Skeleton of a Supervisor for Enterprise Information Systems

James D. Jones[1], Steve Reames[2], and George Pandzik[1]
1 Computer Science
2 Management Information Systems
Angelo State University, 2601 W. Avenue N., San Angelo, Texas 76909
james.d.jones@acm.org,

Abstract. We describe the key elements of a working program which can assist in the overall management of the information resources of an organization. This program can reason about the relationships between components of an EIS, and concludes that a problem exists when those relationships become abnormal. This program can also reason about the behaviors of the components of an EIS, and concludes that a problem exists when those behaviors are abnormal.

1 Introduction

One of the ultimate goals of enterprise information systems (aka enterprise resource planning systems) is to develop and/or coordinate systems such that the conglomerate of the information resources of an organization works together for the good of the organization in the best of possible ways. This entails that information is timely and accurate, and more importantly, that the digestion and dissemination of that information occurs at the most appropriate of times in the most appropriate of ways.

To assist in this goal, we propose an architecture for a high level reasoning module that will sit on top of an enterprise information system (hereafter, EIS). Such a module can reason about the relationships among components within a system. It can also reason about the behavior of the components of a system. If either the behavior or the relationships vary from accepted norms, then a problem within the system is detected.

Please use the following format when citing this chapter:

Jones, J., D., Reames, S., Pandzik, G., 2006, in International Federation for Information Processing, Volume 205, Research and Practical Issues of Enterprise Information Systems, eds. Tjoa, A.M., Xu, L., Chaudhry, S., (Boston:Springer), pp.431-441.

We make the simplifying assumption that an EIS that is composed of many components. It is assumed that these components communicate with each other and with our program via message passing. (The actual mechanism by which our program receives information is immaterial.) The purpose of our program is to identify problems within the EIS. These problems can be with either the software or with the data. Problems are identified when messages conflict with each other, or when messages conflict with the program's beliefs.

1.1 Programming Tool: Answer Set Programming (SMODELS)

The basic strategy is as follows. An agent architecture is used to represent entities, their interrelationships, and their actions. In the example presented here, the only action is that of transmitting a message. These details are represented in a logic programming paradigm, called Answer Set Programming, which is more clearly specified as A-Prolog [1, 2]. A-Prolog can be viewed as a purely declarative language with roots in logic programming [3, 4], the syntax and semantics of standard Prolog , and in the work on nonmonotonic logic [5]. The inference engine used is SMODELS. This inference engine is aimed at computing answer sets (stable models) of programs of A-Prolog [6, 7]. This paradigm is very powerful in that it can represent multiple views of the world. This is important, because in the presence of uncertainty, the ability to postulate different ways that the world could be is very important. Not only can we represent different ways of perceiving the world, but we can also reason about those differences. Further, this paradigm allows the software to introspect with respect to its own beliefs.

1.2 Knowledge Representation Tool: Action Language Formalism

We are using specific action language formalism. This action formalism is very powerful in that it allows us to reason about prerequisites to actions, consequences of actions, co-requisites of actions, mutual exclusivity of actions, and sequences of actions. It also allows us to represent and reason about time. For example, things that are true in one moment of time, may or may not be true in another moment of time. Not only can we take time into account in our reasoning, but we can combine this with our introspective ability, and determine what was believed at some time in the past. We can also project into the future. This ability to predict is essential. In fact, one method of detecting problems is the fact that present observations may not match our earlier predictions. Further, predicting future actions allows us to identify potential opportunities for problems, allowing us to take proactive action.

To adequately represent and reason about these actions, we use the family of action languages [8, 9] originating with the action language A [10]. These languages encapsulate the significant issues broached by situation calculus [11], a formalism developed to deal with time-varying variables (called fluents.)

The structure of this paper is as follows. First, we present a very simple problem that the program is trying to solve. Next, we discuss elements of a working program to detect problems with regard to our simple scenario. This presentation is followed by a discussion of the results of the program.

