Appears in *Proceedings of the Tenth Conference on Innovative Applications of Artificial Intelligence*,
AAAI Press, 1998, 1087-1093

ANSWER: Network Monitoring Using Object-Oriented Rules

Gary M. Weiss*, Johannes P. Ros and Anoop Singhal

AT&T Labs
480 Red Hill Road
Middletown, NJ 07748
{gmweiss, hros, anoopsinghal}@att.com

Abstract

This paper describes ANSWER, the expert system responsible for monitoring AT&T's 4ESS switches. These switches are extremely important, since they handle virtually all of AT&T's long distance traffic. ANSWER is implemented in R++, a rule-based extension to the C++ object-oriented programming language, and is *innovative* because it employs both rule-based *and* object-oriented programming paradigms. The use of object technology in ANSWER has provided a principled way of modeling the 4ESS and of reasoning about failures within the 4ESS. This has resulted in an expert system that is more clearly organized, easily understood and maintainable than its predecessor, which was implemented using the rule-based paradigm alone. ANSWER has been deployed for more than a year and handles all 140 of AT&T's 4ESS switches and processes over 100,000 4ESS alarms per week.

Introduction

Network reliability is of critical concern to AT&T, since its reputation for network reliability has taken many years to establish and is one of its most valuable resources. Nonetheless, in today's fiercely competitive environment, high levels of reliability must be achieved in a cost-effective manner. This paper will focus on the problem of efficiently monitoring the 140 4ESS switches in the AT&T network, which collectively handle virtually all of AT&T's long distance traffic.

Problem Description

Upon detecting a problem or experiencing an anomalous event, a 4ESS switch will generate an alarm and send it to one of AT&T's two technical control centers for further processing. At these sites, an expert human "analyst" examines the alarm, possibly runs diagnostics on the switch and then determines if human intervention is required. If intervention is required, the analyst creates a "trouble ticket" and electronically dispatches it to a technician at the site housing the switch. This maintenance process requires the analysts at the technical control centers to monitor over 100,000 alarms per week. Many of these alarms indicate a transient problem—a problem that requires further monitoring but does not require immediate human intervention. Thus, a maintenance process that requires each alarm to be handled by an analyst will be extremely time-consuming, costly and wasteful. Nonetheless, for many years the maintenance process operated in such a manner.

The *problem* is to provide a monitoring system that partially assumes the role of the analyst, so that only those problems that require further analysis are brought to the analysts' attention. Such a system will reduce staffing costs and allow the analysts to focus their attention on the more difficult problems. Specifically, such a monitoring application should correlate alarms coming from different components, ignore transient problems, identify chronic problems, and run diagnostics when appropriate. If a problem is isolated that can only be fixed by a technician, then a trouble ticket should be autonomously generated and electronically dispatched to a field technician. Only when further analysis of problem is required will an "alert" be generated and sent to an analyst at one of the technical control centers. Thus, the monitoring system must provide *filtering* of the 4ESS alarms and intelligent work delivery. In addition, the monitoring application must operate in a manner that allows for changes and updates to be handled easily and quickly as the AT&T network continues to evolve. This objective became a priority due to deficiencies in previous solutions, and is a key focus of this paper. Finally, one overriding requirement on any solution is that it must be capable of processing the alarm messages in real-time.

Why Use AI?

The decision to use an expert system to perform network monitoring and alarm filtering was a very natural one, given the need to automate a task performed by human a expert within a limited domain. Furthermore, the fact that the task to be automated is essentially a diagnosis task lends even more support to this decision, since diagnosis is amenable to AI techniques and such problems have been widely solved using AI methods. In fact, there are many examples within the telecommunication industry of such systems. NYNEX uses the MAX expert system (Rabinowitz, Flamholz, Wolin & Euchner 1991) to locate

*Also Rutgers University

problems in the local loop (the segment connecting each subscriber to the central office), Pacific Bell uses the trouble locator system (Chen, Hollidge, & Sharma 1996) to locate troubles in a local cable network and AT&T uses the Scout system (Sasisekharan, Seshadri, & Weiss 1996) to identify recurring transient network faults.

