Appears in *Case-Based Reasoning Research and Development: Proceedings of the Third International Conference on Case-Based Reasoning*, 1999, pp. 188-202. Springer-Verlag.

A Unified Long-Term Memory System*

James H. Lawton

Air Force Research Laboratory
Rome Research Site
Rome, NY 13441
lawton@ai.rl.af.mil

Roy M. Turner & Elise H. Turner

Department of Computer Science
University of Maine
Orono, ME 04469
{rmt,eht}@umcs.maine.edu

Abstract. Memory-based reasoning systems are a class of reasoners that derive solutions to new problems based on past experiences. Such reasoners use a *long-term memory* (LTM) to act as a knowledge base of these past experiences, which may be represented by such things as specific events (i.e. *cases*), plans, scripts, etc. This paper describes a Unified Long-Term Memory (ULTM) system, which is a dynamic, conceptual memory that was designed to be a general LTM capable of simultaneously supporting multiple intentional reasoning systems. Through a unique mixture of content-independent and domain-specific mechanisms, the ULTM is able to flexibly provide reasoners accurate and timely storage and recall of episodic memory structures. In addition, the ULTM provides support for recognizing opportunities to satisfy *suspended goals*, allowing reasoning systems to better cope with the unpredictability of dynamic real-world domains by helping them take advantage of unexpected events.

1.0 Introduction

Memory-based reasoning systems are a class of reasoners that derive solutions to new problems based on past experiences. Included in this class are case-based [7,2] and schema-based [15] reasoners. The purpose of a *long-term memory* (LTM) in a memory-based reasoning system is to act as a knowledge base of the past experiences, which may be represented by such things as specific events (i.e. *cases*), plans, scripts, etc. The key functions of an LTM are the storage and retrieval of such representational structures. The proper performance of both of these functions is based directly on how the structures are organized in the LTM's knowledge base, and to what extent the LTM can match new experiences to existing structures.

The Unified Long-Term Memory (ULTM) system is a dynamic, conceptual memory [9,5,7] that was designed to be a general LTM capable of simultaneously supporting multiple intentional reasoning systems. Through a unique mixture of content-independent and domain-specific mechanisms, the ULTM is able to flexibly provide reasoners accurate and timely storage and recall of episodic memory structures. In addition, the ULTM provides support for recognizing opportunities to

* This material is based upon work supported by the National Science Foundateion under Grant No. BES—9696004.

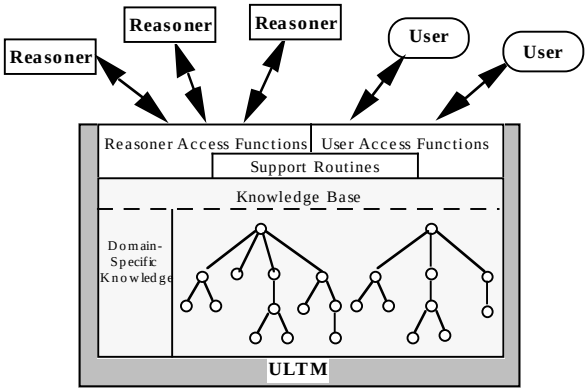

Fig. 1 – ULTM Overview

satisfy *suspended goals*, allowing reasoning systems to better cope with the unpredictability of dynamic real-world domains by helping them take advantage of unexpected events.

As shown in Fig. 1 both reasoning systems and the people who develop them (i.e. "users") access the ULTM's knowledge base through its interface functions. The knowledge base is divided into two parts: the domain-specific knowledge, which the ULTM uses to control its behavior and interaction with the reasoning system(s) using it, and the memory items themselves. The memory items stored in the ULTM's knowledge base represent the various reasoners' experiences. As with many such memory systems, the basic structure for storing and organizing items in the memory is a Memory Organization Packet (MOP) [9]. Unlike most conceptual memories, the MOPs in the ULTM are generic in nature, meant to be the building blocks that reasoning systems will use to create their own structures to be stored in and retrieved from memory. These are the structures the reasoners actually work with, and, although they will be called different names in the various reasoning systems, we generically refer to these representations as either a case, if it represents a specific experience, or a MOP, if it represents a generalization of several cases or other MOPs. These MOPs and cases are organized into a hierarchy, with more general MOPs pointing to (loosely speaking), or *indexing*, more specialized MOPs or cases.

The ULTM has been tested with two particular memory-based reasoners: *Orca*[1] [16,17] and *CoCo* (a generalization of JUDIS [13]). Orca is a schema-based reasoning (SBR) system currently being developed as an intelligent control system for autonomous underwater vehicles (AUVs). SBR systems represent most or all problem-solving knowledge explicitly as MOP-like declarative knowledge structures called *schemas*, which are used to guide all facets of behavior. CoCo is a conversational controller that is to be part of a natural language interface to a system of multiple AUVs. CoCo uses knowledge about intentions and conventions in discourse, represented as *Conversation MOP*s (or C-MOPs) [4], to organize the conversation goals of a distributed system.

