From: AAAI-84 Proceedings. Copyright ©1984, AAAI (www.aaai.org). All rights reserved.

SELECTIVE ABSTRACTION OF AI SYSTEM ACTIVITY

Jasmina Pavlin and Daniel D. Corkill

Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003

ABSTRACT

The need for presenting useful descriptions of problem solving activities has grown with the size and complexity of contemporary AI systems. Simply tracing and explaining the activities that led to a solution is no longer satisfactory. We describe a domain-independent approach for selectively abstracting the chronological history of problem solving activity (a system trace) based upon user-supplied abstraction goals. An important characteristic of our approach is that, given different abstraction goals, abstracted traces with significantly different emphases can be generated from the same original trace. Although we are not concerned here with the generation of an explanation from the abstracted trace, this approach is a useful step towards such an explanation facility.

I. Introduction: The Problem with Traces

Understanding the problem solving activities of a large knowledge-based AI system is often difficult. Simply tracing the activities quickly inundates an observer's ability to assimilate the many inferences and their relationships. Despite their unsatisfactory nature, activity traces remain a popular means of recording system activity because they are easily generated.

A *trace* is a chronological execution history of the system. It records the many primitive *events* that comprise the problem solving process. An example of a small part of a trace, containing the events arising from executing one knowledge source in the Distributed Vehicle Monitoring Testbed [2] is shown in Figure 1. Traces generated by the Testbed typically contain thousands of primitive events.

When investigating a particular system behavior, many events in the trace are unimportant or are meaningful only when considered with respect to other events. In addition, conceptually adjacent processing activities (for example, an activity that creates data used by another activity) can be quite distant in the trace. Understanding the system's behavior directly from its trace requires that the user weed out those events that are irrelevant to the question at hand and group salient events into a meaningful description of system activity. Even for the designer of the system, this is a tedious and time consuming task.

This research was sponsored, in part, by the National Science Foundation under Grant MCS-8306327 and by the Defense Advanced Research Projects Agency (DOD), monitored by the Office of Naval Research under Contract NR049-041.

```
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
Executing Node 2 --- Inv Ksis 4 -- Time Frame 4 -- Node Time 51
BLACKBOARD EVENT -> quiesence external receive
BLACKBOARD EVENT -> quiesence external send
INVOKED KSI ------> ksi:02:0004 46 s:gl:vl 51 (g:02:0037
                    g:02:0047 g:02:0052 g:02:0079)
                    (h:02:0016 h:02:0017 h:02:0018) {1917}
CREATED HYP ------> h:02:0019 vl ((3 (16 16))) 1 {1200}
SUPPORTING HYP ---> h:02:0016 gl ((3 (16 16))) 1 {600}
SUPPORTING HYP ---> h:02:0019 gl ((3 (16 16))) 2 {1200}
SUPPORTING HYP ---> h:02:0019 gl ((3 (16 16))) 3 {1200}
BLACKBOARD EVENT -> hyp-creation vl (h:02:0019)
INSTANTIATED KSI -> ksi:02:0017 goal-send:vt (g:02:0099)
                    (h:02:0019) <1200 -10000> {2194}
INSTANTIATED KSI -> ksi:02:0018 jf:vl:vt (g:02:0099)
                    (h:02:0019) <1200 2432> {1452}
UNSUCCESSFUL KSI -> jb:vl:vt g:02:0105 (h:02:0019) (nil nil)
                    -10000
INSTANTIATED KSI -> ksi:02:0019 ff:vl:vt (g:02:0105)
                    (h:02:0019) <1200 2432> {358}
RERATED KSI ------> ksi:02:0018 jf:vl:vt (g:02:0026 g:02:0059
                    g:02:0072 g:02:0089 g:02:0099) (h:02:0019)
                    <10000 2432> {1452 to 5985}
RERATED KSI ------> ksi:02:0012 s:gl:vl (g:02:0065 g:02:0100)
                    (h:02:0019 h:02:0020 h:02:0021) <1320 1032>
                    {1012 to 1072}
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
```

Figure 1: A Portion of A Trace.

What is needed is to automate the recognition, grouping, and potential deletion of traced activities in a manner that appropriately summarizes the behaviors under investigation. In this paper, we present an approach for selectively abstracting a trace based upon user-specified abstraction goals. An important characteristic of our approach is that abstracted traces with significantly different emphases can be generated from the same original trace. For example, an abstracted trace that emphasizes redundant processing activity might be quite different from one that emphasizes unsuccessful solution paths. The selective abstraction process is described, followed by an example and a presentation of additional issues related to the abstraction process. Relations with other work on presenting system activity is discussed in the last section.