A variety of AI techniques could be used by an expert system to perform network monitoring and alarm filtering. Machine learning methods are of particular relevance for diagnosis and are the basis of the trouble locator system, which employs a causal network and the Scout system, which employs data mining techniques. However, a rule-based expert system was deemed most appropriate since it was believed that in some cases our problem required highly specific domain knowledge—knowledge available from the 4ESS analysts in the form of "rules". It should be noted, however, that there have been several recent efforts to supplement the knowledge in the rule-based system with knowledge induced via machine learning methods (Weiss & Hirsh 1998; Weiss, Eddy, Weiss, & Dube 1998).

Previous Approaches

Originally, no alarm filtering was provided and the analysts had to operate on the raw alarms directly. A system to help monitor the 4ESS switch was first implemented in C. Due to software maintenance problems with this approach, in 1990 this system was redesigned and reimplemented using C5, a C version of the popular OPS5 rule-based programming language (Cooper & Wogrin 1988). However, as additional rules were added to this new 4ESS-ES (4ESS expert system) in response to changes in the network, this system became less maintainable. We believe this resulted due to the lack of a clearly defined domain model and the widespread use of fairly shallow, ad-hoc, rules. Similar problems have been observed in other "first-generation" expert systems. Thus, this system did not satisfy our ease of maintainability criterion.

Application Description

ANSWER (Automated Network Surveillance with Expert Rules) is a complete operation support system (OSS) for maintaining the 4ESS switches in the AT&T network. In this paper we focus on the expert system component, which is the central and most important part of the application. The remaining part of the application is primarily concerned with allowing the expert system to interface with its environment (e.g., collect messages from the 4ESS switches). For the purposes of this paper, ANSWER will refer to the 4ESS expert system component. Each 4ESS switch is handled by its own instance of ANSWER; thus, since there are 140 4ESS switches, there are 140 instances of ANSWER. These instances of ANSWER run on a single HP T520 server with 4 GB of RAM.

Functional Overview

ANSWER is responsible for the operation support system's intelligent behavior. The basic input/output functionality of ANSWER is shown in Figure 1.

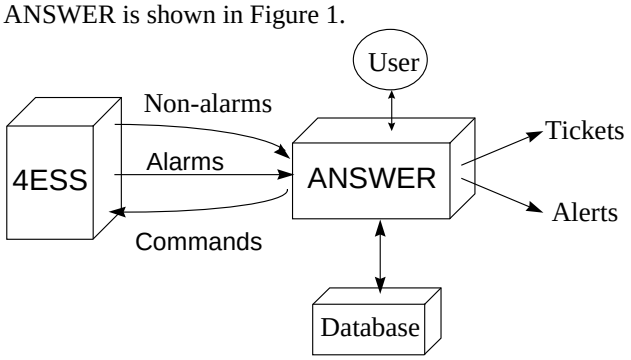

Figure 1: Functional View of ANSWER

The key inputs to ANSWER are alarm and non-alarm messages from the 4ESS switches. The alarm messages provide information about problems detected by a specific device within the 4ESS while non-alarm messages provide status information or the results of diagnostics previously requested by ANSWER. The key outputs of ANSWER are alerts to the analysts at the technical control centers and tickets to the on-site technicians. ANSWER can also send commands to the 4ESS to run diagnostics. Users can interact with ANSWER to retrieve information, such as the status of a 4ESS component, and to customize the behavior of ANSWER. A database provides long-term persistent information storage, so that users can examine the past history of the 4ESS switch. The database is also used to store information required by ANSWER, including information specific to each 4ESS. The rules in the expert system are responsible for determining ANSWER's behavior when receiving a new alarm.

The Object Model

One of the key advantages and distinguishing characteristics of ANSWER is that, in addition to using rule-based programming, it also uses object-oriented technology—a technology now in widespread use in industrial applications. For object oriented technology to be useful in this context, there must be a way for ANSWER to model the 4ESS as a collection of objects. A simplified version of the object model is shown in Figure 2, using Rumbaugh's Object Modeling Technique (Rumbaugh, Blaha, Premerlani & Eddy 1991).