[1] In fact, much of the core functionality of the ULTM is based on Orca's schema memory.

Slot	Description
`predictive`	A list of those features expected (by the user) to uniquely identify items in memory.
`elaboration-heuristics`	MOP-specific heuristic functions for index elaboration.
`preference-heuristics`	MOP-specific preference heuristics for MOP selection.
`index-generation-heuristics`	MOP-specific heuristic functions for generating indices.
`suspended-goals`	Place to attach goals in the hope they will be recalled opportunistically.
`bookkeeping`	A placeholder for bookkeeping information, such as recency and frequency statistics, generalization information, and predictive feature tracking.
`exemplars`	Place to store cases and MOPs that, while they fit the current MOP, could not be immediately indexed.

Fig. 2 - MOP Slots

This paper describes the unique capabilities of the ULTM. These include the ability to support multiple reasoning systems simultaneously, the various mechanisms for providing domain-specific knowledge to the ULTM that is used to "fine-tune" the retrieval and storage processes for each reasoner, and the support for recognizing potential opportunities to satisfy suspended goals. It is assumed that the reader is familiar with conceptual memory and memory-based reasoning. Background information on these topics can be found in [9,5,7].

2.0 Memory Structures

A key contribution of the ULTM is that it is capable of supporting multiple memory-based reasoning systems simultaneously. The foundation for this capability lies in the core memory structures used: the MOPs. The ULTM's MOPs are an extension of traditional MOPs. They are generic in nature, providing basic support for knowledge representation, along with extensive support for memory functions. It is expected that the reasoning systems using the ULTM will base their memory structures (i.e. their plans, scripts, etc.) on the ULTM structure `MOP`[2], inheriting these core capabilities.

As with other conceptual memories, the memory items in the ULTM's knowledge-base are organized as a network in which each node is either a MOP or a specific experience (i.e. a case). Each MOP contains generalized information characterizing the episodes it indexes, called its *norms* or *content frame*, and a set of indices for those episodes based on their differences. Indices point from an *indexing MOP* to either an individual case or another, more specialized MOP (the *indexed MOP*), thus forming a MOP/sub-MOP hierarchy [5].

In addition to the actual memory items, the ULTM's knowledge base contains the domain-specific knowledge needed for correct memory operation. Much of this knowledge is in the form of reasoner-specific *heuristic functions*, which are used to tailor the ULTM's retrieval and storage mechanisms to fit the particular domain. This knowledge is associated with the corresponding memory items through slots in the

[2] The knowledge representation system used by the ULTM is the frame system FrameWork [14], which is itself implemented using the Common Lisp Object System (CLOS) [11].

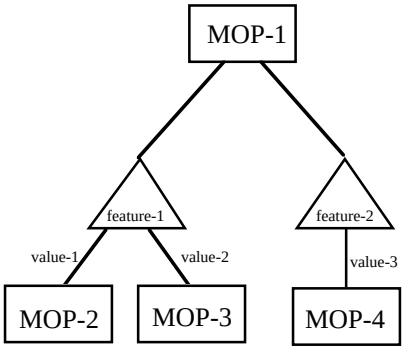

Fig. 3 – Index Structure

`MOP` structure, which are listed in Fig. 2. The meaning of each of these slots is explained throughout the remainder of this paper.

An index (see Fig. 3) in a conceptual memory is a two-tiered structure of features and values, which are taken from an *indexing vocabulary* [7]. An indexing vocabulary is a set of feature names and associated values that are used to construct the indices in the MOPs. The ULTM's indexing vocabulary (shown in the example in Section 4.3) requires the feature names to be slots in the MOPs being used by the reasoning system. The implementer of the reasoning system (i.e. the "user") must specify which slots of the system's frames should be considered *predictive features*, those that will be used to search and generate indices, by listing those slots in each MOP's `predictive` slot. The ULTM's indexing vocabulary also requires that the index values be described as properly formatted predicated functions, as used in [16,17].[3]

3.0 Memory Retrieval

Memory retrieval occurs when a reasoning system requests the LTM to recall any memory items matching a given *probe*, which is a description of a situation made up of features and associated values. The LTM searches its collection of stored experiences, recalling those that most closely resemble the probe.

The ULTM uses the standard retrieval process for MOP-based conceptual memories: directed search [5,7]. Memory retrieval is initiated by a reasoning system by calling the `retrieve` function. This function performs a search starting from the appropriate starting points, or *contexts*, looking for items in memory that most closely match the given probe, guided by the predictive features.

Determining which (if any) of the MOPs or cases indexed from a given MOP match a probe proceeds as follows: for each feature listed in `predictive`, one or more values are found either in the probe, working memory (if appropriate), or

[3] Using a more principled approach to indexing vocabularies, such as the Universal Index Frame (UIF) for intentional systems [10], is being considered for future work. But, since the UIF is too general to be used directly, customizing it was beyond the scope of this project.

through *index transformation*. These values are then matched against the MOP's indices, by determining if they can make the index value's predicate functions true. If there is a match, the MOP or case pointed to by the matching index is added to a set to be searched further or (possibly) returned.

