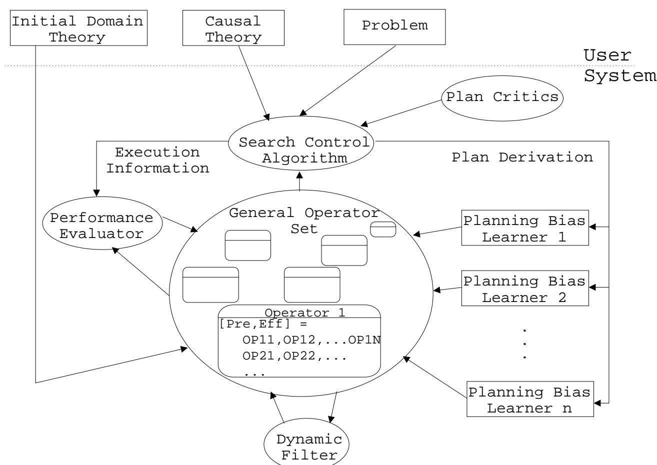
Combining a problem solver with a learning module can alleviate some of the complexity of problem solving [12, 18, 26], however, the system will still be limited by the underlying problem solver's biases. Should a planning bias be inappropriate, then the learning system may be unable to overcome these limitations. Therefore, most learner/problem solver systems use a *weak* planner, that is a planner that makes few assumptions about the domain. For example, systems that learn macro–operators usually are based on problem solvers that use depth–first or best–first search algorithms. However, by doing this, these systems give up those *planning biases* that have proven useful in many domains. Our research takes a different approach. Rather than giving up useful planning biases such as macros, abstractions, and case–based planning, we are developing a multi–strategy planning system that can combine these biases on a single problem. The idea is to design a problem solver that can dynamically adjust its bias to improve its performance.

To achieve this goal, we require:

- A common representation for different planning biases.
- A search control strategy that can take advantage of different planning biases.
- Planning bias learners that can recognize when certain biases are appropriate and that can update the common representation so that this bias is used by the search algorithm.

The architecture of the learner/problem solver is shown in figure 2. The user provides an initial domain theory and a causal theory of the domain. The initial domain theory consists of a set of operators describing all primitive actions that an agent can perform in the domain. This set initializes the general operator memory. The search control algorithm is described in section 3.2. It uses the set of general operators and plan critics to find a solution to the problem. Once the problem solver finds or fails to find a solution, the

Figure 2: Learner/Problem Solver Architecture



derivation is passed on to a set of planning bias learners (see section 3.3). The planning bias learners use surface or operationalized abstract features to identity parts of the search that could have been improved using a given bias, and create new general operators that implement the given bias. Execution information is passed on to a performance evaluator that updates information used by the dynamic load balancing algorithm, such as the average amount of work for a given general operator. The system uses a dynamic filter to remove general operators that do not prove to be useful any more. This can happen, for example, if the user's long term activities change from one set of tasks to another. This shift may invalidate previously appropriate biases.

Primitive operators consist of a list of pre–conditions and a list of effects and can contain formal parameters. To overcome the frame problem, the learner/problem solver makes the STRIPS assumption [5]. The effects of an operator consist of an add and a delete list, which specify predicates that are added or removed from the state description during execution of the operator. Predicates not mentioned in the effect lists remain unchanged during execution. An example operator definition is shown in table 1.

The STRIPS assumption makes it impossible to describe conditional effects of operators, such as an operator for flipping a light switch, which can not be modeled using add and delete lists alone, since the change depends on the initial switch state. The light will come on if it was off before and vice versa. This can complicate the specification of a domain. There-

Table 1: Example: Operator to move a robot from position 1 to 2

```
Operator:      Move-Robot($P1,$P2)
Preconditions: Robot-at($P1) AND
               Free-path($P1,$P2)
Effects:
   Delete:     Robot-at($P1)
   Add:        Robot-at($P2)
```

fore, the user can provide a causal theory to model conditional effects if so required.