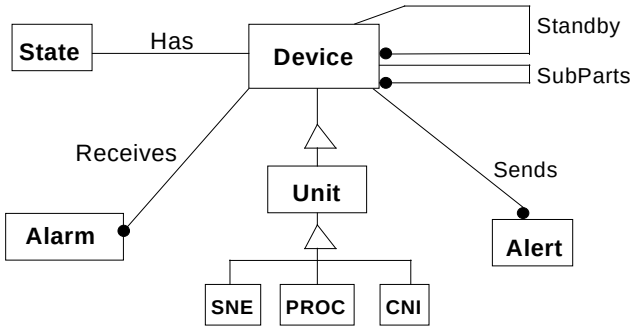

Figure 2: The 4ESS Object Model

The 4ESS is readily viewed as a collection of devices and consequently *device* is the central object in the 4ESS object model. The object model is described by the following:

- *subparts*: an aggregation relation on device that allows any device (including the 4ESS itself) to be viewed as a hierarchical collection of devices
- *standby*: a one to many relation on device that specifies the devices able to take over for a device if it fails
- *state*: the state of a device (e.g., in-service) and the time it entered that state
- *alarm*: each device has an ordered list of alarms, which contains the alarms generated by the 4ESS on behalf of the device
- *alert*: alerts are associated with a device and are created when ANSWER decides the device requires further analysis, by a human

In addition, Figure 2 specifies an inheritance hierarchy for the device class. Device is intended to represent a relatively abstract object. The device class has a subclass named Unit which represents a generic 4ESS device. Finally, the unit class has many subclasses, of which only three are shown. These subclasses represent either a very specific 4ESS hardware component (e.g., PROC represents the main processor) or a specific class of equipment that shares many common characteristics. Inheritance allows us to share and/or specialize device behavior, as appropriate. By having an object-oriented language integrated with a rule-based language, we inherit not only methods and data members (i.e., functions and variables), but also rule-driven behavior. For example, ANSWER's PROC class contains only rules which specify behavior unique to processor devices—generic device behavior is inherited from the unit and device classes.

The Device Model and Model Instantiation

Devices are the key object in our model and therefore it is important to understand how these objects are used, and in particular, how they are created. A key requirement for our model is that it be *dynamically* built from the information sent to it from the 4ESS. There are two reasons for this requirement: flexibility and efficiency. The flexibility of a dynamic model arises from the fact that no up-front configuration information is required—which is essential since each 4ESS switch is unique and components are continually added and removed. The second advantage of a dynamic model is that it permits us to model only the components which have abnormal activity, thereby reducing the size of the model and thus realizing time and space savings.

The expert system is primarily driven by two types of events: 4ESS (alarm and non-alarm) messages and one-second timer ticks. Each 4ESS message refers to a device by specifying up to three levels of device information (i.e., the 4ESS device hierarchy is at most 3 levels deep, excluding the 4ESS itself). At each level in the hierarchy, a device type is specified, along with an integer-valued device identifier, to ensure that each device within a 4ESS switch is unique. The timer ticks update the expert system's internal clock and drives all of its time-based behavior. A device model, which is a "snapshot" of the devices in the object model, is shown in Figure 3. This figure will be used to describe the device creation process.

