Learning about Constraints by Reflection

J. William Murdock and Ashok K. Goel

Georgia Institute of Technology
College of Computing
Atlanta, GA 30332-0280
murdock,goel@cc.gatech.edu

Abstract. A system's constraints characterizes what that system can do. However, a dynamic environment may require that a system alter its constraints. If feedback about a specific situation is available, a system may be able to adapt by reflecting on its own reasoning processes. Such reflection may be guided not only by explicit representation of the system's constraints but also by explicit representation of the functional role that those constraints play in the reasoning process. We present an operational computer program, SIRRINE2 which uses functional models of a system to reason about traits such as system constraints. We further describe an experiment with SIRRINE2 in the domain of meeting scheduling.

1 Introduction

All systems have constraints: restrictions on what things that system does. However, a dynamic environment may place demands on a system for which its constraints are not adequate. One potential use of machine learning is the modification of the constraints of a computer-based system to meet new requirements. It is often necessary (or at least useful) for a machine learning system to possess explicit representations of the concepts which it is learning about. Thus the goal of learning new constraints raises the question: how can a system represent and reason about its own constraints?

Consider, for example, the task of scheduling a weekly meeting among a group of users with fixed schedules. A system which performs this task is likely to have a wide variety of constraints, e.g., the set of times during which a meeting might be scheduled. If, however, a meeting scheduler has restrictions on the times that it can schedule meetings and those restrictions turn out to be invalid, it will have to modify those constraints.

One way that a system can determine that its constraints are inadequate is through feedback which is specific to a particular situation. A human user might not want to abstractly specify general constraints for all situations. Indeed such a user might not be able to completely define the constraints for a complex system. However, if a user can provide acceptable results for specific problems in which the meeting scheduler fails, it should be possible for that meeting scheduler to apply machine learning techniques to adapt to the desired functionality.

E. Stroulia and S. Matwin (Eds.): AI 2001, LNAI 2056, pp. 131–140, 2001.

If an agent has a model of its own reasoning process, it may be possible to include constraints within that model, and thus represent not only what the constraint values are but also *what functional role they play* in the agent's reasoning. Under these circumstances, model-based diagnosis may be used to identify which constraints are having what effect on a specific output; the combination of this information with situation-specific feedback can thus enable adjustment of the constraints.

The SIRRINE2 system is an agent architecture for implementing agents with self-knowledge. The language for specifying agents in SIRRINE2 is the Task-Method-Knowledge (TMK) language which provides functional descriptions of reasoning processes. One aspect of these descriptions can be the explicit representation of agent's constraints and the functional role they play in the agent's behavior.

2 TMK Models

Agents in SIRRINE2 are modeled using the Task-Method-Knowledge (TMK) language. Variants and predecessors of this language have been used in a number of existing research projects such as AUTOGNOSTIC [5], TORQUE [3], etc. A TMK model in SIRRINE2 is directly accessible to the evolutionary reasoning mechanism as declarative knowledge about the agent's processing. The work presented here extends the range of applications of TMK by focusing on its usefulness for the addition of new capabilities, rather than, for example, correcting failures in the original design, as is done in AUTOGNOSTIC.

In order to use a TMK model, the user also provides an initial *knowledge state* which the agent is to operate under (for example, a meeting scheduler might be provided with a list of schedules for the people in the meeting). During the execution of an agent, a *trace* of that execution is recorded. The trace and the model are both used by the portions of SIRRINE2 which perform evolution. These evolutionary reasoning mechanisms generate additional, intermediate knowledge. A particularly significant variety of intermediate knowledge is a *localization*, i.e., an identification of a potential candidate for modification by the system.

Processes in TMK are divided into *tasks* and *methods*. A task is a unit of computation which produces a specified result. A description of a task answers the question: *what* does this piece of computation do? A method is a unit of computation which produces a result in a specified manner. A description of a method answers the question: *how* does this piece of computation work? Task descriptions encode functional knowledge; the production of the specified result is the function of a computation. The representation of tasks in TMK includes all of the following information: :input and :output slots, which are lists of concepts which go in and out of the task; :given and :makes slots, which are logical expressions which must be true before and after the task executes; and some additional slots which refer to how that task is accomplished. The reference to how the task is accomplished may involve a simple table or piece of executable source code (in which case the task is said to be a *primitive task*) or it may involve a list of methods which accomplish the task.