II. Trace Nets and Abstraction Actions

We begin the selective abstraction process by transforming the sequence of primitive events in the trace into a trace net. A *trace net* is a data structure that records the input/output relationships among problem solving activities as well as their execution ordering. We represent a trace net as a Petri net [4] on which the execution history is imposed. Figure 2 shows a trace net for a small run of the Distributed Vehicle Monitoring Testbed. The portion corresponding to the trace fragment of Figure 1 has been indicated. Not all events in the original trace have been

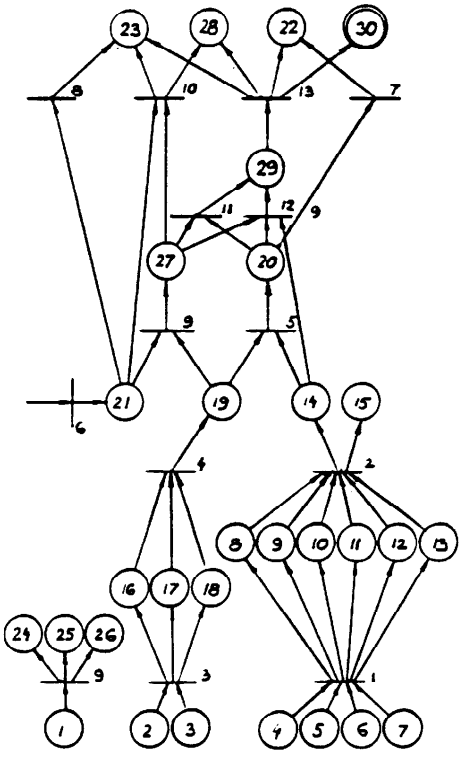


Figure 2: An Example Trace Net.

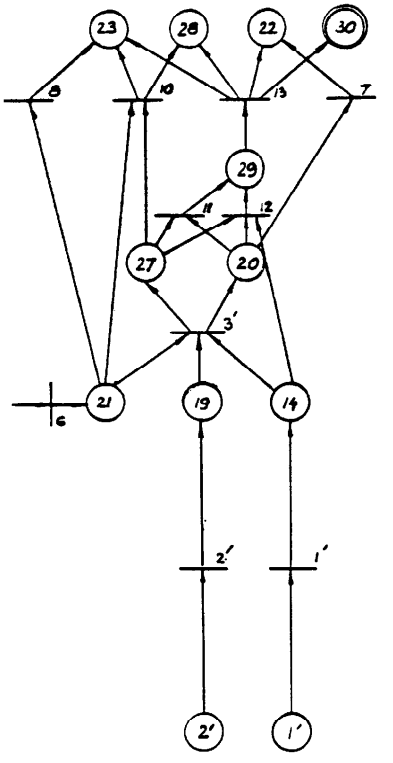


Figure 3: An Abstracted Trace Net.

represented in Figure 2. In the examples presented in this paper, we have chosen to represent only knowledge source executions and hypothesis creation. Trace events that are unrelated to this modeling level have been eliminated from the trace net.

Formally, a *trace net* T is a pair (N, E) where N is a Petri net structure $N = (D, A, I, O)$ with:

D	set of data units
A	set of activities
$I: A \mapsto D^*$	an input function connecting input data units to activities
$O: A \mapsto D^*$	an output function connecting activities to output data units

and where E is a partial order* over the set of activities called the *execution order*. A data unit might represent a single hypothesis or fact and an activity a single knowledge source execution or inference rule application.

An *abstracted trace net* $\hat{T}$ is then generated by appropriately collapsing and deleting portions of the original trace net T. The abstracted trace net contains a (generally) smaller set of data units and a (generally) smaller set of activities than T as well as a reduced (and possibly empty) execution order. The execution order of the abstracted trace net can be empty if the order in which activities execute is considered irrelevant. In an abstracted trace net a data unit might represent a group of hypotheses or facts and an activity a group of knowledge source executions or inference rule applications.

The following is a view of the overall abstraction process:

ABSTRACTION GOALS
⇓
ABSTRACTION PATTERNS
⇓
ABSTRACTION PREDICATES
⇓
ABSTRACTION ACTIONS
⇓
TRACE NET ⟶ *ABSTRACTED TRACE NET*

An *ABSTRACTION ACTION* either ***deletes*** an object* from the trace or ***lumps*** a group of objects into a single object. The *ABSTRACTION GOAL* determines which objects can be deleted and/or lumped. An example of an abstraction goal is "show redundant processing and the solution path." An object is deleted if it is considered unimportant with respect to the specified abstraction goal. A group of objects is lumped if it can be considered a single object with respect to the abstraction goal. In order to transform the abstraction goal into abstraction actions, the system needs to know what *ABSTRACTION PATTERNS* in the trace are relevant to the abstraction goal. For the redundant processing abstraction goal, important patterns are multiple activities creating the same output data units. Each *ABSTRACTION ACTION* is controlled by *ABSTRACTION PREDICATES* which are logical functions over the trace.

