Towards Adaptive Web Agents

J. William Murdock and Ashok K. Goel
College of Computing
Georgia Institute of Technology

Abstract

There is an increasingly large demand for software systems which are able to operate effectively in dynamic environments. In such environments, automated software engineering is extremely valuable since a system needs to evolve in order to respond to changing requirements. One way for software to evolve is for it to reflect upon a model of its own design. A key challenge in reflective evolution is credit assignment: given a model representing the design elements of a complex system, how might that system localize, identify and prioritize prospective candidates for potential modification. We describe a model-based credit assignment mechanism. We also report on an experiment on evolving the design of Mosaic 2.4, an early network browser.

1. Introduction

Software systems are often very good at providing consistent, reliable behavior in static environments but are traditionally much less useful in environments which require flexibility and adaptation. In recent years, however, there has been an increasingly great interest in software environments which undergo rapid change. A particularly notable dynamic software environment is the World Wide Web. Because the web is very complex, many software systems which operate over the web fall under the intelligent agent paradigm; they involve not only strong interaction with the environment but also complex deliberative reasoning. However, even complex reasoning systems can be very rigid in their behavior. Because the nature of the web changes very rapidly, it is important for web-based agents to be able to adapt to these changes.

Consider, for example, an intelligent agent which retrieves and presents information from the web: an intelligent web browser. Such an agent must have a range of documents that it is initially able to present to the user; a typical web browser has a small set of document types which it can, itself, display and a much larger set of document types for which it has access to an external viewing program. However, an agent browsing the web is likely to eventually encounter a type of file which for which it has no display mechanism. Such an agent may need to *evolve* in order to adapt to this novel demand.

The SIRRINE2 system is a reasoning shell which supports the addition of new capabilities, as in the new external viewer problem. SIRRINE2 is an automated tool but is part of a larger research project, MORALE [1], which includes both automated *and* interactive tools and techniques for the evolution of legacy software systems. The MORALE tool suite includes a broad range of capabilities for analyzing and evolving software systems and their requirements. Integration of these tools is done through sharing information in the ACME [2] architectural interchange format.

Agents in SIRRINE2 are defined by *functional models*: explicit representations of the agent which define not only what the elements of the agent are and how they are combined but also what those elements are for. The addition of new capabilities in SIRRINE2 occurs through *reflection*: an agent's reasoning about itself. A functional model of an agent acts as the self-knowledge which that agent reflects over.

The process of adding capabilities through model-based reflection involves many challenges. One interesting issue is that of credit assignment: determining which components have some specified effect on the behavior of a system. The credit assignment process in SIRRINE2 requires four kinds of knowledge as input: a model of an agent, a trace of the execution of that agent in some situation, the result of that execution, and an alternative desired result. The process produces a list of possible localizations of credit. Each such localization refers to an element of the agent design which may potentially be responsible for the difference between the actual result and the desired result. Additionally, each localization contains a tag which identifies the nature of the relationship that the identified element has with the difference between the desired and actual results. Each localization identifies a potential candidate for a modification to the agent; thus the credit assignment process plays a central role in guiding the evolution of software agents in SIRRINE2.

2. Knowledge Structures

Agents in SIRRINE2 are modeled using the Task-Method-Knowledge (TMK) language. Predecessors of this language have been used in a number of other research projects such as AUTOGNOSTIC [4]. The agent's *TMK model* is directly accessible to the evolutionary reasoning mechanism as declarative knowledge about the agent's processing. 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? Each 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, "primitive tasks" are defined which may not be further decomposed. Descriptions of knowledge in TMK is done through the explicit specification of both the concepts and the relationships that the agent uses. The knowledge portion of the model is linked to the task portion in that the functional specification of a task refers to the knowledge that the task manipulates.

In addition to the models themselves, the credit assignment process in SIRRINE2 makes use of a *trace* of an execution, the results of that execution, and some description of the desired results of the execution. A result (either actual or desired) is represented as a *knowledge state*. The output of the credit assignment process is a list of *localizations*. The form and content of traces, knowledge states, and localizations are closely tied to the TMK language for modeling agents. For example, the elements of a trace refer directly to the tasks and methods which were invoked as the trace was generated.