3.1 Index Transformation

When a value for a feature cannot be found in the given probe or in working memory, or when the value does not match any of the known index values, it may be necessary to infer a value for that feature. Sycara and Navinchandra [12] identify three general methods to perform this process of *index transformation*: elaboration, mutation and abstraction.

Both index elaboration and mutation use heuristics to infer values for features when none can be found in the given probe or in working memory. Elaboration heuristics provide more detail, while mutation heuristics make key changes to known values (e.g. changing sizes, substituting ingredients, etc.) [12]. In the ULTM, we lump both of these transformation methods together and refer to them simply as transformation heuristics.

Since index transformation relies heavily on domain-specific knowledge, it is impossible for any LTM to infer values for every feature. Instead, the ULTM provides a mechanism for the user to provide this domain-specific knowledge in the form of heuristic functions associated with MOPs. While the ULTM does provide a few generic index transformation functions, it is expected that the user will provide the majority of these heuristics.

Because specifying transformation heuristics may be complicated, the ULTM actually provides two mechanisms to add them: as rules of a rule-based system or as regular functions. The rule-based system provides a simple, expressive mechanism for adding elaboration knowledge that may be generally applicable, especially in cross-domain applications. However, there may be times when expressing the desired heuristic information is too difficult using the somewhat restrictive rule syntax, or when the heuristic knowledge may only be applicable to a given set of MOPs or cases (those used by a particular reasoner). In these situations the user would use the more general heuristic function mechanism.

Adding a new transformation rule requires first defining the new rule and then adding the rule to the ULTM's index rule-based system (`index-RBS`). For example, in the SMART simulator [17] where Orca is tested, it is possible to get values for the depth and altitude of an AUV directly from working memory. Suppose, however, one needed to determine how deep the water is at the AUV's current location, which we will call the bottom-depth. This value, the sum of the AUV's depth and altitude, is not directly available, and thus must be computed. The rule that computes this is[4]:

```
Rule index-bottom-depth-rule
   If feature is bottom-depth
      and ?d = current depth from WM
      and ?a = current altitude from WM
```

[4] For the sake of readability, this rule is not given in the actual index-RBS rule syntax.

```
Then
   Conclude bottom-depth = (?d + ?a)
```

Similarly, this heuristic could be described in a function, (e.g. `index-bottom-depth-fcn`). Once this function is defined, the ULTM would be told when to apply it by associating the function (through the `elaboration-heuristics` slot) with the MOP (or MOPs) for which elaborating the bottom-depth feature may be needed. What is important is that in either case (rule or function), if the ULTM cannot find a value for a given predictive feature in the probe or working memory, it will employ any relevant elaboration heuristics to infer a value. In this way a user can tailor and augment the ULTM's general directed search mechanism to insure correct behavior.

Index abstraction is another, somewhat more general, form of index transformation. Instead of using heuristic rules or functions, index abstraction exploits the structure of knowledge represented in a hierarchical frame system. If a direct match for a feature value cannot be found, abstraction attempts to find a match on a similar value (where similar refers to how closely connected the two values are in the knowledge hierarchy) by traversing up generalization and down specialization links.

The ULTM does not do retrieval-time index abstraction, however. Rather, when indices are created, their values are abstracted as much as possible (with respect to the indexing MOP). This method is more efficient, since abstraction need only be done once, and it uses the execution context in effect at storage time, which more accurately describes the situation under which the MOP or case is being stored. This storage-time index abstraction is discussed in more detail in Section 4.1.

3.2 Preference Heuristics

The search of memory described above will produce a set of MOPs and/or cases (which we will collectively refer to as MOPs) that have matched the various features in the given probe. However, the retrieval should only return a limited number of MOPs: those that match “best.” The problem of choosing the best matching MOPs, known as the *selection problem* [6], is handled by the ULTM through the use of *preference heuristics* [6]. These heuristics are functions that rank the set of MOPs according to various criteria.

The ULTM provides several of the more common heuristic functions, which are based on those used in PARADYME [6]. These functions rank the retrieved cases based upon the following criteria: how well they (i.e. the retrieved cases) relate to the reasoner's current goals, how salient and specific the features of the retrieved cases are with respect to the given probe, and how frequently and recently the retrieved cases were previously recalled. Each of the common heuristic functions provided by the ULTM is given a particular case and a list of other cases to rank it against. It returns a numeric score -- either a bonus (value > 0), penalty (value < 0), or neutral (0) value -- which is added to the cases’ composite score. The cases are ranked by highest composite score after applying all of the relevant preference heuristics.