The user can start the problem solving process by presenting the system with an example problem, which consists of an initial state and a goal state. The task is to generate a sequence of operators plus variable instantiations, that will transform the initial state into the goal state.

3.1 Representation of General Operators

The search space of the problem solver, the state space, is implicitly defined by the set of operators. An explicit representation of the state space, though more powerful, is impractical because of its size. To be able to combine different problem solving strategies, the operator set must be powerful enough to represent different biases. However, to enable dynamic and flexible combination of different biases, there must be a single underlying representation method. The learner/problem solver uses a generalized version of the standard operator representation, so called general operators.

Figure 3 describes the general operator schema. The most important information associated with an operator is its pre–conditions and effects. Pre–conditions and effects may be incomplete specifications, that is, they can be at different levels of abstraction. This type of abstraction is similar to ABSTRIPS [22] and ALPINE [11], which also delete predicates from pre–conditions and effects to produce an abstraction hierarchy. Additionally, a set of refinements is associated with each operator. A refinement is a sequence of operators, that was previously successful in achieving the top–level effects, given the top–level pre–conditions. However, the pre–conditions and effects of a general operator may be more abstract than those of its refinements. All that is required is that the pre–conditions and effects of refinements subsume those of the associated operator. In [4], we show that this representation is powerful enough to represent a large variety of different planning biases such as

Figure 3: General Operator Representation

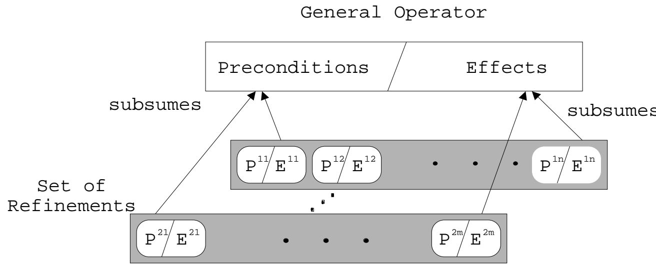


Table 2: General Operator Example

```
General Operator: Move-Medium-Disk($P1,$P2)
Preconditions: On-Medium($P1)
Effects: On-Medium($P2)
Refinements:
 1: Move-Medium($P1,$P2)
 2: Move-Small($P1,$P3),Move-Medium($P1,$P2)
```

macro–operators, abstractions, cases, reactive rules, subgoaling, goal regression, and many more.

A simple example of a general operator is shown in table 2. It is an abstract operator to move the medium disk in the towers of Hanoi problem.[3] The first refinement is for the case in which the small disk is already on the medium peg, the operator sequence simply moves the medium disk. In the second case, the small and the medium disk are on the same peg. The refinement first moves the small disk out of the way, and then moves the medium disk. Note that this set will not always be complete. In this example, the refinements do not cover the case in which the small disk blocks the target peg. When this case occurs the planner will construct it.

3.2 The Search Control Algorithm

The search control algorithm uses the general operator set to find an operator sequence that transforms the initial state into the goal state. The search algorithm first tries to use *stronger* biases, that is biases that restrict the search space more. If a bias fails, the search space is extended and a *weaker* bias is used. Thus, the system will gracefully degrade to an exhaustive search procedure in the worst case. For example, if a solution is not found the search algorithm will move from using instantiated cases to macros, then to abstraction hierarchies.

[3]This example has been tested on Baltes' prototype problem solver [4].

Given an initial state and goal, the search control method tries to find a general operator that solves the specific problem. If the general operator found is an instantiated case, the search algorithm will use plan critics to adapt the case to the new situation. Plan critics are generalizations of those proposed by Sacerdoti [23]. A plan critic is a local repair method to make a plan valid in the new situation, for example `substitute-variable`. They use abstraction hierarchies created by the planning bias learners to guide the possible adaptations of a plan. For example, if the part of the original plan that fails is part of a refinement of some abstract operator, the plan critic will try to use another refinement. If the general operator found is an abstraction, the algorithm tries to create a refinement that fits the current situation. If no general operator can be found, the search algorithm tries to generate a plan by combining general operators.