3 Credit Assignment Algorithm

The process of evolutionary reasoning in SIRRINE2 consists of four major steps. During the first step, the system attempts to execute an agent for some given knowledge state. If this execution is successful, SIRRINE2 is done; if not, it proceeds to the remaining three steps. The second step is credit assignment, which produces a list of localizations of candidates for modification. The third step is modification, which takes this list and steps through it (in order) until it finds a localization which it is able to address (i.e., for which it can make a change which may allow the agent to execute successfully), and then makes this change to the agent. In the fourth step, the revised agent is executed to verify that the change made successfully implements the desired new functionality.

The algorithm for the credit assignment process is presented in the sidebar. A main algorithm called assign-credit checks if credit assignment is necessary and, if so, passes control to two mutually recursive routines labeled assign-task-credit and assign-method-credit here. The former checks for possible flaws in a task using a task-trace knowledge element and then, if necessary, runs assign-method-credit on the method trace for the method that was invoked for that task. That routine then checks the method for possible flaws and then calls assign-task-credit on the traces of the subtasks invoked by that method.

The assign-task-credit routine provides most of the analytical power of the credit assignment process. The behavior of this routine depends on both the nature of the task which was invoked and the nature of the actual and desired knowledge states involved in that task's execution. If the task is not intended to produce the desired knowledge state, it is possible that this task needs to be redesigned to do so for the current problem. Thus if the TMK model indicates that the task invoked generally does not produce the knowledge for which the actual and desired knowledge states differ, a :task-does-not-produce-value localization is recorded for that task. If the task is a primitive task which *is* intended to produce the desired state and it executed successfully, it is possible that the primitive task needs to be fixed so that it behaves as intended. In this case, a :primitive-generates-invalid-state localization is recorded for that task. If the task being analyzed is a primitive task which is intended to produce the desired state and it did execute unsuccessfully, this failure could be responsible for the difference between the actual and desired knowledge states. In this situation, a :primitive-fails localization is recorded. Lastly, if the task is not primitive and it is already designed to produce the desired knowledge state, its failure to do so can only be attributed to a problem with the subcomponents of this task. In this situation, no localization is attributed to that task and credit assignment simply continues by calling the assign-method-credit routine on the method which the task invoked.

A key issue here is that of the order in which a trace is analyzed. This can be important because a trace may be very large and moving through the elements of a trace in an arbitrary order could take a prohibitively long time. SIRRINE2 uses two heuristics to control the order of analysis: *Functional Abstraction* and *Causal Proximity*. Functional Abstraction has traces analyzed from top to bottom, i.e., most abstract to least abstract. This is done in order to prefer more general solutions where possible. The Causal Proximity has the subtasks of a method analyzed from last to first. This is done in order to prefer analyzing elements

```
function assign-credit(Trace tr, TMK-Model start-tmk,
                       Knowledge-State end-ks,
                       Knowledge-State desired-ks)
   Knowledge-State-Difference ks-difference
   List of Localizations list-loc
   ks-difference = difference(end-ks, desired-ks)
   If null(ks-difference)
     list-loc = list(make-localization(:no-failure))
   Else
     list-loc = assign-task-credit(top-task(tr), start-tmk,
                                   ks-difference)
   Return list-loc

function assign-task-credit(Task-Trace ttr,
           TMK-Model start-tmk,
           Knowledge-State-Difference ks-difference)
   Task ta
   List of Localizations list-loc
   ta = task-trace-task(ttr, start-tmk)
   list-loc = ()
   If not(produces-differences?(ta, ks-difference)
     list-loc += make-localization(
                   :task-does-not-produce-value,
                   ta, ks-difference)
   Else If and(primitive?(ta),successful?(ttr)))
     list-loc += make-localization(
                   :primitive-generates-invalid-state,
                   ta, task-trace-input-state(ttr),
                   ks-difference)
   Else If primitive?(ta)
     list-loc += make-localization(
                   :primitive-fails,
                   ta, task-trace-input-state(ttr),
                   ks-difference)
   If not(primitive?(ta))
     list-loc += assign-method-credit(
                   task-trace-method-trace(ttr),
                   start-tmk, ks-difference)
   Return list-loc

function assign-method-credit(Method-Trace mtr,
            TMK-Model start-tmk,
            Knowledge-State-Difference ks-difference)
   Task-Trace ttr
   List of Localizations list-loc
   list-loc = ()
   For ttr = last(method-trace-subtask-traces(mtr)) to
          first(method-trace-subtask-traces(mtr))
      list-loc += assign-task-credit(ttr, start-tmk,
                                     ks-difference)
      list-loc += make-localization(
                    :method-does-not-produce-value,
                    method-trace-method(mtr, start-tmk),
                    ks-difference)
   Return list-loc
```