While the set of preference heuristic functions provided in the ULTM should be generally applicable to many intentional reasoning systems, it is likely that the reasoners using the ULTM will also need to apply some domain-specific knowledge

to the ranking of retrieved cases. To support this, a mechanism is provided to allow the user to specify their own MOP-specific heuristics, by associating new preference heuristic functions with relevant MOPs through their `preference-heuristics` slot. The ULTM automatically applies these additional functions whenever it retrieves MOPs or cases that have such functions associated with them, adding their returned values to the composite score.

3.3 Predictive Feature Tracking

The ULTM provides limited support for predictive feature tracking, which refers to the recording of how often each predictive feature leads to a reminding. This is done by keeping a record of *feature references*, which are how often each of a MOP's predictive features is used, as well as *MOP references*, which are how often a MOP has been searched. This information is automatically associated with the MOPs (through their `bookkeeping` slot), and is retrieved with a set of accessor functions.

One should note that this form of feature tracking is very limited. It does not keep track of which features actually *contributed* to determining which MOPs were actually returned by a memory search. Rather, it merely tracks which features led to possible choices, at the individual MOP level. To truly track the predictiveness of a given feature, the ULTM's mechanism would need to be extended with more sophisticated machine learning techniques.

Also, the ULTM does not currently do anything with this tracking information. Rather, it is provided for use by reasoning systems in such things as preference heuristics and perhaps feature "forgetting." For example, one could create a preference heuristic that gives a bonus to MOPs or cases that were arrived at through features with a high feature-reference to mop-reference ratio. Similarly, one could remove (forget) features from a MOP's `predictive` list if that ratio drops below a certain threshold.

4.0 Memory Storage

As new events are experienced, the reasoning process may want to store them in memory so that they may be later retrieved. The same search process is used to find a place to store a new MOP or case in memory as would be used to retrieve it. That is, using the case as a probe, its features are used to first select an initial context, and then to traverse indices matching those features. At each MOP encountered during the search, there are four possibilities that may occur for each of the MOP's predictive features that the probe has a value for (modified from [5]):

1) Nothing else is indexed in the MOP by that feature.
2) One or more other MOPs are indexed in the MOP by that feature, but with values that differ from the probe's.
3) One or more other MOPs are indexed by that feature/value pair.
4) The feature/value pair is one of the MOP's norms.

For the first of these possibilities, we know that the probe (the MOP or case being stored) contains a value for a predictive feature that is not currently being used in an index. As such, we could just generate an index using that feature/value pair. But in the ULTM, to be consistent with the retrieval process, as well as to keep the number of indices from growing out of control, we do not. Instead, we collect all of these "leaf" MOPs found during the search of the knowledge base and, after the search has completed, pass them through the preference heuristics. For each MOP selected by the preference heuristics, a set of indices is generated by determining the differences between it and the probe. MOP differences are determined by comparing the values for each predictive feature in the indexing MOP against values for those features in the probe (the indexed MOP). Values that differ are used to generate indices. The index generation process is discussed in Section 4.1. It may also be necessary to update the norms of any MOP we add indices to, which is done through the process of *MOP generalization*, described in Section 4.2.

For the second possibility, we could treat the MOP currently being searched similarly to how it is treated in the first possibility: as a leaf node. But, since it is actually an internal node, we know that it would be unlikely to be selected by the preference heuristics (because of the specificity preference). Thus, in the ULTM we have decided to directly index the probe under the current MOP when this situation occurs. We know the probe has a value for a predictive feature that is not currently being used in an index, so we can simply generate an index using this feature/value pair (using the index generation process described in Section 4.1). As before, it may be necessary to update the indexing MOP's norm (Section 4.2). It should be noted that for this possibility indices are generated during the search process.

In the third possibility, there are two situations we need to contend with. First, if the probe is more specialized than the sub-MOP that was found indexed under the current MOP, then the search simply continues from the indexed sub-MOP. Otherwise, the probe is indexed under the current MOP, using the same difference method described for possibility 1 above. Unlike possibility 1, however, these indices are generated during the search.

Finally, for the fourth possibility, no indices are generated for the given feature/value pair. So that it won't be lost, however, if the probe cannot be indexed by any predictive feature, it is added to the MOP's `exemplars` list. The MOPs in this list, along with any indexed MOPs, are used to update a MOP's norms through the generalization process (Section 4.2).

4.1 Index Generation

Once a location for the MOP or case being stored is found, one or more indices must be generated for it. The ULTM uses the same mechanism for generating new indices, regardless of which possibility from Section 4.0 applies. The index generation process, shown in
Fig. 4, is given a feature and a value, which we will call the *probe filler*, that was found for that feature either in the probe, in working memory, or through transformation. The job of the index generation mechanism is to first abstract the probe filler as much as possible (with respect to the corresponding filler in the

indexing MOP, called the *MOP filler*) and then convert it into a properly formatted index value (predicate function). In keeping with its overall philosophy, the ULTM provides a set of general mechanisms which can be augmented with domain-specific knowledge to accomplish this task.