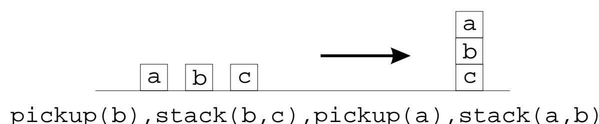
We will illustrate the search control with an example from a slightly extended version of Nilsson's blocksworld [?]. The example shows how case–based techniques, abstractions, and macros are combined.

There are four primitive operators in the domain, shown in table 3. Suppose the problem solver is given the problem of constructing a stack of three blocks, shown in figure 4. This problem is solved using means–ends analysis, since no learning has taken place so far. The resulting plan is shown in figure 4 and is added to the general operator set as a case, see table 4. Suppose the system is then asked to solve the problem in figure 5. Before `y` can be stacked on `z`, `x` has to be taken off. The solution to the second problem is passed on to a set planning bias learners. An analysis of the solution shows that the operator sequence has identical post–conditions, but different pre–conditions than the standard `pickup(x),stack(x,y)` sequence. One of the planning bias learners creates the abstract general operator in table 5 under these conditions, and adds it to the operator set. This abstract operator can stack two blocks together independently of whether the first block is clear or not. If the first block is clear it is picked up and stacked on the target block. Otherwise the second refinement clears it first.

After this preliminary training, the problem solver is given a new blocksworld problem, shown in figure 6. The task is to construct a tower. There are now six operators; four primitive ones and two learned general ones.

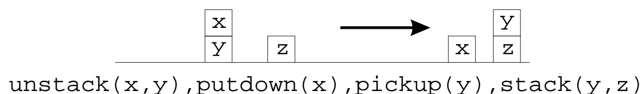
Using the "strongest bias first" strategy, the problem solver retrieves the first problem as the most sim-

Figure 4: First blocksworld problem



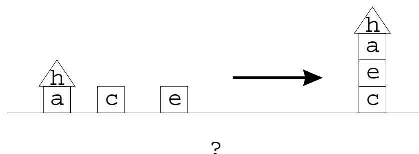
pickup(b),stack(b,c),pickup(a),stack(a,b)

Figure 5: Second blocksworld problem



unstack(x,y),putdown(x),pickup(y),stack(y,z)

Figure 6: New blocksworld problem



?

ilar case and tries to execute it. This will fail for the first operator, since there is no block `b` in the new problem. The problem solver tries to use different adaptation methods to make the old case fit the new situation. One of the adaptation methods is *substitute–variable*, which replaces a variable instantiation by a different one. As shown in figure 7, by replacing block `b` by block `e` in the case, the first part of this problem can be solved.

In the next step, the plan fails because `pickup(a)` can not be executed, since `a` is not clear. However, the problem solver indexes the refinement hierarchy and finds that the operator sequence `pickup(a),stack(a,e)` is a refinement of `abstract-stack`. This refinement is replaced by a different refinement of the abstract operator. Using the second refinement, the second part of the problem can be solved, as shown in figure 8.

In the third step, the system generates a subproblem recursively to complete the plan, since the previous case is now fully adapted to the new situation. This remaining subproblem is easily solved and the plan completed, see figure 9.

Table 3: Operators for the Blocksworld

Pickup($X)	the hand picks up block $X from the table
Putdown($X)	put block $X on the table
Stack($X,$Y)	put block $X on block $Y
Unstack($X,$Y)	take block $X off block $Y

Table 4: Learned operator for first problem

```
General Operator: Case1(a,b,c)
Preconditions: ...
Effects: Solves first problem
Refinements:
   1: Pickup(b),Stack(b,c),Pickup(a),Stack(a,b)
```

Table 5: Abstract operator created by the second problem

```
General Operator: Abstract-Stack($X,$Y)
Preconditions: ...
Effects: $X on block $Y
Refinements:
   1: Pickup($X),Stack($X,$Y)
   2: Unstack($Z),Putdown($Z),Pickup($X),
      Stack($X,$Y)
```

3.3 The Planning Bias Learners

Since the search space is implicitly defined by the general operator set, the performance of the system is critically dependent on the general operator set. This set must be updated to improve future performance. After a problem solving episode, the system will pass the derivation to a set of planning bias learners. The task of these learners is to (a) find instances in which a planning bias can be useful, (b) create a general operator to implement this bias. The problem is that adding an operator will in general (a) increase the branching factor (adding more possibilities for achieving a goal), and (b) increase the matching cost of the domain (if will be more expensive to test whether an operator is applicable). Therefore, the generation of new operators must be carefully controlled. This learning task is in general intractable. Therefore, the type of general operators that are created must be restricted.