An example trace net and one of its abstractions are shown in Figures 2 and 3, respectively. A circle represents

* In a single-processor system, E is a total order.

* An object is either a data unit or an activity.

a data unit, a bar represents an activity, incoming arrows connect the activity to its input data units, and outgoing arrows connect the activity to its output data units.

We have found the following to be a relatively general and sufficient set of abstraction predicates. Corresponding actions are illustrated in Figure 4.

1. *unused-data-deleted?* If true, data which are not input to any activity (except for the solution) are deleted.
2. *no-output-activities-deleted?* If true, activities with no output data are deleted.
3. *shared-i/o-activities-lumped?* If true, two activities in which the first provides inputs to the second are replaced by a single activity.
4. *shared-input-activities-lumped?* If true, activities which share input data are replaced by a single activity.
5. *shared-output-activities-lumped?* If true, activities which share output data are replaced by a single activity.
6. *input-context-data-lumped?* If true, a group of data which are input to a single activity are replaced by a single data unit.
7. *output-context-data-lumped?* If true, a group of data which are output of a single activity are replaced by a single data unit.
8. *execution-order-ignored?* If true, activities can be lumped even if they are not successive in terms of execution order.

III. An Example: "Show redundant processing"

The following are the predicate values for the abstraction goal "show redundant processing and the solution path." The condition $condition_1$ is derived from the pattern for the redundant processing abstraction goal and is true when data are created by multiple activities. Since this pattern represents a relation between a data unit and its creating activities, $condition_1$ parameterizes predicates 1, 3, 5 and 7:

1. *unused-data-deleted?* : true, except for the solution and $condition_1$;
2. *no-output-activities-deleted?* : true;
3. *shared-i/o-activities-lumped?* : true, except for $condition_1$;
4. *shared-input-activities-lumped?* : true;
5. *shared-output-activities-lumped?* : false;*
6. *input-context-data-lumped?* : true;
7. *output-context-data-lumped?* : true, except for $condition_1$;
8. *execution-order-ignored?* : true.

Figure 3 shows the abstracted trace net obtained from the trace net in Figure 2 by performing the abstraction actions applicable with the above predicate values. The actions performed were (listed by type):

* Since predicate 5 is defined to cause the lumping only if this pattern is true, its value can be set to false, to eliminate testing.

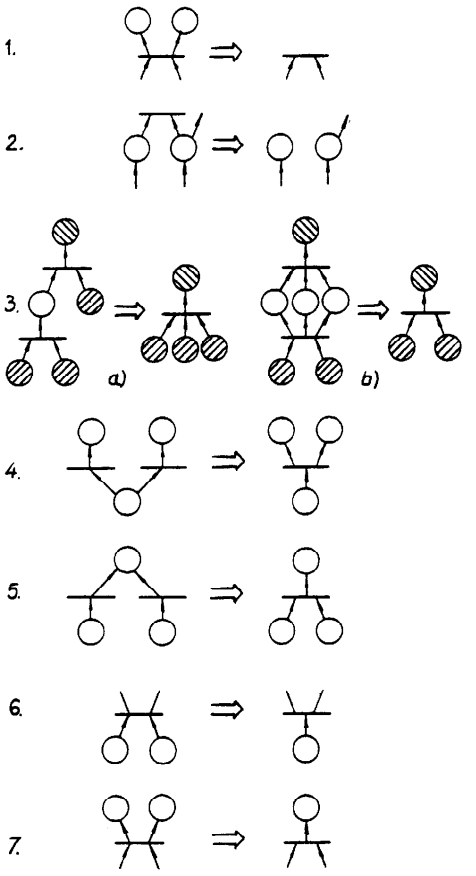


Figure 4: The Actions for Abstraction Predicates 1–7.

- *delete-data*: deleted data 1, 24, 25 and 26 because predicate 1 is true;
- *delete-activity*: deleted activity 9 because predicate 2 is true;
- *lump-data*: lumped data 2 and 3 into i' because predicate 6 is true; lumped data 4, 5, 6 and 7 into $2'$ because predicate 6 is true;
- *lump-activities*: lumped activities 1 and 2 into $1'$ because predicate 3 is true; lumped activities 3 and 4 into $2'$ because predicate 3 is true; lumped activities 5 and 9 into $3'$ because predicate 4 is true;

In this example, the abstracted trace has 12 data units and 10 activities—a reduction of 45% in the number of objects from the original trace. More importantly, only irrelevant information is abstracted out. All the relevant information is preserved; i.e., all of the activities generating redundant data can still be seen.