Method descriptions encode the mechanism whereby a result is obtained. This mechanism is encoded as a collection of states and transitions, and the states refer to lower-level tasks which contribute to the effect of the method. Each non-primitive task is associated with a set of methods, any of which can potentially accomplish it under certain circumstances. Each method has a set of subtasks which combine to form the operation of the method as a whole. These subtasks, in turn, may have methods which accomplish them, and those methods may have further subtasks, etc. At the bottom level are the primitive tasks, which are not decomposed any further.

Descriptions of knowledge in TMK is done through the specification of *domain concepts*, i.e., kinds of knowledge and *task concepts*, i.e., elements of knowledge. For example, a domain concept in the domain of meeting schedulers would be a time slot. Two task concepts for this domain concept might be the time slot being considered for a meeting and a time slot in which some person is busy. TMK represents abstract knowledge about the kinds of constraints that exist in a domain as domain concepts. The connection of a particular set of constraints to a particular agent is then represented by a task concept.

In addition to the task concepts and domain concepts, modeling of knowledge in TMK includes information about relationships between knowledge elements. *Domain relations* are defined over domain concepts and abstractly describe the kinds of relationships that may exist over concepts in the domain. An example of a domain relation would be one that indicates that two time slots overlap. *Task relations* are defined over tasks concepts and involve a specific instantiation of a domain relation over some specific task concepts.

3 Evolution Algorithm

Below is a high-level overview of the algorithm which SIRRINE2 uses in the execution of an evolving agent.

```
function execute-agent(TMK-Model start-tmk, Knowledge-State start-ks)
    Trace tr
    Knowledge-State end-ks
    Knowledge-State desired-ks
    List of Localizations list-loc
    TMK-Model new-tmk
    (end-ks, tr) = execute-task(start-tmk, start-ks)
    If trace-outcome(tr) == success
      Return (end-ks, tr)
    desired-ks = acquire-feedback()
    list-loc = assign-credit(tr, start-tmk, end-ks, desired-ks)
    While list-loc != ()
      new-tmk = modify(start-tmk, first(list-loc))
      If new-tmk != start-tmk
        (end-ks, tr) = execute-agent(new-tmk, start-ks)
        Return (end-ks, tr)
      Else
        list-loc = rest(list-loc)
    Return (failure, tr)
```

The process begins with an initial TMK model and a starting knowledge state. For example, the execution of a meeting scheduling agent might begin with a description of that agent and a knowledge state which indicates a list of schedules to be considered. The primary task of the TMK model is then executed (this involves recursively selecting and invoking a method for that task, decomposing that method into subtasks, and then executing those subtasks). The task execution process returns both a trace and a resulting knowledge state (for example, a successful execution of a meeting scheduling agent might result in knowledge of a time slot for a meeting to be held). If the resulting trace indicates that the task has been successfully completed, the agent execution is successful. However, if the task was unsuccessful, some evolution of the agent is needed. At this point, the credit assignment process is used to identify possible causes for the agent's failure. The system then steps through the identified localizations one at a time until one of them allows the modification process to make a change to the TMK model. The modified TMK model thus describes a new, evolved agent. The original problem is then attempted again using this modified agent.

4 The Meeting Scheduler

One of the agents which have been modeled in SIRRINE2 is a relatively simple meeting scheduling system. The problem addressed by this meeting scheduler is finding a time slot for a weekly meeting of a given length for a group of people each with a given weekly schedule. For example, if three people want to meet, and one of them is always busy in the morning, and one is busy all day on Mondays, Wednesdays, and Fridays, and another is busy all day on Tuesday, the meeting scheduling agent will decide that the meetings should be held on Thursday afternoons.

Figure 1 presents the tasks and methods for the model of the meeting scheduling agent. The top level task of the agent is the task of scheduling a meeting. It has one method which it uses, that of enumerating a set of slots and checking those slots against the schedules. The slot-enumeration method sets up three subtasks: finding a first slot to try, checking that slot against the list of schedules, and finding a next slot to try. It also defines a set of transitions which order these subtasks; in particular, it starts with the find-first-slot subtask and then loops between the check-slot-schedules and find-next-slot until either the the check-slot-schedules task succeeds, (i.e., a slot has been found which satisfies all of the schedules) and thus the method has been successful or the find-next-slot task fails (i.e., there are no more slots to consider) and thus the method has been unsuccessful. Both the find-first-slot and find-next-slot tasks are primitive, i.e., they are directly implemented by simple procedures. The check-slot-schedules task, however, is implemented by the schedule-enumeration method. This method steps through each individual schedule in the list of schedules and checks the proposed slot against each of them until a conflict is determined or all of the schedules have been checked.