Most learning problem solving systems adapt independently of the current state of the planning system. For example, systems that learn macro–operators such as Iba's MacLearn [10] use some heuristic (e.g., peak to peak heuristic) to create new operators, and this is

Figure 7: Step 1: Case Adaptation

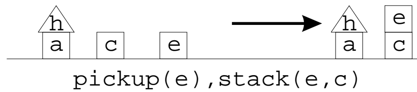


Figure 8: Step 2: Replace refinement

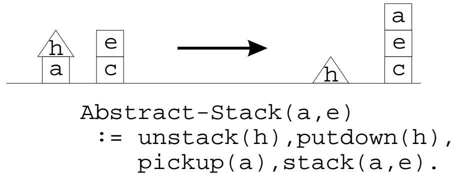
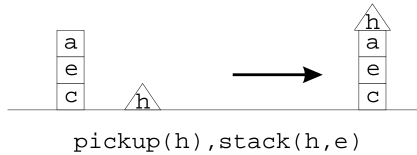


Figure 9: Step 3: Recursively generated blocksworld problem



independent of the current operator set. Our research emphasizes that the learner consider the current state of the system when creating new operators. For example, Baltes' implementation [4] of one learning module creates abstractions by comparing parts of a successful plan to other operators in the general operator set and extracting those that can be used to build abstraction hierarchies. Although only a prototype system, it learned an abstraction hierarchy (equivalent to that learned by ALPINE [11]) that reduced its time complexity from exponential to linear in the length of the solution.

4 Distributed Implementation of the Learner/Problem Solver

This section explains the distributed learner/problem solver created by Baltes. There are two reasons for a distributed implementation. First, to provide cheap processing power. Second, to distribute the workload and free the user's workstation for doing manual tasks, so that the learner/problem solver is not intrusive.

Currently, the implementation platform of the learner/problem solver is a network of transputers connected to a PC–compatible host running LINUX.[4] Transputers are specifically designed for parallel architectures based on the CSP model of concurrency. In the CSP model, developed by Hoare, processes run asynchronously and can communicate via synchronized channels. Transputers provide four high speed serial interfaces for creating networks, and support parallel execution in the instruction set. This provides a cost–effective development environment. We plan to port the system to a network of UNIX workstations.

Engineering a large distributed software system is challenging in itself. To overcome these problems, the design used an object oriented version of the CSP model of concurrency. Each object is associated with one process; processes run asynchronously. Therefore, objects can handle only one message at a time. An object can send synchronous or asynchronous messages to other objects. A synchronous message will halt the calling process until it receives an answer from the receiver; asynchronous messages allow the sender to continue.

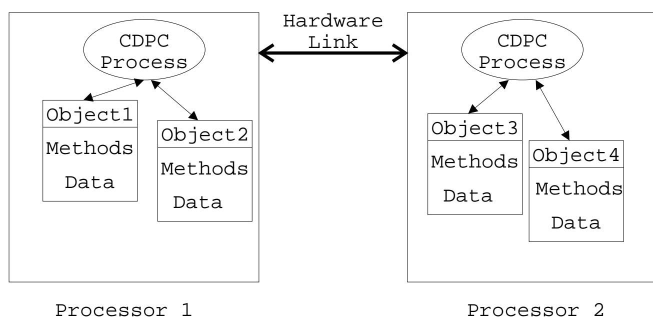
The load distribution scheme must assign processes to processors. Since search spaces in planning problems are highly irregular structures, an *a priori* work distribution is infeasible. Instead, a dynamic load balancing system distributes work at run time by swapping objects to other processors. The design of the dynamic load balancer has been separated out as an important research area. It is similar to the overall problem solver design, since the goal was to provide a general weak low level architecture that is tightly controlled by higher levels, in this case the planner.