The probe filler is first *minimally abstracted*, which converts certain "raw" values (e.g. numbers and instance objects) into more standard values used by the ULTM. Next, any *slot-specific abstraction heuristics* are applied to the probe filler. These functions allow the user to abstract any value in non-standard (i.e. domain-specific) ways, thus extending the abstraction mechanism. By using such functions, indices can be generated that match less specific probes. The ULTM first looks for slot-specific abstraction heuristics associated with the probe, and then with the indexing MOP.

For example, suppose we are indexing `new-MOP` under `old-MOP`, the feature we are indexing on is `depth`, and we are given a probe filler of 50. Minimal abstraction would convert 50 into the range: `(range (low 50) (high 50))`. Suppose further that the `depth` slot of `new-MOP` has a slot-specific abstraction heuristic function associated with it that abstracts a range representing a depth by subtracting 10 from the low value and adding 10 to the high value, thus "widening" the range. The new value for depth would thus be the range: `(range (low 40) (high 60))`.

After applying any slot-specific abstraction heuristics, the highest abstraction of the probe filler (with respect to the corresponding MOP filler) is found. This primarily applies to values for which an abstraction hierarchy can be used (e.g. frames). If the probe filler is a descendent of the MOP filler, it is abstracted as far as possible up the hierarchy such that it is still a descendent of the MOP filler. If the probe filler is not a descendent of the MOP filler, but they do have a common ancestor, then the probe filler is abstracted up to the common ancestor. The MOP filler will be updated later during MOP generalization (Section 4.2). If the fillers are unrelated, no further abstraction is performed.

After the probe filler has been abstracted as much as possible, it is used to generate index values (i.e. properly formed predicate functions). While a default predicate form is provided by the ULTM (a general pattern matching predicate), the ULTM provides a mechanism to apply MOP-specific heuristic functions to the abstracted

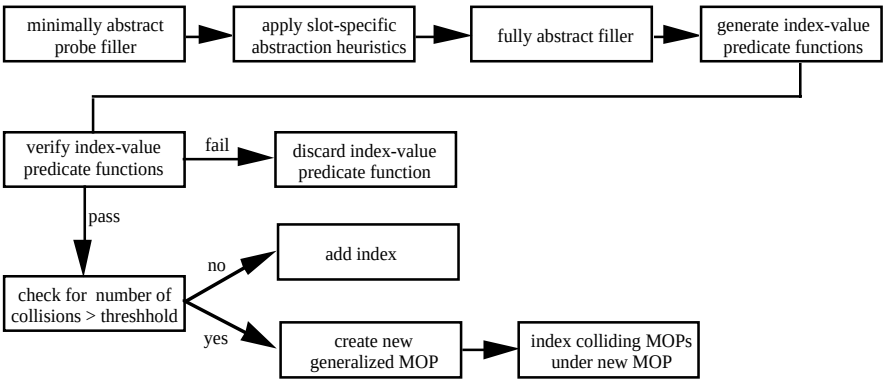

Fig. 4 - Index Generation Process

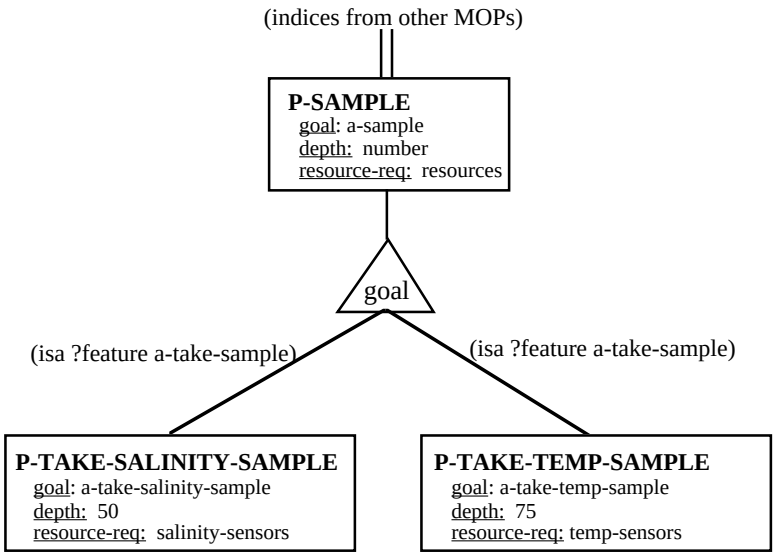

Fig. 6 - Initial Memory Contents

filler to produce index values. These heuristics are associated with MOPs through the `index-generation-functions` slot.

It is possible to generate an index that is too abstract, especially using the default mechanisms. To detect this, we verify the generated index functions by seeing if they match the indexing MOP. If they do, the index function is discarded. If not, the index function next must be tested to see if it causes a *collision* with any of the indexing MOP's existing indices. Two indices collide if their index values match and they point to different sub-MOPs. An index is simply added to the indexing MOP if it does not cause a collision.