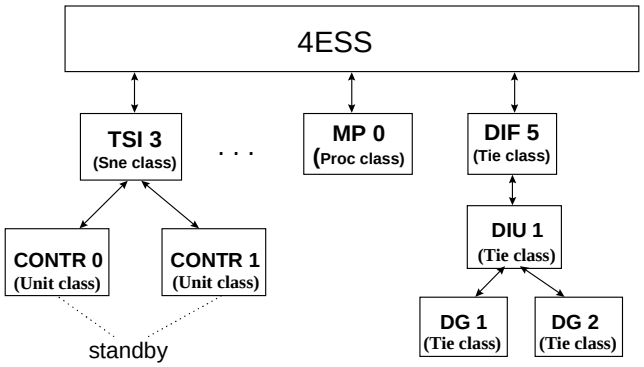

Figure 3: A Device Model

When the expert system receives an alarm and the device specified in the alarm doesn't already exist in the model, then it is created. The class of the device object is determined by a table lookup. For example, when a TSI:3 device is created, an object of class SNE is created. The model requires that all "ancestors" of this device (i.e., devices that contain this device) be present in the model, so if they do not already exist in the model, they are created. For example, when ANSWER receives an alarm for the device "DIF:5 DIU:1 DG:2" it will first attempt to create the DG:2 object. However, if the DIU:1 or DIF:5 of which this DG:2 is a part do not already exist in the model, then they will be created first. When devices are created, data driven rules check for "invariant" relations and create them in the model. For example, if "TSI:3 CONTR:1" is created and "TSI:3 CONTR:0" already exists in the model, then a *standby* relation will be formed between these two objects (controllers for the same device can take over for one another). Device deletion is driven by timer events; a device is deleted if it is in the "in-service" state and has remained in that state, without experiencing any problems, for a specified period of time.

Alternative Solutions

By 1992 it became clear that object-oriented technology would improve the maintainability of our rule-based expert systems. In 1993 a formal process was defined for evaluating existing object-oriented languages that provided data-driven rules. This process involved defining required and desired language features[1] and then creating a "representational benchmark", a set of mini-problems

[1]Required features included the ability to represent a taxonomy of objects and to have typed slots; desired features included the ability to put procedural code in the left-hand side of a rule.

similar to those we encounter when constructing systems that must represent large, physical models (like a model of a telecommunication network element). The evaluation process involved requesting vendors of 10 object-oriented languages with data-driven rules to implement these benchmarks. These 10 languages could be classified into three categories: "rules in C++" (Ilog rules, CLIPS++, Rete++, RAL), other "rules + objects" languages (Kappa, ART Enterprise, G2, Nexpert Object) and AT&T internal languages (C5, MCL). The vendor's returned the implementations and the languages were evaluated based on the implementations and the language features. Subsequently, however, the importance of integration with C++ was raised in priority and the focus then became limited to the "rules in C++" languages. Only one language came close to satisfying all requirements, but this rule language lacked the required ability to handle inter-object pointers. The vendor of the language was requested to enhance the language, but stated that the language could not be easily extended to add this capability. As a consequence, a decision was made to develop a rule-based extension to C++.

R++: A Rule-Based Extension to C++

R++ is a relatively small extension to the C++ language which adds object-oriented rules. R++ rules are simply another type of C++ member function and share the object-oriented properties of C++ member functions: inheritance, polymorphism and dynamic binding. R++ is implemented as a preprocessor which runs before the C++ compiler is invoked. Figure 4 shows a partial declaration of ANSWER's Device class. Note that each device's *sub_parts* and *standby* devices are represented as a set. The optional *monitored* keyword in the class declaration identifies data members which may trigger rule evaluation. The declaration also includes the declaration of the *link_standbys* rule, which ensures that the standby relation is always kept up-to-date (a standby relation exists between two devices if one is capable of taking over for the other).

```
class Device {
protected:
  String                        type;          // type of device(e.g., TSI)
  int                           number;        // uniquely identifies device
  monitored State               *state;        // ptr. to device's state info.
  monitored Alarm               *new_alarm;    // ptr. to newest alarm
  monitored Device              *part_of;      // ptr. to containing device
  monitored Set<Device>         sub_parts;     // set of sub-parts devices
  monitored Set<Device>         standby;       // set of standby devices
rule link_standbys;
};
```

Figure 4: Declaration of Class "Device"

Rules have a special *if-then* syntax, where the if (antecedent) and then (consequent) parts are separated by an arrow (⇒). The rule in figure 5 can be translated as: *if, for this device, there is a new alarm and the state of a standby of this device is out-of-service, then send an alert.*

```
// alert if get alarm and standby out of service
rule Device::alert_if_standby_out_service
{
   new_alarm &&
   Device *stby = standby &&
   State *st = stby→state &&
   st→name == "out-of-service"
⇒
   send_alert("standby_oos, this, stby);
};
```