that are temporally closer to the final (inadequate) result.

4. Experiments

Our work on SIRRINE2 has involved experiments with several systems, most notably a simulated web browsing agent and a meeting scheduling agent. The web browsing agent which we have experimented on here is named Tiny Intelligent Mosaic (TIM). TIM is intended to initially imitate the behavior of the Mosaic web browser and to intelligently evolve new capabilities. It was built by imitating both the functionality and structure of Mosaic for X Windows, version 2.4. Our initial (pre-evolutionary) model of TIM is based on an architectural analysis of Mosaic 2.4 which was developed using the SAAM methodology [1].

The primary goal of TIM is the display of a document. The system is given a URL by the user as input. The first thing it does is divide that URL into pieces and use those pieces to issue a request to the network. A MIME tag is then extracted from the response generated by the network. If the MIME tag corresponds to a type which can be displayed internally (e.g., HTML), the system does so. Otherwise, if the MIME tag corresponds to a type for which an external viewer is known, that external viewer is invoked on the document retrieved. The task-method decomposition illustrated in Figure 1 follows directly from this analysis.

The Mosaic "Wish List" [3] includes "on-the-fly selection of external viewers" as a desired improvement to Mosaic. We have a working example using SIRRINE2 which addresses a limitted version of this task, i.e., adding a single specific new external viewer.

We have run several experiments with TIM. In one of these experiments, TIM is asked to retrieve the document at the following URL:
http://www.cc.gatech.edu/grads/m/Bill.Murdock/dcsp.pdf

This document is an Adobe Portable Document Format (PDF) file of MIME type application / pdf. Mosaic 2.4, on which TIM is based, does not have a mapping for this MIME type to an external viewer program. Consequently, TIM fails for this task. SIRRINE2 then receives feedback from the user indicating that the command acroread is appropriate to this problem. SIRRINE2 uses this knowledge to evolve TIM so that it is capable of invoking this command when it encounters documents of that MIME type.

First TIM runs and attempts to solve the problem as specified. During execution, a trace of reasoning is generated. For this problem, TIM fails, because it is unable to display the PDF file. After this, feedback is provided by the user which indicates that the command which should have been invoked is acroread. This fact is provided in the form of a desired knowledge state. Since TIM has failed, SIRRINE2 uses the credit assignment mechanism to produce a list of possible localizations of failure. The element of particular

