RC23115 (W0402-097) February 20, 2004
Computer Science

IBM Research Report

Learning Procedures for Autonomic Computing

Tessa Lau, Daniel Oblinger, Lawrence Bergman, Vittorio Castelli
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Corin Anderson
Google
2400 Bayshore Parkway
Mountain View, CA 94043

Learning Procedures for Autonomic Computing

Tessa Lau, Daniel Oblinger, Lawrence Bergman, and Vittorio Castelli
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
tessalau@us.ibm.com

Corin Anderson
Google
2400 Bayshore Parkway
Mountain View, CA 94043

1 Introduction

Today's skilled IT professionals bring to bear an enormous amount of knowledge about how systems are configured, how they function on a day-to-day basis, and how to repair them when they break. However, there are not enough skilled IT professionals to meet the ever-growing demand. Autonomic computing offers a way out of this dilemma: offload the responsibility of managing complex systems onto the systems themselves, rather than relying on limited human resources.

This problem raises a large challenge: how will we transfer the knowledge about systems management and configuration from the human experts to the software managing the systems? We believe this problem is fundamentally a knowledge acquisition problem. Our approach to solving this problem draws on machine learning and knowledge representation. Our core idea is based on programming by demonstration: by observing several human experts each solve a similar problem on different systems, we generalize from traces of their activity to create a robust procedure that is capable of automatically performing the same task in the future. Our solution is based on the observation that solutions to similar problems share similar sub-procedures. By capturing these nuggets of problem-solving knowledge from multiple experts, we form a robust procedure that encapsulates the important parts of the procedures executed by all of the experts.

We are currently employing this approach to acquire desk-side technical support procedures, such as upgrading a network card, troubleshooting email problems, and installing a new printer. Our system captures traces of multiple desk-side support representatives as they perform one task, such as diagnosing a dysfunctional network adapter, under a variety of operational conditions. From these traces, our system generalizes and aligns the traces into a single general procedure for repairing network adapters. An important feature of our approach is that it works across applications, via instrumentation of the Windows operating system.

This paper describes our formulation of this problem as a machine learning problem. First we define the problem and describe how various problem characteristics affect the difficulty of the learning problem. We then outline the subproblems we have identified, and describe our approach to each. Finally, we conclude with a summary of current results and directions for future work.

2 Procedural knowledge acquisition

We formulate the problem of procedural knowledge acquisition as follows.

> Given as input one or more traces of an expert's keyboard and mouse actions as she demonstrates a procedure, output a procedure model that, when executed on a new system, performs the same task.

Our approach to this problem is based on machine learning: given traces of a procedure's execution behavior, induce the procedure. Other research challenges include knowledge representation (how to represent a procedure, a procedure step, and the state of the world) and procedure execution (taking a generalized procedure model and mapping it into the concrete actions required to perform the procedure on a new system).

Clearly, the type of procedure as well as the quality of the traces determines how difficult it will be to construct the procedure model. We have identified a number of problem characteristics that affect problem difficulty:

- **Procedure structure complexity**: A straight-line procedure with no deviations from the main path will be easier to learn than a procedure that has many conditional actions or alternative paths.
- **Trace noisiness**: Execution traces in which the expert performs extraneous steps, or in which unexpected events happen asynchronously, will increase learning difficulty.
- **Incremental or batch learning**: The choice of learning algorithm depends on how it is going to be used. Incremental learning, where a procedure model is updated dynamically as traces are created rather than overnight in a batch process, places different constraints on the algorithms that may be used to learn procedures.
- **State observability**: The choice of action to perform at each step of the procedure depends on how much information is available to the system in making that decision. If the choice can be made based on some information displayed on the user's screen, the problem is easier than if the choice is made based on some hidden variable, such as remembered state stored in the expert's mind.

In the next section, we describe some of the research challenges we have identified while working on this problem, and outline the approaches we have taken on each challenge.