In addition to the tasks and methods illustrated in Figure 1, the TMK model of the meeting scheduler also contains explicit representations of the knowledge

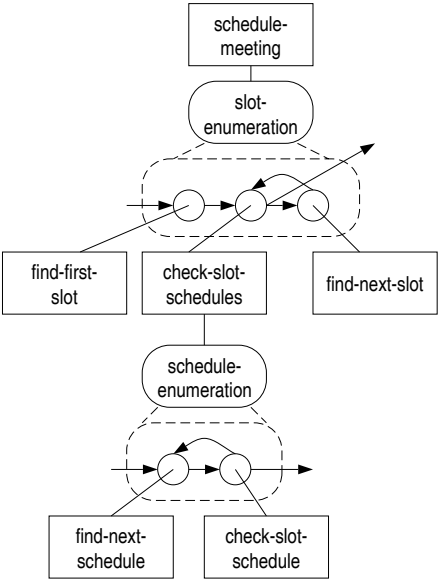

Fig. 1. The tasks and methods of a simple meeting scheduling agent. Rectangular boxes represent tasks; round boxes represent methods. The circle-and-arrow diagrams within the round dotted boxes represent the control portion of the methods, i.e., the transitions and their references to the subtasks.

contained by the meeting scheduler. The meeting scheduling agent contains eight domain concepts:

- length: A length of time, represented as a number of minutes
- day: A day of the week, e.g. Thursday
- time-of-Day: A time of the day, e.g. 2:00 PM
- time: A moment in time, represented by a combination of the day and time-of-day domain concepts, e.g., Thursday at 2:00 PM
- time-slot: An interval in time, containing two times: a start time and an end time, e.g. Thursday at 2:00 PM until Thursday at 3:00 PM
- schedule: A list of time-slots indicating when an individual is busy
- schedule-list: A list of schedules
- time-constraints: A list of time-slots indicating when meetings can be held, typically Monday through Friday from 9:00 AM to 5:00 PM

A description of a task in TMK explicitly refers to task concepts which describe the task's inputs and outputs. These task concepts explicitly refer to the domain concept of which the designated input or output should be an instance. For example, the schedule-meeting task has as its output a time slot for which a meeting should be held; this meeting time slot is a task concept which refers to the general time-slot domain concept.

The time-constraints domain concept is an example of an explicit representation of an agent's constraints. The meeting scheduler has one task concept for the

time-constraints domain concept. This task concept, slot-generation-constraints, acts as an input to both the find-first-slot and find-next-slot tasks, constraining what kind of time slots these tasks can generate. The slot-generation-constraints task concept illustrates the crucial role that task concepts play in the integration of TMK models; it serves as the link between the find-first-slot and find-next-slot tasks which are influenced by the constraints, and the time-constraints domain concept which models the constraints.

5 Experiment

We describe here a brief experiment which illustrates the behavior of SIRRINE2 in a constraint evolution problem. Some time ago, our research group was faced with the problem of scheduling a weekly meeting to discuss technical issues. Our goal was to hold a 90 minute meeting. One member of the group sent out email asking what times people would be available for such a meeting. The members of the group sent back a list of times during which they were busy.

In order to conduct an experiment with our meeting scheduling agent, we decided to run the system on this data. The schedules were typed into the meeting scheduler. The scheduler considered a long sequence of 90 minute time slots, checking each one against the schedules of the people in the group. Every slot that it considered conflicted with at least one schedule; consequently, the meeting scheduler failed to generate a time for the meeting. Ultimately, it was decided (by the head of the group) that the meeting would be held on Tuesdays from 4:30 to 6:00 PM, i.e., the assumption that meetings must be held during standard business hours was violated. At this time, we provided SIRRINE2 feedback informing it of the time that was selected. At this point, the system was able to do some self-adaptation so that it would have generated this answer in the first place.

There are many possible changes that can be made which would have lead to this result. The meeting scheduler could be changed to always schedule meetings on Tuesdays 4:30 to 6:00 PM. Alternatively, it could be made to schedule meetings on Tuesdays 4:30 to 6:00 PM only when it receives exactly the same set of schedules as it did in the experiment and to simply use its existing mechanisms in all other cases. A reasonable compromise would be to allow meetings to generally be scheduled until 6:00 PM. However, the current model of the meeting scheduling domain does not provide sufficient information to identify a reasonable compromise. Consequently, a simpler change was made in which the Tuesdays 4:30 to 6:00 slot is suggested whenever a 90 minute time slot is requested and no other time is available.

During this experiment, the meeting scheduler went through the following stages:

Execution: The meeting scheduler runs and attempts to solve the problem as specified. During execution, a trace of reasoning is generated. For this problem, the agent fails, because it is unable to find a slot which fits into all of the schedules indicated.