Figure 5: Alert if Standby Out of Service R++ Rule

R++ also provides the ability to write rules which operate on container classes, like sets and lists. Using an R++ feature called "branch binding", a rule can be applied to each element of the container (the at-sign "@" is the branch-binding operator). The rule in Figure 6 relies on the fact that the list called *Devices* contains all of the devices in the model. For those not familiar with C++, the variable *this* always refers to the object itself—in the case of the rule in Figure 6, the device on whom *link_standbys* is being performed. The antecedent of the rule in Figure 6 can be translated as: *for each device dev in the set Devices, check if dev is the standby of this device.* The consequent then updates the standby field of each device to reflect the new standby relationship.

```
// link standby devices together
rule Device::link_standbys
{
   Device *dev @ Devices &&
   is_standby(dev)
⇒
   this→add-standby(dev);
   dev→add-standby(this);
};
```

Figure 6: Link Standby R++ Rule

The key difference between rules and ordinary C++ member functions is that changes to data members in the antecedent of the rule automatically causes the rule to be evaluated and, if the antecedent evaluates to TRUE, then the consequent is executed. Thus, rules are *data-driven*. In the example in Figure 5, whenever a new alarm is received on a device or the state of one of its standby devices changes, the rule will be automatically reevaluated. The key difference between R++ rules and rules in other rule-based languages is that R++ rules are *path-based*. This means that the antecedent of an R++ rule can only include data members which the class has access to—typically through a pointer reference. Even though R++ rules are therefore less expressive than rules in other languages (because they cannot reference arbitrary objects), we consider this an *advantage* because it ensures that R++ rules *respect the object model.* Finally, R++ is very

efficient and in most cases is far more efficient that other rule-based systems, due to the path-based nature of its rules. For those interested in a more in-depth understanding of R++ or in receiving a copy of R++, see the R++ home page (Patel-Schneider, 1998).

Uses of AI Technology

AI technology plays a central role in ANSWER. R++ rules declaratively encode domain knowledge and C++'s object-oriented language features allow the 4ESS switch to be modeled as a hierarchical collection of devices. Although object technology is not commonly associated with AI, it nonetheless can be considered an AI technology. In fact, most of the key ideas behind object-oriented languages have come from AI (e.g., frame-based languages). R++ integrates these two complementary technologies and thus allowed us to get the benefits of each.

AI has always been interested in problem solving. In our case, the problem is related to diagnosing 4ESS faults. The AI problem solving techniques we use include abstraction and a primitive form of model-based reasoning.

Use of AI Techniques for Diagnostic Reasoning

The 4ESS object model provides an abstract model of the 4ESS and thus provides a framework for our diagnostic reasoning. This model implicitly contains information about the structure and behavior of the 4ESS. For example, if a standby relationship exists between two device objects, A and B, then device A's standby field will point to device B, and vice-versa. Much of the *reasoning* in ANSWER is accomplished by using *affective relations.* Affective relations define a highly abstract, non-behavioral, representation for modeling devices and are named for the fact that one component affects another in a diagnostically important way (Crawford et al. 1995; Singhal, Weiss & Ros 1996; Mishra et al. 1996). These relations are too weak to *simulate* device behavior, but serve to organize the domain knowledge in a coherent way; ad-hoc heuristics are replaced by a smaller set of general principles based on affective relations. Affective relations express aspects of the design at a level of abstraction that expert troubleshooters use to link symptoms to faults, and hence are easily acquired.

Two important affective relations used by ANSWER are the standby and sub-part relations. These relations are very general and can potentially apply to any device. Rules which maintain and use the standby relation were shown earlier in Figures 5 and 6. The sub-part relation is very important for diagnosis, since it can be used to isolate faults. For example, ANSWER has a rule that states that if many of device A's sub-parts fail, then the fault is most likely located in device A (*not* in its sub-parts).

It is worthwhile to compare the reasoning in ANSWER with that of its C5-based predecessor, the 4ESS-ES. In the 4ESS-ES, the reasoning was not based on affective relations, due to the lack of a general model of the 4ESS; instead, the reasoning in that system was based on many (overly) specialized ad-hoc rules. There were many cases where a single general rule was not quite adequate due to small differences in device behavior. The solution to this problem in the 4ESS-ES was to have a completely separate, nearly identical, rule for each device; the solution in ANSWER is to have a single general rule and specialize it using inheritance, as necessary.