When considering an object for an action application, the predicates are evaluated in the listed order, except that predicate 8 is used as a constraint for all activity lumping predicates (3, 4 and 5). That is, if the value of predicate 8 is false, and the activity considered for lumping with the current activity is not the next in the execution order, then the action can not be taken. Predicates 1 and 2 are evaluated first because they cause deletion of

objects and thus result in less work for the remaining abstraction actions. Activity lumping predicates (3, 4, and 5) are considered before data lumping predicates (6 and 7) because they can either make a data lumping action unnecessary (see Figure 4 Case 3b) or make a new data lumping action possible. Such is the situation in Figure 4 Case 4 where the output data can be lumped after, but not before, the activity lumping action. Predicates 4 and 5 are related since both predicates must be true for a lumping action to occur if the candidate activity shares both inputs and outputs with other activities. Consequently, the false value of either predicate excludes the lumping action, and both predicates must be evaluated before the action can be taken. A similar relation (and the same evaluation order) holds for predicates 6 and 7.

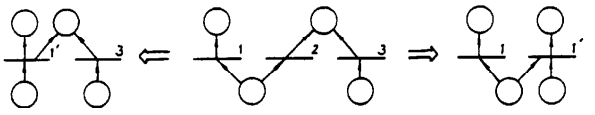
The abstraction process traverses the trace net from the top down (from the solution data to the system input data), iteratively evaluating all predicates and performing all applicable actions until quiescence. There are two reasons for our use of the top-down ordering. First, some abstraction goals are tied to the solution of the system (such as "show only the solution path"). Second, in interpretation systems (such as our Distributed Vehicle Monitoring Testbed), an activity typically has more input data than output data, and a top down traversal causes earlier lumping.

Abstraction predicates are general means of specifying the context of the abstraction action application, based on the structural properties of the trace net. Although they can be generated from abstraction goals, it is important for the user to have the ability to access the predicates directly. We view the transformation of abstraction goals into abstraction predicates as merely an aid to generating a suitable set of default predicate values—not as the sole means of specifying parameter values.

If the predicate values are allowed to be arbitrary logical functions, the result of two actions can depend on their ordering. Consider an example with three activities, the first two sharing inputs, and the last two sharing outputs. Predicates 4 and 5 have the following values:

4. *shared-input-activities-lumped?* : true, if all inputs are shared;
5. *shared-output-activities-lumped?* : true, if all outputs are shared.

The result of first considering lumping activities 1 and 2 is different from the result of first considering lumping activities 2 and 3 (see Figure 5). Before activities 1 and 2 are lumped, two lumping actions are applicable (activities 1 and 2; activities 2 and 3). After lumping activity 1, the conditions for lumping activities 2 and 3 are no longer true (not all the inputs are shared). Similarly, lumping activities 2 and 3 eliminates the possibility of lumping activities 1 and 2. If non-monotonic actions (where applying one action precludes applying another action) are specified through predicate values, additional action ordering should also be specified.



Left: Lumping activities 1 and 2.
Right: Lumping activities 2 and 3.

Figure 5: Order Dependent Lumping Actions.

IV. Adding Domain Dependent Information to the Abstraction Process

The abstraction process that we have outlined so far is completely domain independent. It has used as inputs only the information about the structure of the trace net (input and output connections and the processing order). However, there are cases where some domain dependent information can be used to improve the abstraction process. Domain-specific information may affect the following:

1. *Abstraction goals.* Some abstraction goals can be satisfied only if certain attributes of objects are known. For example, if the abstraction goal is to show whether the system was distracted during processing, a corresponding pattern is a sequence of activities in which there is a shift from processing "good" input data, to processing "bad" input data. In order to recognize this pattern, the abstraction process needs to know what constitutes "good" and "bad" data. Also note that this is one type of abstraction goal for which the execution order is important, and the predicate *execution-order-ignored?* must have the value true.
2. *Abstraction predicates.* Domain specific predicates can further reduce the amount of information in the trace. Consider a predicate that deletes a data unit of a smaller scope when an equivalent data unit of a larger scope are both inputs to an activity. The assumption here is that the smaller scope data contain redundant information. The notion of scope is domain dependent. For example, in our vehicle monitoring domain, the scope can be represented by the length of the track of a vehicle.
3. *Lumping mechanism.* The result of lumping may be sensitive to the type of objects that are being lumped. Consider, for example, the lumping of two activities where the output of the first is the input for the second in the vehicle monitoring domain. There are two types of activities: merging and synthesis [3]. A merging activity combines input tracks to produce a longer track. A synthesis activity combines lower level input data to obtain higher level output data. In the same vehicle monitoring example, an activity can combine acoustic signals into harmonic groups (signal-to-group synthesis), or it can combine harmonic groups corresponding to different acoustic sources associated with a vehicle type to identify the vehicle (group-to-vehicle synthesis). The lumping action for two merging activities is shown in Figure 4, case 3a. The lumping of two synthesis activities is shown in Figure 4, case 3b.