2 Illustrative Problem

A very simple problem has been designed to demonstrate some of the techniques that will be followed. In this simple scenario, there are 3 entities. The only actions these entities can perform are communicating messages. (This is not a limitation; it serves only to simplify the problem.) A bias that is purposefully designed into the software is that it assumes that fewer problems are preferred over more problems. That is to say, when faced with competing explanations of the world, favor the explanation(s) in which there are fewer entities experiencing a problem. This matches our intuition.

3 Discussion of the Program

The program has the following structure: the objects of the domain, the background theory of the domain, and general rules of inertia. The objects of the domain are: time periods, and their general relationships to each other; entities; messages transmitted by the entities; fluents (time-varying variables); and actions that can be performed. The background theory of the domain consists of dependencies among the fluents, and causal laws. Causal laws state the impact of actions. The general rules of inertia are used to define what remains constant between time periods, and to define what changes between time periods.

3. 1 Description of objects in the domain

The objects of the domain are: the time periods involved, the entities, the messages the entities transmit, valid truth values, fluents, and the actions that can be performed. The first statement below defines the number of time periods that will be reasoned about. The second statement defines those time periods as integers between 0 and n. The third statement defines a total ordering upon the time periods, and defines what time periods are next to other periods.

```
const n=10.
time(0..n).                                    1)
next(T1,T2) :- time(T1),                       2)
      time(T2),
      T2 = T1 + 1.
```

This last statement defines that time period T2 immediately follows time period T1. Time T1 is next to time T2 if both T1 and T2 are time periods, and time T2 equals time T1 + 1 time unit. Note that in this definition, next is not symmetric. That is, while T1 is next to T2, the reverse is not true. (Next means "subsequent", not "adjacent".)

Other objects in our domain which we need to represent include the entities we are reasoning about, the messages that are transmitted.

```
entity(e1).                                    3)
message(m1).                                   4)
```

In the process of transmitting the message, the entity also relates his/her belief in the truth of the message. For example, "the forecast for today is for severe thunderstorms, but I don't believe it will rain at all." Therefore, in order for a communicator to express belief or disbelief in a proposition, true values need to be objects of our domain. Those truth values are defined with statements such as the following.

```
truth_value(1).        %% 1 represents "true"   5)
```

Background knowledge is given by a collection of domain dependent fluents. A fluent is a time-varying variable. An example would be something like the president of the United States. This same term has different denotations at different moments of time. For instance, reagan = president(usa) can be true or false depending on time. If this question is asked with respect to 1980, the answer would be yes. If this question is asked with respect to 1990, the answer would be no.

In the next three rules, inertial fluents are being described. Domain dependent fluents do not have to be inertial, but are described as such for simplification. A fluent is inertial if its truth value stays the same unless it is changed by an action. For example, if the light is on, it remains on until an action changes that fluent. Such actions might include: turning the light off, turning the circuit off, or a power outage. As example of this definition is the following.

```
fluent(i,f1).                                  6)
```

A fluent is described by two parameters, as in fluent(i,f1). The first parameter identifies the type of fluent. The second parameter identifies the fluent itself. There are two values for type of fluent: i means the fluent is inertial, and n means the fluent is not inertial. We described inertial fluents above. A fluent is not inertial if it is true only for the time period in which an action affected it. For example, if at time T1 my daughter said that she loved me, the fact that that message was uttered is true only for time T1. It is not true that she uttered that at time T2 UNLESS she uttered it again.

This example also demonstrates another issue. With the utterance of that message, two fluents were affected. I learned something about my domain from the

message: the fact that my daughter loves me. This is an inertial fluent, and would remain true through all successive time periods unless an action changes that. The second fluent affected by this example is the fact that the message was uttered. This is a non-inertial fluent, and is true only in time T1, unless it is uttered again.