Application Development, Use and Payoff

Application Development

The requirements for the expert system were available, having been written by system engineers for the 4ESS-ES. The design and implementation of ANSWER began with an object-oriented analysis of the 4ESS. Next, the design phase led to an object model, similar to what was shown earlier in Figure 2. Then, R++ code was written to implement some very basic functionality, to show the viability of combining rules and objects. Three developers then worked full-time over a one year period to complete the design, implementation and testing of the ANSWER expert system. The expert system contains 218 rules distributed over 23 object classes. Much of the basic functionality, or "building blocks" of the expert system (e.g., the code to query the database for information) are written as procedural functions using C++. These functions are then used within the declarative R++ rules that define ANSWER's intelligent behavior. Thus, for each programming task we were able to use the programming paradigm we felt was most appropriate. The expert system contains 3000 lines of R++ source code and 8000 lines of hand-written C++ source code. The 3000 lines of R++ source code were subsequently translated by the R++ preprocessor into 17,000 lines of C++ code, yielding a total of 25,000 lines of C++ source code.

The construction of the expert system progressed smoothly, with the majority of the problems resulting from outdated documentation. Several bugs were found with the implementation of the R++ language, but this was expected since we were the first serious application to use R++. However, these bugs were fixed quickly and they actually affected our productivity less than problems in our commercial development tools. Overall, our experiences with R++ were very favorable.

Application Deployment

Deployment of ANSWER began toward the end of 1996 and the system has been fully deployed since July of 1997. ANSWER handles 140 4ESS switches and approximately 100,000 alarms per week. The system executes 24 hours a day, 7 days a week (a disaster recovery machine ensures availability). Although initially there were a few problems with the overall operation support system, very few problems have been found with the expert system component.

Payoff and Benefits

We begin this section by describing the most *visible* benefits of ANSWER. ANSWER is able to intelligently filter the 4ESS alarms so that a much smaller number of alerts is generated. Experience has shown that the number of alerts is typically one-tenth the number of alarms. It is also important to note that these alerts are more meaningful than the alarms. The deployment of the expert systems (4ESS-ES and then ANSWER) have enabled the total number of analysts in the technical control centers to be reduced from 48 to 18 (this includes all 3 work shifts). Furthermore, ANSWER also saves a great deal of human effort by autonomously generating tickets, propagated with detailed relevant information, for those problems it can isolate—it takes on average one hour for an analyst to manually create a ticket containing the same information. ANSWER is also able to fix certain problems simply by sending a "restore" command to the 4ESS switch[2]. This use of ANSWER's "auto-ticketing" and "auto-restore" capabilities are expected to further reduce the number of analysts from 18 to 12.

We will now focus on the benefits of ANSWER over the previous C5-based expert system—that is, on the benefits derived from using object-oriented technology and object-oriented rules. The main benefits derived from using R++ and our object-oriented modeling approach are:

- *increased comprehensibility*: the object model provides an organizing framework for the expert system. Rules associated with each object class are located in a separate file. In the 4ESS-ES no such organizing principle existed and the rules were located in a few large files. Furthermore, in ANSWER the scope and impact of rules are easily determined, since R++'s path-based rules cannot reference arbitrary objects—*they must obey the object model.*
- *improved maintainability*: the use of an abstract object model, encapsulation as provided for by C++ objects, the use of an inheritance hierarchy to organize and share behavior, and the increased comprehensibility of the code lead to a highly maintainable system.
- *speed and reduced hardware costs*: our R++ based expert system was much faster than the 4ESS-ES, which was implemented in C5 (C5 is in turn faster than most commercial rule-based languages). Partly as a consequence of this, all 140 instances of ANSWER are able to execute on a single server, greatly reducing hardware costs.
- *reduced learning curve*: since we were already familiar with C++, learning R++ was quite simple. Learning a new rule-based language would have taken significantly more time.
- *easier integration with the rest of the OSS*: Since R++ is a superset of C++, the expert system was trivially able to interface to other parts of the operational support system. The 4ESS-ES, on the other hand, required a separate, hand-coded interface layer so that information could flow between the C5-based expert system and the remainder of the system, which was coded in C++.
- *use of procedural code within the expert system*: for some tasks, C++ code is clearly more effective than rules; R++ allows the developer to choose the most appropriate programming paradigm.