V. Capabilities of Our Approach

Our approach of reducing the trace net into an abstracted trace net results in several important capabilities:

- *On/off-line abstractions.* The abstraction process can be performed either during processing, with only a partial trace available, or after a solution has been found. The difference is that before processing is complete the full implications of activities and the relations among all their created data are not known. Thus, goals which depend on these implications can not be satisfied. At the end of processing, solution data can be marked as special type of data, which can be related to the abstraction goals.
- *Zooming.* The result of lumping is linked to the lumped objects. If the user decides that portions of the trace are "overabstracted," the abstraction of any lumped object can be restored to see more details. Similarly, more constraining predicates can be applied to portions of an "underabstracted" trace.
- *Highlighting.* When a class of abstraction patterns is defined in the system, it can be used not only to determine the predicate values but also to tag the instances of the patterns found in the trace net. These tags serve to highlight the patterns to the user (for example, by blinking on an output graphic device).
- *Iterative abstractions/feedback.* A uniform representation of the trace net and the abstracted trace net facilitates an iterative approach to goal satisfaction. If a particular abstraction goal does not sufficiently reduce the trace net, further abstraction goal refinement can be obtained from the user. In particular, at the beginning of an investigation the user may not know what abstraction goals appropriately abstract the "interesting" activities that occurred in the system. By using an initial abstraction goal to reduce the trace net, the user may be able to improve his understanding of the system's behavior to the point of selecting a more suitable abstraction goal. This iterative process of selecting an abstraction goal and viewing the resulting abstraction is a powerful investigative technique.

VI. Discussion

Perhaps the system that comes closest to our work is the GIST behavior explainer, which generates an explanation from the trace of a symbol evaluator [5]. The GIST behavior explainer has a single abstraction goal: the selection of interesting and surprising events, where the notion of interesting and surprising is domain dependent. The main focus of the behavior explainer is the generation of a natural English explanation, and the issues in generating natural language are different from issues in generating symbolic descriptions. For example, an important strategy in reducing the complexity of natural language explanations is to restructure the explanation so that the relationships being described are more easily comprehended [6]. Such presentation strategies are not an issue in our work.

Work on recognizing patterns of events as a tool for debugging distributed processing systems also has much in common with our approach [1]. However, we have taken the approach of displaying the whole abstracted trace net, rather than isolating patterns of activity and presenting them to the user. We feel patterns are best understood in the context of other events or in the context of the overall solution path.

The trace net to abstracted trace net transformation has been implemented and is being used in conjunction with the Distributed Vehicle Monitoring Testbed. Presently, the generation of abstraction predicates from an abstraction goal is not implemented, and the user must set their values directly. However, even its current state, the implementation is significantly improving our abilities to investigate problem solving activities in the Testbed. As we increase our experience with the selective abstraction process, we hope to automate the transformation of abstraction goals into abstraction predicates.

REFERENCES

[1] Peter Bates and Jack C. Wileden.
Event definition language: An aid to monitoring and debugging of complex software systems.
Proceedings of the Fifteenth Hawaii International Conference on System Sciences, pages 86–93, January 1982.

[2] Victor R. Lesser and Daniel D. Corkill.
The Distributed Vehicle Monitoring Testbed: A tool for investigating distributed problem solving networks.
AI Magazine 4(3):15–33, Fall 1983.

[3] Jasmina Pavlin.
Predicting the performance of distributed knowledge-based systems: A modeling approach.
Proceedings of the Third National Conference on Artificial Intelligence, pages 314–319, August 1983.

[4] James L. Peterson.
Petri Net Theory and Modeling of Systems, Prentice-Hall, 1981.

[5] Bill Swartout.
The GIST behavior explainer.
Proceedings of the Third National Conference on Artificial Intelligence, pages 402–407, August 1983.

[6] J. L. Weiner.
BLAH: A system which explains its reasoning.
Artificial Intelligence 15(1):19–48, September 1980.