The fluent described next, received(M,A,Truth_value), says that the system received a message M from entity A with some truth value. That is, entity A says that message M is true (or conversely, false). A Truth_value of 1 means true, and a Truth_value of 0 represents false. Notice that this fluent is not inertial. Receiving a message at time T does not imply receiving the same message at time T+1.

```
fluent(n,received(M,A,1)) :-  entity(A),                    7)
                              message(M).
```

The example presented here is very simple, and is used to illustrate our approach. The example consists merely of entities transmitting messages. At this point in the discussion, it would be natural to expand the domain by defining actions which would allow entities to achieve some collection of goals. A general scenario would be to plan and execute actions to achieve those goal(s), and to modify those plans according to information received from the entities. It also would be reasonable to allow us to request certain types of information from the entities, etc. For the moment, these actions and goals will be ignored. For the purposes of the example, only one action will be introduced. The following rule defines that action as that of transmitting a message. This rule states that entity A transmits message M with truth_value V. Stating this in the language of fluents, A says that fluent M has truth value V.

```
action(issue(A,M,V)) :- entity(A),                          8)
                        message(M),
                        truth_value(V).
```

3.2 Background theory of the domain: dependencies among fluents

The foregoing discussion concludes the presentation of those types of rules which define the objects of the domain. The second major portion of the program is the background theory that is needed to reason and act in this environment. This additional background consists of: (a) dependencies between fluents, and (b) dynamic causal laws. Some dependencies will be domain dependent, and others will be domain independent. Dynamic causal laws describe actions that may be performed in the environment. They specify prerequisites to actions, consequences of actions, actions that may be performed in parallel, and actions that are mutually exclusive with each other. The following two rules specify the domain dependent dependencies for the example.

```
holds(m2,T,0) :- time(T),                                   9)
```

```
        holds(m1,T,1).

holds(m2,T,0) :- time(T),                                    10)
        holds(f1,T,1),
        holds(m3,T,1).
```

The basic form of the holds predicate is holds(Fluent, time, Truth_value). Hence, holds(m2,T,0) means that fluent m2 is false at time T. Similarly, holds(m1, T, 1) means that fluent m1 is true at time T. The first rule states a dependency between two fluents: fluent m2 and fluent m1. It states that m2 and m1 CANNOT be true at the same time. The second rule creates a dependency between three fluents: m2, f1, and m3. This rule states that at a point in time, if f1 is true, and if m3 is true, then m2 MUST be false.

The next fluent dependency rule is the following:

```
holds(M,T,V) :- time(T),                                     11)
        entity(A),
        message(M),
        truth_value(V),
        holds(received(M,A,V),T,1),
        not holds(problem_entity(A),T,1).
```

This rule states that the information received from entities that are not known to be problem entities is to be believed. The form of negation here, not, is weak negation. It is not known for a fact that entity A is not a problem entity. However, there is not any evidence to believe that entity A is a problem entity, therefore that entity is to be given the benefit of the doubt. To not do so would place us in a state of paralysis: we would never act until we were absolutely certain of all the facts. Unfortunately, real life rarely is so simple.

Note that according to rule 21, we believe precisely what the entity informs us of. In the example just given about rain, the entity informed us that he/she/it does not believe the message. This is signified by the variable V in the formula holds(received(M,A,V),T,1) of rule 21. Looking at the head of the rule 21, this same V is the truth value we assign to the fluent representing the message. In this case the meaning is that we adopt the entity's belief as our own belief.

Rather than believing what an entity transmits, we could take the posture of doubting what an entity transmits. Here is an opportunity for greater generalization. In what kinds of situations should information from an entity/person/source should be distrusted? Perhaps the person (entity) is unqualified to speak about a subject. So, strictly speaking, there is not a problem with the entity, yet there is reason to not believe the message without further investigation.

3.3 Background theory of the domain: dynamic causal laws

We have been discussing the lengthy and very important topic of background theory needed to reason and act in this environment. We mentioned that there are two broad components to this: dependencies between fluents, and Dynamic causal laws. We have just finished discussing dependencies between fluents, and will now discuss dynamic causal laws. Recall that dynamic causal laws deal with the actions of the domain, fully specifying their impacts and their relationships to one another. Normally, it is this section that is the most important and most lengthy of the two. However, since there is only one action in the example, this incredibly important section may seem trivial.

```
holds(received(M,A,V),T2,1) :- next(T1,T2),                    12)
                    entity(A),
                    message(M),
                    truth_value(V),
                    occurs(issue(A,M,V),T1).
```

If the message is issued at time T1 then it will be received at time T1+1. The occurs predicate means that the action occurred at time T1.