3 Research challenges

Given a trace of low-level events, the first challenge is to *segment* the trace in order to identify procedure steps, procedure components, and boundaries between one procedure and the next. Next we *generalize* traces, mapping from the concrete actions performed on specific windows to a more generalized representation that will work across systems. With several traces, a further challenge is to simultaneously *align* portions of the traces such that subsequences of similar functionality are grouped together. Finally, an *execution* process takes the generalized procedure and runs it on the target system, selecting the correct action to perform at each step.

3.1 Segmentation

Our system captures expert behavior by using low-level Windows operating system instrumentation. This low level instrumentation is necessary to achieve our goal of learning cross-application procedures, and thus does not rely on instrumentation of every application used in the procedure. Our instrumentation provides information about the windows displayed on screen (window titles, button labels, field contents, etc.), mouse and keyboard actions along with the target window of each action, and notification when windows are created, modified, or destroyed. For example, suppose the user launches an application by double-clicking on an icon on the desktop. Our instrumentation reports that the user depressed and released the mouse button twice at location (5, 8) in window with id 10060, then reports a large number of window-creation events, one for each widget in the application being started.

Unlike simple macro recorders, which simply record a user's actions and play them back verbatim, our system models the interaction between the user and the system as a conversation in which user and system take turns participating. Modelling the conversation at this level enables an autonomic procedure to monitor and react to unexpected effects of user actions. For example, if an application failed to appear after the user double-clicked on its icon, then the next steps in the procedure should not be taken until this problem has been resolved.

We use the term *segmentation* to refer to the translation of a low-level event stream into a stream of high-level, semantically meaningful events. This is a challenging problem since events from different high-level actions can be interleaved in the low-level stream. For example, suppose our user double-clicks to launch an application, and then an instant message pops up on her screen, and finally the application appears. The challenge of segmentation is to infer semantics from the low-level stream that describe what actually happened, and to model the cause and effect of a user's actions.

Our approach to segmentation employs grammar parsing techniques. We have defined a number of grammar rules expressing high-level actions in terms of sequences of low-level events. For instance, the sequence of pressing and releasing key “A”, then pressing and releasing key “B”, is parsed into the high-level action “type string AB”.

This approach fails when the high-level actions are interleaved in the event stream. For instance, if a user double-clicks on an icon to launch an application and then performs a different action before the application appears on screen, a grammar-based approach will have difficulty making sense of the sequence. Our solution to this problem will be to maintain multiple candidate parses of the event stream, and select the best parse only after more data has been seen (e.g., more traces from different experts in which the application appears immediately after double-clicking on its icon).

In addition, we wish to segment high-level action streams into procedures and sub-procedures. For example, the event stream described above may be part of a single “launch application” step, which may be in turn be part of the “find out whether Service Pack 3 is installed” subtask, which may be part of the “diagnose network adapter” procedure. Previous systems required the user to identify the start and end of each procedure (by pressing a button on a GUI, for example). However, this may prove to be too much of a burden as procedures become more complex and are logically broken down into sub-tasks, some of which may be common across multiple procedures. For example, a procedure for diagnosing email problems may include a sub-procedure for checking whether the workstation is able to connect to the network. Manually indicating the boundaries of each of these subtasks is certainly going to require too much user effort. One research goal in our work is to consider automated approaches to the segmentation problem.

3.2 Generalization

A segmented trace is a sequence of high-level actions that reference specific windows in the system used for the demonstration. Generalization is the process of identifying salient features in the trace that uniquely describe the actions at a level of detail that enable the trace to be run in a different environment. For instance, when the user double-clicked at location (5, 8) in window with id 10060, a generalized version of this action (that is portable to more systems) could be “double-click on the icon on the desktop named ‘My Computer”’.

In some cases, there may be more than one generalization of a particular user action. For instance, if a user types the string “tomato” into a text field, she may be entering her username, her password, or a constant string. If the wrong generalization is selected, the procedure could fail when run by a user with a different username and password.