Feedback Acquisition: Information is provided by the user which indicates that the slot which should have been selected is Tuesdays from 4:30 to 6:00. This fact is provided in the form of a desired output knowledge state.

Credit Assignment: If the meeting scheduler fails, as it does in the example, SIRRINE2 attempts to identify a particular procedural element (task or method) which may be responsible for the failure. This process returns a list of possible localizations of failure since there may be many possible causes for the observed result. The element of particular interest in this example is the find-next-slot task. This task should have produced a time slot but, in the final iteration of the system, didn't; because SIRRINE2 was given a time slot as feedback, it posits that the failure of the system may be a result of this task not producing that time slot. Note that this step is the one which critically depends on the presence of explicit constraint knowledge; the assignment of credit for the find-next-slot uses the :input slot of that task to determine that the slot-generation-constraints task concept influences its results.

Modification: Given a particular failure localization, the modification mechanism is intended to make a change to the system which corrects the identified fault. Figure 2 illustrates the revised meeting scheduler. The primitive find-next-slot task is modified to be a non-primitive task with two methods: find-next-slot-base-method simply invokes the existing primitive for finding slots, and find-next-slot-alternate-method always produces the produces the Tuesdays from 4:30 to 6:00 slot. The :given slot for the alternate method indicates that it is to be run if and only if a 90 minute time slot is requested under knowledge conditions similar to the ones found here. Similarity, in this situation, is defined by the :input slot of the task, i.e., the alternate method is chosen whenever the values of the task concepts specified in the :input slot exactly match their values in the earlier execution. The redesign strategy presented here is one example of a way to expand upon a set of constraints: by providing an alternative functional element with different constraints and establishing appropriate conditions for selecting that elements, the overall constraints of the model are expanded.

Execution: Finally, the meeting scheduler is run again with the same set of schedules. During this execution, the original find-next-slot primitive is run repeatedly until no other time is available and then the new primitive is run. Thus the scheduler correctly finds the Tuesdays from 4:30 to 6:00 slot, confirms that it does fit the schedules presented, and then terminates successfully.

Note that a specific result of this experiment is that learning is enabled by the explicit representation of the constraints, *combined* with the connection between this representation and the functional descriptions of the computational units; the task concepts in the TMK models provide the integration of constraint representation and that representation's functional role. The learning of new constraints here takes place with only a single trial; one problem is presented and one piece of feedback is received. Furthermore, it is done without any direct mapping from constraint knowledge to final results; the model simply indicates that the constraints data structure affects the find-first-slot and find-next-slot tasks. Credit assignment over the model and an execution trace is needed to

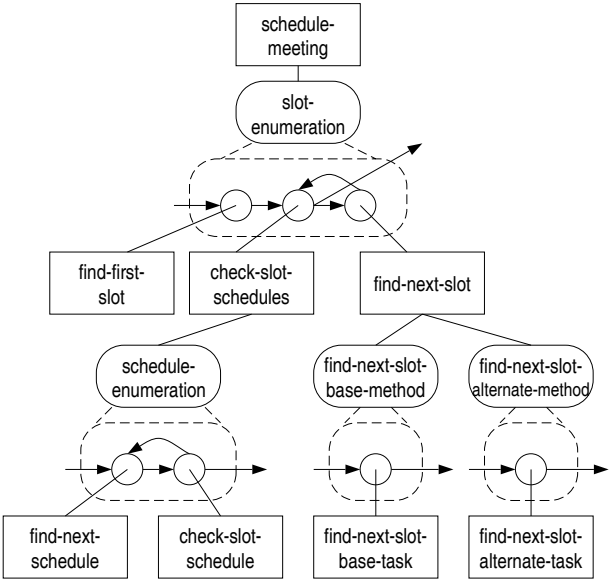

Fig. 2. The revised meeting scheduling agent in which the find-next-slot task has been altered.

determine how the constraints affected the results during the execution. The modification made is guided by this credit assignment and leads to an enhanced system whose constraints are consistent with the feedback provided by the user.