3.4 Laws of Inertia

The rules of inertia are used to define what fluents remain unchanged from one time period to the next. Inertial fluents remain unchanged unless they are specifically impacted by one or more actions. These rules are a standard part of any action theory. One of the rules of inertia is the following.

```
holds(F,T2,1) :- next(T1,T2),                                  13)
        fluent(i,F),
        holds(F,T1,1),
        not holds(F,T2,0).
```

3.5 Program execution: rules for detecting problems

The following two rules are used to discover problems,. The first rule declares that it is possible for any entity to be a problem entity. The second rule states that it is desired to conclude that an entity is a problem entity only if we are forced to conclude this by the facts of the program.

```
{holds(problem_entity(A),0,1) : entity(A)}.                    14)
minimize{holds(problem_entity(e1),0,1),holds(problem_entity(e2),0,1),  15)
        holds(problem_entity(e3),0,1)}
```

The first rule is a choice rule. Roughly speaking {p(X): q(X)} says that an answer set A of the program containing this rule may contain an arbitrary subset p of q, i.e. an arbitrary collection of atoms p(t1),...,p(tn) such that for every i, q(ti) is in A. For instance, a program

```
{p(X) : q(X)}
q(a)
```

has answer two sets {q(a)} and {q(a),p(a)}. In the case of the first answer set, the arbitrary subset chosen was the empty set, hence only q(a) appears in the answer set. In the case of the second answer set, the arbitrary subset p of q chosen was p(a). There are no other arbitrary subsets. Choice rules can be viewed as shorthand for a fairly large number of normal rules. They allow shortening the program and thereby improving its efficiency.

4 Program Execution

This final section shows a series of executions of the program. At the end of each execution, the program will print out the fluents that are true. The execution of the program starts with a very simple case, and produces very clean, intuitive results (Execution 1). Building upon this example, a contradiction will be introduced, which suggests problems (Execution 2). However, the source of problems is not clear, so multiple models of the world will be maintained to account for the uncertainty that is introduced. This ability to represent multiple views of the world is unique among the techniques of artificial intelligence, and is provided solely by the semantics of logic programs.

This execution will then be expanded with additional information which allows the software to very clearly identify the source of problems (Execution 3). This additional information illustrates nonmonotonic reasoning: in step 2 it will be believed that one of two entities are equally likely to be the problem; in step 3 that previously held belief that one of the entities is possibly the problem will be withdrawn .

4.1 Execution 1: E1

Let us say that initially fluent f1 is true, and fluent f2 is false. The values of other fluents are unknown. Our starting values are stated as follows.

```
holds(f1,0,1).
holds(f2,0,0).
```

The program will execute for 1 time period and display the fluents which are true at that time. Let us say that E1 = "entity e1 says that message m1 is true". This is denoted by the following rule:

```
occurs(issue(e1,m1,1),0).
```

This rule states that at time 0, it happened (occurred) that entity e1 issued message m1. Entity e1 believes that message m1 is true. If the program is run, the following will be displayed.

```
Stable Model: holds(m1) holds(f1) holds(received(m1,e1,1))
```

holds(m1) is true by rules 12 and 11. holds(f1) is true by rule 14. holds(received(m1,e1,1)) is true by rule 12. This result is as expected. This basically says that what was true at the beginning (f1) is still true, a message was communicated, and that message was received.

4.2 Execution 2: E2

Now, let us try a different execution, E2. E2 will be the same as E1, except that it has an additional action: "e2 says that m2 is true". So, the rules to embody this execution are as follows:

```
holds(f1,0,1).
holds(f2,0,0).
occurs(issue(e1,m1,1),0).
occurs(issue(e2,m2,1),0).
```

According to rule 9, message m1 and message m2 CANNOT be true at the same time. Hence, either entity e1 is wrong, or entity e2 is wrong.