Our approach to action generalization is based on a machine learning technique called version space algebra [6], a framework for efficiently enumerating the space of possible generalizations for concrete actions, and maintaining the set of consistent generalizations given one or more examples of the target action. For example, if one user enters the string “tomato” and the next enters the string “cabbage”, the learning algorithm discards the hypothesis “type the constant string tomato”, although the hypothesis “type the user's password” is still plausible.

A more advanced form of generalization is needed to recognize procedures that differ based on the version of the operating system being run. For instance, the organization of the network control panels differs across versions of Windows. A procedure to adjust one setting on Windows 98 requires different steps than a procedure to adjust the same setting on Windows XP. Automatically recognizing and generalizing procedures that differ at this level remains an area for future work.

3.3 Alignment

Our goal is to learn robust procedures from traces generated by different experts, under a variety of conditions. Traces may contain steps in different order, or may contain new subprocedures in which a whole sequence of steps has been added (perhaps to recover from a previous failure). A robust model of the procedure must capture both the well-worn path through the procedure (the common case, in which all the steps succeed) as well as less common paths in which one or more of the procedure steps results in an unexpected error.

The *alignment* problem is to recognize and align together subsequences of similar functionality across multiple traces, so that multiple examples of the same step in different traces can be used to generalize the step. For example, suppose one expert performed an extra set of actions at one point during the trace (for instance, if a command produced an unexpected outcome) before returning to the main procedure. When the expert returns to the main path, alignment is necessary to both detect the deviation as well as line up future actions in this trace with the main path demonstrated by other experts for this procedure.

Our approach to the alignment problem is inspired by previous work in DNA sequence alignment. However, unlike aligning DNA sequences, alignment of procedure traces is intimately tied to the generalization process. Two experts performing steps (e.g., typing a string into a text field) might both be entering their password into a login dialog. Alternatively, one of them might be changing the host name of this computer while the other is typing the subject line of an email message. The alignment of these steps in the procedure model depends on how well they generalize, while the generalizations depend on which steps are aligned together.

Our solution to the combined alignment/generalization problem is based on Input/Output Hidden Markov Models [1]. IOHMMs provide a mechanism for considering all possible alignments and iteratively selecting the locally best alignment. The output of an IOHMM is a probabilistic finite state machine with classifiers at each node that predict both the next action and the next node, given the current state, which includes features that are visible on the user's screen. The classifiers capture the generalization of actions, while the probabilistic finite state machine captures the different possible paths through the procedure.

3.4 Execution

A learned procedure must be *executed* on a new machine to accomplish the task modeled in the procedure. Execution is more than simply a matter of replaying a sequence of actions one by one, however. First, the procedure may contain conditional steps that should only be executed under certain conditions, such as steps that depend on the installed version of the operating system or specific drivers. Second, the system must monitor the result of each action to detect unexpected results, by matching the observed result of the action against the expected result contained within the procedure. For example, one type of unexpected result takes the form of asynchronous events, such as new mail notification or instant message popups. These events could grab the keyboard focus or otherwise interfere with procedure playback. Third, the procedure may be run on a system with different characteristics than the training systems (such as a production server rather than a test server, or an upgraded version of the operating system, or simply one with more users and hence more load). A reliable execution system must take all these factors into account in order to ensure that a procedure can be executed reliably on any system.

Our use of an IOHMM to model procedures addresses these challenges by incorporating a classification mechanism into the procedure representation. At each step, our system examines what is visible on screen and uses that information to inform its decision of how to proceed in the task. For example, if an asynchronous event occurs, the system recognizes that an unexpected window has appeared, and outputs a different probability distribution over the space of possible next actions.

One of our future goals is to interleave learning with execution. As a user is executing a procedure on a new system, she may reach a point where an unexpected failure occurs that is not represented in the procedure. If the user knows how to recover from this failure, she can demonstrate the recovery procedure as she goes, and these steps become part of a more robust procedure. Dynamically updating procedure models during execution will enable autonomic procedures to stay up-to-date even as conditions change over time.