Each processor runs a special process, the central dispatch controller or CDPC, which distributes messages to objects that do not reside on the local processor and collects statistics about the performance of the system. Figure 10 is an example of the object oriented parallel architecture. Since all messages are distributed by the central dispatch controller, an object can easily be swapped to a different processor. The calling objects do not need to know the location of the destination object.

The load balancer is demand driven, that is work is swapped out to other processors only if they request work. This improves the error tolerance of the system. Should a processor break down, it will not disable the whole system, the processor will simply stop requesting work.

[4] A PD Unix Operating system, copyright Linus Torvalds.

Figure 10: Object Oriented Parallel Architecture



Dynamic load balancing requires estimates of the work for an object and the object–object communication. *A priori* estimates are in general difficult to obtain. However, since our problem solver emphasizes learning to improve performance on similar problems in the future, suitable estimates can be extracted from previous problem solving episodes. The implementation focuses on parallelization of the search, since this is the most time consuming part of the problem solving process. As shown in figure 2, the estimates are updated by the performance evaluator.

The goal of learning is to reduce the number of reasoning steps required to find a solution. For example, macro–operators reduce the number of operators that have to be added to a plan. Case–based reasoning provides a base case and assumes that adapting this takes fewer steps than generating the plan from scratch. Parallelization supports learning, therefore, by exploring different alternatives in parallel. Rather than assigning a large number of processes to a single general operator refinement, they are assigned to different ones. This allocation scheme has a number of advantages: (a) different refinements require mostly local computation and are therefore relatively independent. (b) a refinement will have a reasonable amount of work associated with it.

The planner level of the system uses state, operator, and partial plan objects. To distribute the work, partial plan objects provide a *split–work* method that can break a partial plan up into two parts. The two parts have roughly an average amount of work left in completing the plan. For example, if the learner problem solver is trying to complete a partial plan that contains an abstract operator with two refinements, the *split–work* method will create two new partial plans using different refinements of the abstract operator.

This section summarizes our distributed problem solver architecture, which is based on an object oriented version of the CSP model. It uses a dynamic load balancing level that is controlled by the planning level, which in turn creates subproblems in a breadth–first manner to take advantage of the learning component.

5 Conclusion

Our design for a learning problem solver provides an engine for a system that understands human instructions and empowers end users with automation. The problem solver provides a uniform, fine–grained, flexible, multi–strategy engine for solving problems based on analyzed past experience, gracefully moving from a case–based approach in a piecemeal fashion to a more expensive search when the new task differs more from previous ones. Various planning biases are learned as new operators, by a set of learning modules. The system architecture is a distributed one, providing semi–remote use of computing resources and effective use of current and expected future technology. It is implemented on a transputer system at present. An object oriented design is used, with one object per process, and is based on the CSP model of concurrency. A tailored dynamic load balancer distributes the work. A dispatch process on each processor controls the distribution of non–local messages. Dynamic estimates of the subtask effort are used in later allocation of processor resources for similar subtasks.

The distributed problem solver testbed enables us to design, experiment with, and evaluate our combined learning/problem solving system for automating users' repetitive tasks.

Acknowledgements

This work is support by the Natural Sciences and Engineering Research Council of Canada, the Alberta Microelectronic Centre, and the University of Calgary.

References

[1] P. Agre and D. Chapman. An implementation of a theory of activity. In *Proceedings AAAI–87*, 1987.

[2] Jacky Baltes. A symmetric version space algorithm for learning disjunctive string concepts. In *Proc. Fourth UNB Artificial Intelligence Symposium*, pages 55–65, Fredericton, New Brunswick, September 20–1 1991.

[3] Jacky Baltes and Bruce A. MacDonald. Case–based meta learning: sustained learning supported by a dynamically biased version space. In *Proceedings of the Workshop on Biases in Inductive Learning, at the International Machine Learning Conference*, Aberdeen, Scotland, July 4 1992.