If the index does cause a collision, but the number of colliding indices is below a certain threshold value, the index is also just added to the indexing MOP as usual. When, however, the number of colliding indices exceeds the threshold, a new MOP is created as a generalization of the colliding MOPs. The newly created MOP is indexed under the current indexing MOP, while the colliding MOPs are indexed under the new MOP by their differences.

4.2 MOP generalization

Generalization [5,7] is the process by which the memory system updates the content frames of MOPs. *Initial generalization* occurs when a new MOP is formed because of a collision. In this situation, the generalization mechanism is given a new MOP and several sub-MOPs. It must fill the content frame of the new MOP with the features common to the sub-MOPs. That is, for each feature the sub-MOPs have in common, it must find the *central tendency* of the fillers for that feature in all of the

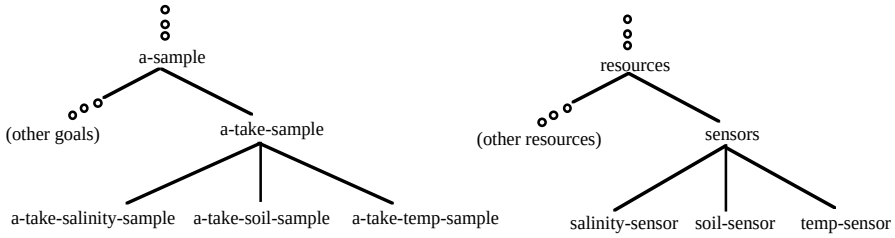

Fig. 7 - Abstraction Hierarchy

sub-MOPs. The central tendency is a sort of "average" value of the fillers taken together, and may be defined differently for each type of filler.

Generalization updates are made after the number of new sub-MOPs (those indexed under a given MOP since the last generalization) exceeds a threshold value. The same central tendency mechanism is used to update a MOP's feature values as was used when it was initially generalized.

To compute the central tendency of a collection of input values, the ULTM first determines the dominant (i.e. most commonly occurring) type of the values. Based on the dominant type, it then calls the appropriate specialized procedure. Currently, specialized procedures are defined for symbols, sets, numbers (including ranges), lists and frames. Users may create other specialized central tendency procedures to support domain-specific filler types.

4.3 Storage Example

This section presents an example of the storage process in detail. To start, suppose our memory contains, among other things, plans for different ocean sampling missions (see Fig. 6)[5]. We wish to add `P-TAKE-SOIL-SAMPLE`, a plan that describes how to perform a soil sampling mission. For the sake of this example, we will assume that `P-SAMPLE`'s only predictive feature is `GOAL`, that our knowledge base contains the abstraction hierarchy (fragments) of goals and resources as shown in Fig. 7, and that the ULTM's index collision threshold is set to 3.

We will assume the search arrives at `P-SAMPLE`. Since the probe (`P-TAKE-SOIL-SAMPLE`) is not a specialization of the MOPs already indexed under `P-SAMPLE` (`P-TAKE-SALINITY-SAMPLE` and `P-TAKE-TEMP-SAMPLE`), it is determined that possibility 1 from Section 4.0 applies. `P-TAKE-SOIL-SAMPLE` is thus indexed under `P-SAMPLE` using the difference procedure from Section 4.0, which determines that the two plans do differ on the predictive feature `GOAL`.

The probe has a filler of `A-TAKE-SALINITY-SAMPLE` for the feature `GOAL`. Minimal abstraction does not change this value, and we will assume that there are no slot-specific heuristics associated with the `GOAL` slots of either `P-TAKE-SALINITY-SAMPLE` or `P-SAMPLE`. We next abstract `A-TAKE-SALINITY-SAMPLE` up the abstraction hierarchy (Fig. 7) to `A-TAKE-SAMPLE`. The ULTM's default index-generation heuristic is used to generate the index value function `(isa ?feature A-TAKE-SAMPLE)`. While this

[5] We omit many of the details of the various MOPs, focusing on only those features and values that are relevant to indexing in this specific example.