4 Related work

Previous programming by demonstration systems [3; 7] relied on instrumenting a single application in order to track user and system actions at a very high level. For autonomic computing, however, we cannot assume that all the necessary applications will be instrumented, thus forcing us to work with the lower-level event stream available from the operating system. In addition, no previous programming by demonstration system has attempted to automatically learn complex conditional procedures from multiple traces.

Our approach is related to planning, specifically contingent planning [8]. Contingent planners formulate a plan (a sequence of actions) that will achieve a goal, even in the presence of uncertainty about the state of the world. The generated plans employ sensing actions to determine the world state before proceeding to the next action. The goal of our research is also to produce contingent plans. However, instead of reasoning about the desired state of the world and using an action model to synthesize a sequence of actions to achieve the goal state, our system uses demonstrated traces to bias the search for plans. Rather than having an expert define the desired state of the system, we learn from a sequence of demonstrated actions.

Programming by demonstration is also similar to previous work in plan recognition [2; 4; 5]. Instead of defining a plan library and matching an agent's actions against a set of known plans, however, our approach allows an agent to define a new plan by demonstration. Our approach also supports plan refinement by incorporating new demonstrations of the same plan. In addition, unlike most plan recognition systems, our system generalizes individual steps in the plan from the concrete actions performed by the agent to a higher-level action description.

The segmentation problem is similar to previous work on learning plan operators from traces. Wang [9] makes the assumption that each plan operator results in a single state change, and relies on the expert identifying correct and complete descriptions of the state before and after each action. In our case, however, each plan operator (such as launching an application) results in a potentially large number of state changes (where each is reflected by a low-level event, such as a widget in a window being created). In addition, asynchronous events (such as occur in the real world, unlike a simulator) make our problem more challenging.

5 Summary and implications for autonomic computing

We have outlined a research agenda for automatically acquiring procedural knowledge for use in autonomic systems. Our research is based on learning procedures by observing experts perform these procedures on live systems, and dynamically building a procedure model that can be executed on a new system to repeat the same task. As traces of a procedure are accumulated over time, the procedure model is updated. This dynamic learning process enables the procedure to adapt to changing conditions.

The success of autonomic computing relies on the ability of systems to manage themselves and react to changing conditions. Currently, knowledge about how to maintain and configure systems is locked within the minds of skilled experts. Our research goal is to facilitate knowledge acquisition from these experts, simply by watching them do what they do best, and to produce intelligent systems that embody this knowledge.

References

[1] Y Bengio and P Frasconi. Input-output HMMs for sequence processing. *IEEE Transactions on Neural Networks*, 7:1231 – 1249, (1996).

[2] E. Charniak and R. Goldman. A probablistic model of plan recognition. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, volume 1, pages 160–5, July 1991.

[3] Allen Cypher, editor. *Watch what I do: Programming by demonstration*. MIT Press, Cambridge, MA, 1993.

[4] B. Goodman and D. Litman. On the interaction between plan recognition and intelligent interfaces. In *User Modeling and User Adapted Interaction*, volume 2, pages 83–115, 1992.

[5] H. Kautz. *A Formal Theory Of Plan Recognition*. PhD thesis, University of Rochester, 1987.

[6] Tessa Lau, Pedro Domingos, and Daniel S. Weld. Programming by demonstration using version space algebra. *Machine Learning*, 2003. To appear.

[7] H. Lieberman, editor. *Your Wish is My Command: Giving Users the Power to Instruct their Software*. Morgan Kaufmann, 2001.

[8] L. Pryor and G. Collins. Planning for contingencies: A decision-based approach. *J. Artificial Intelligence Research*, 1996.

[9] Xuemei Wang. Learning by observation and practice: An incremental approach for planning operator acquisition. In *Proceedings of the 12th International Conference on Machine Learning*, 1995.