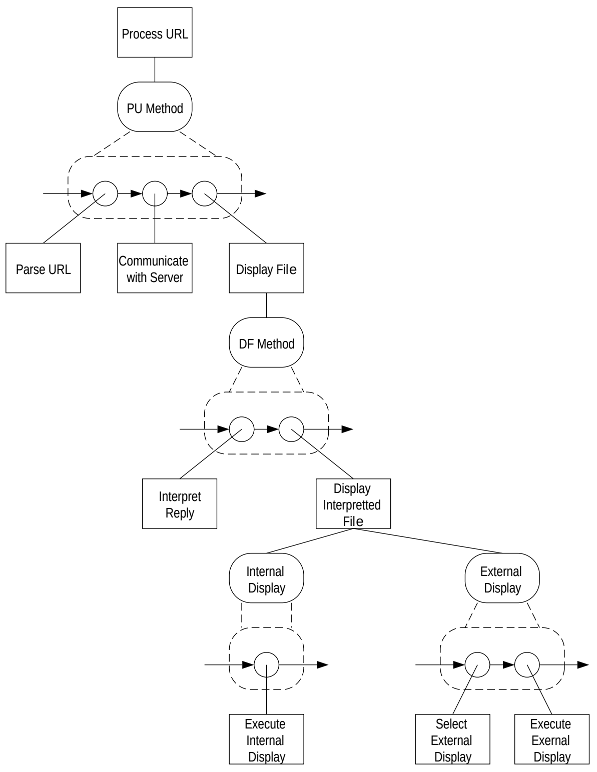

Figure 1. The tasks and methods of TIM. Rectangular boxes indicate tasks while rounded boxes indicate methods. Transition diagrams underneath methods indicate the control imposed by methods on their subtasks.

interest in this example is the select-external-display task. This task should have produced a command but didn't; because SIRRINE2 was given a command as feedback, it posits that the failure of the system may be a result of this task not producing that command.

Given the list of localizations, the modification mechanism is makes a change to the system which corrects the identified fault. The modification mechanism steps through the localizations produced by credit assignment in the order in which they were assigned; i.e., the modification process uses the same heuristics of abstraction and proximity that the credit assignment mechanism does. In the example, the select-external-display task is the first task for which the localization mechanism has identified a failure for which the modification mechanism has an applicable strategy. Since this task is implemented as a simple look-up table, the modification mechanism is able to make the change by simply adding another element to the table: one which maps MIME type application / pdf to command acroread. After modification is done, TIM is run again with the same URL request. This time the modified task is successful and the program successfully displays the document.

In this and other experiments we have run, the Causal Proximity heuristic for ordering credit assignment has been effective in providing a useful organization of potential localizations. Obviously, proximity is not an optimal heuristic in *all* possible cases; in some situations the inability to accomplish some result may arise as a consequence of decisions made much earlier in the reasoning process. However, the fact that this heuristic has proven effective in our current experiments reinforces our initial intuitions that proximal failures are relatively common, and thus favoring them is an efficient ordering heuristic.

The Functional Abstraction heuristic, however, has not provided substantial leverage in the experiments we have conducted. In particular, the revisions made in these experiments is done at the bottom level, and thus the preference to first consider localizations at higher levels is not especially useful. This result is a consequence of the relative simplicity of the agent modification mechanisms; these simple mechanisms are only effective at low levels of abstraction. As more sophisticated modification mechanisms are developed in future work, the effectiveness of the Functional Abstraction heuristic may increase.

Acknowledgments

This work has benefited from discussions with our colleagues in the MORALE group, especially Spencer Rugaber. This effort has been sponsored by DARPA under agreement number F30602-96-2-0229.

References

[1] G. Abowd, A. K. Goel, D. F. Jerding, M. McCracken, M. Moore, J. W. Murdock, C. Potts, S. Rugaber, and L. Wills. MORALE – Mission oriented architectural legacy evolution. In *Proceedings International Conference on Software Maintenance 97*, Bari, Italy, 1997.

[2] D. Garlan, R. T. Monroe, and D. Wile. Acme: An architecture description interchange language. In *Proceedings of CASCON 97*, pages 169–183, Toronto, Ontario, Nov. 1997.

[3] NCSA. Mosaic for X wish list. *http://www.ncsa.uiuc.edu/SDG/Software/XMosaic/wish-list.html*, Nov. 1997. Accessed September 1998.

[4] E. Stroulia and A. K. Goel. Redesigning a problem-solver's operators to improve solution quality. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 562–567, San Francisco, Aug. 23–29 1997. Morgan Kaufmann Publishers.