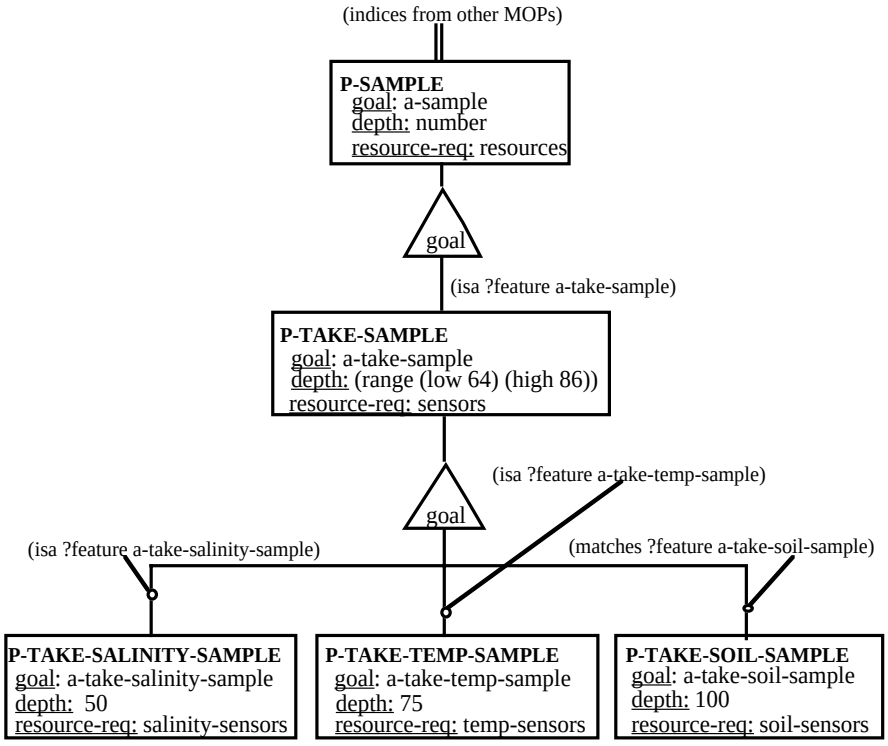

Fig. 8 - Final Memory Contents

function will verify, it causes collisions with the existing indices for both `P-TAKE-SALINITY-SAMPLE` and `P-TAKE-TEMP-SAMPLE`. Since the collision threshold (3) has been met, we must use our collision handling procedure, which causes a new MOP (which we'll call `P-TAKE-SAMPLE`) to be generated and initially generalized. Initial generalization fills `P-TAKE-SAMPLE`'s `GOAL` slot with `A-TAKE-SAMPLE`, its `DEPTH` slot with `(range (low 64) (high 86))`[6], and its `RESOURCE-REQ` slot with `sensors`. The final structure of memory is shown in Fig. 8.

5.0 Support for Opportunism

When using reasoning systems that utilize a conceptual memory, goals that cannot be immediately satisfied can be suspended and stored in memory, indexed by the blocked goals along with the features that are blocking their progress, but which are not currently available. This process is referred to as *predictive encoding* [8]. These suspended goals can then presumably be found by the regular search mechanism the memory system uses whenever a reasoning system requests a retrieval with an

[6] Computed as the mean of 50, 75, and 100 ± (0.5 * standard deviation), which is 75 ± 11.

appropriate probe. This approach is referred to as *opportunistic memory* [3], and is supported by the ULTM.

5.1 ULTM Opportunism Support

There are two sides to the opportunity recognition problem: the reasoning system's and the memory system's. First, the reasoning system must be able to identify what circumstances are blocking a goal's progress. Then, using the goal and the circumstances impeding it to form a probe, the memory system can use its regular search mechanisms to find places to attach the suspended goal. Any time in the future the memory system retrieves a MOP or case with a suspended goal attached to it, it needs to notify the reasoner, which must then determine what to do with that goal.

To provide support for opportunism, the ULTM's `MOP` structures have a slot called `suspended-goals`, which is used by the memory system to associate suspended goals with the MOPs. Further, two functions are provided to allow reasoning systems to suspend and remove goals in memory: `suspend-goal-in-ltm` and `unsuspend-goal-in-ltm`.

The function `suspend-goal-in-ltm` searches *all* memory contexts (i.e. all starting points), retrieving any MOP the goal could be associated with. All contexts are searched to increase the chances that a cross-domain opportunity will be recognized. The goal, along with a descriptor of the reasoning system suspending the goal, is associated with each MOP's `suspended-goals` slot, while the goal maintains a list of MOPs it is suspended on. The latter list is used by the `unsuspend-goal-in-ltm` function to simplify finding everywhere the goal was attached.

Any time the ULTM finds a suspended goal, it must notify one or more reasoning systems. It uses both an asynchronous and a synchronous mechanism for this task. The synchronous method is simple: in addition to the list of MOPs that were found to match the probe, the `retrieve` function also returns a list of suspended goals that are attached to those MOPs.

The synchronous mechanism allows the reasoning system making a retrieval request to detect when a suspended goal has been found. However, that reasoning system may not be the one that originally suspended the goal. The asynchronous notification mechanism is able to notify the reasoner that suspended the goal by calling a *handler function* registered by the reasoner. These handlers are user-defined, and are expected to send a message to the registered reasoner, allowing it to deal with the suspended goal asynchronously.

5.2 Beyond Suspended Goals

We, along with almost all other researchers working in the field of opportunistic reasoning, have focused almost exclusively on the recognition of opportunities to satisfy suspended goals. This is for a good reason: goals are fundamental to intentional reasoning systems. In fact, Francis [1] claims that opportunities *must be* relevant to some goal held by the reasoning system. In spite of this contention, in this

section we consider predictively encoding in memory things other than goals that would lead to opportunity recognition.

Stepping back for a moment, we note that the key functionality of the predictive encoding mechanism is not that it can support the recall of suspended goals, but rather that it allows the reasoning system to be *reminded* of something, anything, that has been previously considered (reasoned about). Thus we can conceivably store anything in memory that will cause the reasoner to interrupt its current activity and reconsider whatever it was reasoning about when the item was stored.