It is expected to learn that either entity e1 or entity e2 is a problem entity. The software will in fact return two models: one in which entity e1 is a problem entity, and the other one in which entity e2 is a problem entity. This is precisely what would happen in the real world. We are faced with a contradiction (m1 and m2 cannot both be true), and lacking any more information, the strongest position one could logically take is to believe that at least one of the entities is a problem entity. The software has performed as expected and desired.

4.3 Execution 3: E3

Let us consider yet another execution of this program, E3. E3 will be the same as E2 with the addition that "e3 says that m3 is true". The rules that would embody this execution are:

```
holds(f1,0,1).
holds(f2,0,0).
occurs(issue(e1,m1,1),0).
occurs(issue(e2,m2,1),0).
occurs(issue(e3,m3,1),0).
```

In the previous execution, E2, there was a contradiction, and there was equal reason to believe that either entity e1 or entity e2 was a problem entity. There was no additional information to solidify our suspicions, therefore we were left in the position of believing that either one of them could be a problem entity, but we could not determine which one with any degree of certainty.

In another execution of the program, consider that we have the same information as before, but with the additional information that "e3 says that m3 is true". The same contradiction exists as before, yet it can now be firmly determined which entity is the problem. Rule 10 says that if f1 and m3 are true, then m2 cannot be true. This is combined with rule 9 that says that m1 and m2 cannot both be true. Hence, it is now believed (without contradiction) that entity e2 is a problem entity.

5 Expanding This Work

We could log observations to endow the module with the ability to learn patterns of communication for the various entities. This might help identify intermittent problems vs. persistent problems. This might also lead to proactive solutions if certain causes for past problems recur.

We should expand our list of actions. A very complex arsenal of actions which interrelate with each other would be interesting. We could make predictions about the consequences of actions, and observe whether those predictions came true. A significant arena of enhancement would be in defining what constitutes a "problem". That is, forms of problems in addition to inconsistencies (and discrepancies between predictions and observations) need to be identified, along with appropriate solutions. As an example, we could model and analyze more deeply the relationships between components. Abnormalities in those relationships could be considered problems. In the scenario here, a problem was defined as an inconsistency caused by a message. An example of a problem in which there is not an inconsistency would be a situation where there is an increase in the purchase of raw materials at the same time in which there is a reduction in cash flow (thereby exacerbating the cash flow problem.)

References

1. M. Gelfond and V. Lifschitz, Classical Negation in Logic Programs and Disjunctive Databases, New Generation Computing, 9, (Ohmsha, Ltd. and Springer-Verlag, 1991) pp. 365-385
2. M. Gelfond and V. Lifschitz, The Stable Model Semantics for Logic Programming, Proceedings of ICLP 88, (1988), pp. 1070-1080.
3. R. Kowalski, Predicate Logic as a Programming Language, *Information Processing* **74**, 569-574 (1974).
4. R. Kowalski, Logic for Problem Solving (North-Holland, 1979).
5. R. Reiter, A Logic for Default Reasoning, *Artificial Intelligence* **13** (1/2), 81-132 (1980).
6. I. Niemela and P. Simons, Smodels - An Implementation of the Stable Model and Well-Founded Semantics for Normal Logic Programs, 4th International Conference on Logic Programming and Non-Monotonic Reasoning, (1997), pp. 420-429.
7. I. Niemela and P. Simons, Extending The Smodels System with Cardinality and Weight Constraints, In J. Miner, editor, Logic Based AI, (Kluwer, 2000), pp. 491-522.
8. C. Baral, M. Gelfond, and A. Provetti, Reasoning About Actions: Laws, Observations and Hypotheses, *Journal of Logic Programming* **31**, 201-244 (1997).
9. C. Baral and M. Gelfond, Reasoning about Effects of Concurrent Actions, *Journal of Logic Programming* **31**, 85—118 (1997).
10. M. Gelfond and V. Lifschitz, Representing Actions and Change by Logic Programs, *Journal of Logic Programming* **17**, 301-323 (1993).
11. M. Gelfond, V. Lifschitz, and A. Rabinov, What Are the Limitations of the Situation Calculus? in S. Boyer (Ed.), Automated Reasoning, Essays in Honor of Woody Bledsoe (Kluwer Academic Publishers, 1991), pp. 167-181.