[4] Jacky Baltes and Bruce A. MacDonald. An integrated planning representation using macros, abstractions, and cases. In *Proceedings of 3rd Workshop on Problem Reformulation and Representation Change*, pages 1–9, Asilomar Conference Center, Pacific Grove, CA, April 29 – May 1 1992. NASA Ames Research Center. Technical Report FIA–92–06.

[5] R. Fikes, P. Hart, and N. Nilson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[6] Kristian J. Hammond. *Case Based Planning*. Academic Press Inc., 1989.

[7] Rosanna Heise. Demonstration instead of programming: focussing attention in robot task acquisition. Master's thesis, Department of Computer Science, University of Calgary, 1989.

[8] Rosanna Heise. Programming robots by example. *Journal of Intelligent Systems*, in press.

[9] Rosanna Heise and Bruce A. MacDonald. Robot program construction from examples. In *Proc. National Irish AI Conf.*, Dublin, Ireland, September 1989. Also in book form, edited by A. F. Smeaton and G. McDermott (Eds.), AI and Cognitive Sciences '89, Springer–Verlag, 1990.

[10] G. A. Iba. A heuristic approach to the discovery of macro–operators. *Machine Learning*, 3(4):285–318, 1989.

[11] Craig A. Knoblock. *Automatically Generating Abstractions for Problem Solving*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1991. Tech. Report CMU-CS-91-120.

[12] R. E. Korf. Macro–operators: A weak method for learning. *Artificial Intelligence*, 26(1):35–77, 1985.

[13] John D. Lewis. A computational model of task acquisition from instruction. Master's thesis, Department of Computer Science, University of Calgary, 1992.

[14] John D. Lewis and Bruce A. MacDonald. Machine learning under felicity conditions: exploiting pedagogical behavior. In *Proceedings of AI / CS '92*, Limerick, Ireland, September 1992. Also presented at the AAAI Workshop on Constraining Learning with Prior Knowledge, July 1992, San Jose, CA.

[15] Bruce A. MacDonald. Instructable systems. *Knowledge Acquisition*, December 1991. (to appear) Accepted by *Knowledge Acquisition* without revision. A much shorter version was presented at the 1990 Banff Knowledge Acquisition Workshop; see conference papers.

[16] Bruce A. MacDonald and Jacky Baltes. Research in instructable systems. In *Proceedings of the Machine Learning Workshop at the Ninth Canadian Artificial Intelligence Conference*. Canadian Society for the Computational Studies of Intelligence, May 1992.

[17] Bruce A. MacDonald and David Pauli. Adaptive robot training by programming and guiding. 1991. Submitted to *Journal of Intelligent Manufacturing*.

[18] Steven Minton. *Learning Search Control Knowledge, An Explanation–Based Approach*. Kluwer Academic Publishers, 1988.

[19] Allen Newell and Herbert A. Simon. *Human Problem Solving*. Prentice-Hall, Englewood Cliffs,N.J., 1972.

[20] V. Nageshwara Rao and Vipin Kumar. Parallel depth first search. part I. implementation. *International Journal of Parallel Programming*, 16(6):479–500, 1987.

[21] V. Nageshwara Rao and Vipin Kumar. Parallel depth first search. part II. analysis. *International Journal of Parallel Programming*, 16(6):501–532, 1987.

[22] Earl D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5:115–135, 1974.

[23] Earl D. Sacerdoti. *A structure for plans and behavior*. American Elsevier, 1977.

[24] Natascha O. Schuler. L-EBE: Learning iterative editing by example. Master's thesis, Department of Computer Science, University of Calgary, 1992.

[25] Stefik. *Planning with constraints*. PhD thesis, Computer Science Dep.,Stanford University, 1980. Rep. No. 80-784.

[26] Iba W. Wogulis and P. Langley. Trading off simplicity and coverage in incremental learning. In *Proceedings of the 5th International Conference on Machine Learning*, pages 73–79. Ann Arbor, 1988.