For example, in the near future we will be undertaking a study into utilizing opportunistic memory to recognize when a group of AUVs should restructure their organization (the *re-organization problem*). Using such an approach, a reasoner would select an initial organization for a group of AUVs based upon the given mission and the currently available resources (e.g. the number of AUVs and the equipment they carry). Suppose, however, that during the process of deciding on the initial organization, another organization (represented by some structure which we will call `Org-1`) is considered that would be superior, but cannot be selected because some resources are missing. `Org-1` could be predictively encoded in memory, using the missing resources as recall cues. Should those resources later become available, `Org-1` would presumably be recalled by the memory system, which would notify the reasoner.

The problem is that the reasoner must then determine what to do with this reminding. When the item suspended in memory was a goal, this was fairly easy: just recheck the conditions that caused the goal to be suspended, and reactivate it if they are now met. We can do this because our reasoning systems already have the infrastructure for dealing with goals. Reasoners would have to be modified to handle remindings of other types. At this point, the extent of those modifications is a research issue to be dealt with in the near future. It should be noted, though, that there is nothing in the ULTM's support for opportunism, as described in Section 5.1, that precludes using it for suspending things other than goals.

6.0 Summary

The ULTM is a dynamic conceptual memory system that is capable of supporting multiple reasoning systems simultaneously. It uses established structures and procedures for all primary memory functions. Through a unique mixture of content independent and domain specific mechanisms, it is able to provide reasoners accurate and timely storage and recall of episodic memory structures in a flexible and robust manner. Additionally, the ULTM provides support for recognizing opportunities to satisfy suspended goals, allowing reasoning systems to better cope with the unpredictability of dynamic real-world domains by helping them take advantage of unexpected events.

References

1. Francis, A.G. Jr. (1997). "Memory-Based Opportunistic Reasoning", Ph.D. Thesis proposal, Georgia Institute of Technology.
2. Hammond, K. (1990). "Case-Based Planning: A Framework for Planning from Experience", *The Journal of Cognitive Science*, 14(3).
3. Hammond, K. (1993). "Opportunistic Memory", *The Journal of Machine Learning*, 10(3).
4. Kellermann, K., Broetzmann, S., Lim, T.-S., and Kitao, K. (1989). "The conversation mop: Scenes in the steam of discourse", *Discourse Processes*, 12(1):27-61.
5. Kolodner, J. (1981). "Organization and Retrieval in a Conceptual Memory for Events", *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*.
6. Kolodner, J. (1989). "Selecting the Best Case for a Case-Based Reasoner", *Proceedings of the Eleventh Conference of the Cognitive Science Society*.
7. Kolodner, J. (1993). *Case-Based Reasoning*, Morgan Kaufman, San Mateo.
8. Patalano, A., Seifert, C., and Hammond, K. (1991). "Predictive Encodings: Planning for Opportunities", *Proceedings of the Fifteenth Conference of the Cognitive Science Society*.
9. Schank, R. (1982). *Dynamic Memory*, Cambridge University Press, New York.
10. Schank, R. and Osgood, R. (1990). "A content theory of memory indexing", Northwestern University, Institute for Learning Sciences Technical Report no. 2.
11. Steele, G. (1990). *Common Lisp: The Language (Second Edition)*, Digital Press, Bedford, MA.
12. Sycara, K. and Navinchandra, D. (1991). "Index Transformation and Generation for Case Retrieval", In *Proceedings of the 1991 Case-Based Reasoning Workshop (DARPA)*, Bareiss, E. (ed.), Morgan Kaufman, San Mateo, CA.
13. Turner, E. (1990). "Integrating Intention and Convention To Organize Problem Solving Dialogues", Ph.D. Dissertation, Georgia Institute of Technology technical report GIT-ICS-90/02.
14. Turner, R. (1987). "Issues in the design of advisory systems: The consumer-advisor system", in *Proceedings of the Eleventh Annual Conference of the Cognitive Science Society*, Detroit, MI.
15. Turner, R. (1994). *Adaptive Reasoning for Real-World Problems: A Schema-Based Approach*, Lawrence Erlbaum Associates, Hillsdale, NJ.
16. Turner, R. (1995a). "Context-Sensitive, Adaptive Reasoning for Intelligent AUV Control: Orca Project Update", In *Proceedings of the 9th International Symposium on Unmanned Untethered Submersible Technology (AUV'95)*, Durham, New Hampshire.
17. Turner, R. (1995b). "Intelligent Control of Autonomous Underwater Vehicles: The Orca Project", Roy M. Turner. In *Proceedings of the 1995 IEEE Conference on Systems, Man, and Cybernetics*, Vancouver, BC, Canada.
18. Turner, R. (1997). "Orca Documentation (for Version 2.1)", CDPS Research Group in-house report, University of Maine. http://cdps.umcs.maine.edu/Docs/orca-2.0/