6 Discussion

Meeting scheduling is one member of the broad class of scheduling problems. The issue of learning new constraints seems particularly applicable to scheduling problems because the nature of many of the constraints is reasonably well defined and understood: the goal is to produce a schedule for one or more actions, and the constraints are those things that define what actions can potentially be taken and when. The CAP system [1] explores one approach to the incremental enhancement of a meeting scheduling agent: the system's autonomous actions (in this case, suggesting default values for meeting parameters such as time and location) are treated as classification problems. A set of inductive machine-learning mechanisms are available in this system to classify situations in terms of the desired action to take (i.e, the desired default value to suggest). The machine-learning algorithms used in CAP are very effective at forming useful generalizations over large data sets and require very little explicit domain knowledge. Unlike the SIRRINE2 approach, there is no need to provide a high-level description of the overall behavior of the system; each individual decision point acts as an independent classification problem and receives its own direct feedback. It is not clear, however, how well this approach generalizes to situa-

tions in which the feedback available has less quantity or less direct connection to the actual decision made.

In [4] decisions are also treated as separate classification problems. This system involves different kinds of decisions (learning over interface actions rather than over parameter values) and has a different set of learning algorithms (including reinforcement learning). However, this system, like CAP, does not possess an explicit description of the relationships between the elements of the system and thus is not able to reason across these elements except in regards to tendencies over very many examples. As in CAP, feedback comes in the form of relatively non-intrusive observations of actual user actions, which makes it feasible to collect large volumes of data. However, this approach may not be appropriate for problems such as the one found in our experiment in which the agent performs comparatively elaborate autonomous reasoning, and only very little information is provided by the user during the agent's execution.

Model-based reflection, embodied in SIRRINE2 and related systems, is a mechanism for developing flexible intelligent agents. One significant alternative to this approach involves automatically combining agent operations using one of the many available planning systems [2,6]. The planning approach is a very appealing one because it does not require a designer at all; a user simply describes the primitive operations available in a domain (similar to primitive tasks in TMK) and some goal (similar to a top level task in TMK), and the planning system combines these operations into a plan for achieving that goal. Also, the planning system does not require as inputs all of the intermediate levels of methods and tasks between the overall task and the primitive operations. Furthermore, a planning system is almost infinitely flexible for a given set of operations; the system has *no* preconceived notions of how operations could be combined and thus can provide a completely new sequence of actions for any new input.

There is, however, a key difference between primitive tasks in TMK and operators in a typical planning system. The :given and :makes slots which describe the known properties of the inputs and outputs of the primitive can be significantly underspecified. This makes it possible for some TMK primitives to be much more complex, coarser grained entities than a planning operator can be. Furthermore, the planning system which combines fine-grained planning operators may be prohibitively slow since it has so much information to reason about. The fact that TMK agents must be designed in advance is a limitation of the method in that it is a strong knowledge requirement, but it is also a benefit in that interpreting a predesigned combination of actions is generally much more efficient that constructing a combination of actions during execution.

Domains which are much more complex then they are dynamic are likely to be well served by hard-coded software agents which avoid the need for an explicit model of the domain altogether. Domains which are much more dynamic than they are complex may be relatively easy to encode in a planning system which can provide virtually unlimited flexibility. Model-based reflection seems best suited to domains which present a balance of complexity and dynamics. SIRRINE2 has been tested on a variety of domains in addition to meeting scheduling. These domains include web browsing, bookkeeping, and conceptual design. All of these domains frequently require consistent, repetitive behavior but occasion-

ally make unpredictable demands which require variations on that behavior. By using models of processes and adapting them as needed, SIRRINE2 is able to efficiently handle routine situations *and* effectively react to new challenges.

Acknowledgments. Effort sponsored by the Defense Advanced Research Projects Agency, and the United States Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-2-0229. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Air Force Research Laboratory, or the U.S. Government.

References

[1] Lisa Dent, Jesus Boticario, Tom Mitchell, David Sabowski, and John McDermott. A personal learning apprentice. In William Swartout, editor, *Proceedings of the 10th National Conference on Artificial Intelligence - AAAI-92*, pages 96–103, San Jose, CA, July 1992. MIT Press.

[2] R. E. Fikes and N. J. Nilsson. STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3–4):189–208, 1971.

[3] Todd Griffith and J. William Murdock. The role of reflection in scientific exploration. In *Proceedings of the Twentieth Annual Conference of the Cognitive Science Society*, 1998.

[4] Pattie Maes and Robyn Kozierok. Learning interface agents. In *Proceedings of the 11th National Conference on Artificial Intelligence - AAAI-93*, pages 459–464, Menlo Park, CA, USA, July 1993. AAAI Press.

[5] Eleni Stroulia and Ashok K. Goel. A model-based approach to blame assignment: Revising the reasoning steps of problem solvers. In *Proceedings of the National Conference on Artificial Intelligence - AAAI-96*, Portland, Oregon, August 1996.

[6] D. E. Wilkins. Can AI planners solve practical problems? *Computational Intelligence*, 6(4):232–246, 1990.