In addition to these benefits, the use of R++ and an abstract object model have led to reusable groups of rules. In fact, some of the rules from ANSWER have been adapted to model and monitor the behavior of other network elements. Another key insight is that we were able to use rules not only to encode domain knowledge, but to facilitate inter-object communication. For example, we were able to use rules to trigger activity based on a value in one object being greater than the value in another object; implementing this without rules requires the programmer to distribute code throughout both of the involved objects. This is an awkward, time-consuming and error prone process, which essentially requires the programmer to hand-generate the code that would automatically be generated by the R++ preprocessor. During the course of the project it became clear that some of the more complicated communication patterns (e.g., publish-subscribe) we were using could also be implemented using rules. Using the terminology of the object-oriented programming community, we focused on *behavioral design patterns*—pieces of reusable code that control how cooperating objects interact and distribute responsibility (Gamma, Helm, Johnson & Vlissides 1995). A description of how we used R++'s object-oriented rules to implement behavioral design patterns in ANSWER is provided by Weiss & Ros (1999).

Maintenance

Given the fact that maintainability was a key factor in the design of our expert system, many maintenance issues have already been discussed. To summarize, our use of an abstract object model and general relations using that model, along with certain features of object-oriented languages (inheritance, encapsulation) have made ANSWER easy to maintain. The expert system is being maintained by one of the three original developers. Based on his experience, we estimate that adding new functionality to ANSWER takes only about one-half the development time that would be required to add the same functionality to the 4ESS-ES. In particular, based on our use of a fairly generic device model, he frequently finds that much of the code necessary to implement the new functionality is already present in the system. Finally, we believe that this reuse has led to higher quality code, with fewer bugs.

[2] Please note that while the 4ESS-ES did perform alarm filtering, the auto-ticketing and auto-restore capabilities are new to ANSWER.

Conclusion

The use of both rule-based and object-oriented technologies in ANSWER has proven to be highly effective. By providing an abstract device object with diagnostically motivated affective relations, a simple form of model-based reasoning was able to be applied to a domain normally too complex for such methods. Thus, this approach has led to a middle-ground between model-based reasoning and heuristic (ad-hoc) first generation expert systems. The use of object technology has also provided a principled approach for designing, implementing and organizing the expert system and is responsible for ANSWER being a more comprehensible and maintainable system than its predecessor, the 4ESS-ES.

Our experience with R++ and the modeling approach described in this paper has been very positive. In fact, we have been surprised at how few problems we have seen since the application has been deployed. We attribute this success to the benefits of our approach, which we have already described in detail, as well as to the up-front work in developing a good, comprehensive, object model.

ANSWER is fully deployed and is monitoring and maintaining all of AT&T's 4ESS switches. Additional operation support systems are now being implemented using R++ and the approach described in this paper—using some design principles we have identified. Active development is continuing on R++, in order to enhance its capabilities. As mentioned earlier, R++ is now publicly available.

The approach described in this paper is very important for one additional reason—we believe it provides a way for data-driven programming to enter the mainstream. Currently, rule-based languages are considered "AI languages" and are not used by the large number of "mainstream" developers. As we showed earlier, in addition to representing knowledge, rules can facilitate inter-object communication. By adding rules to an existing, popular language, these and other benefits of data-driven programming, long known to AI programmers, can be shared by a much larger population of developers.

References

Chen, C., Hollidge, T., and Sharma, D. 1996. Localization of Troubles in Telephone Cable Networks, *Innovative Applications of Artificial Intelligence*, Vol. 2, AAAI Press, Menlo Park, CA., pp. 1461-1470.

Cooper T., and Wogrin, N. 1988. *Rule-Based Programming with OPS5*, Morgan Kaufmann, San Mateo, CA.

Crawford, J., Dvorak, D., Litman, D., Mishra, A., and Patel-Schneider, P. 1995. Device Representation and Reasoning with Affective Relations, *Proceedings of the Fourteenth International Conference on Artificial Intelligence* (IJCAI-95), pp. 1814-1820, Montreal, Quebec, Canada.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.

Mishra, A., Ros, J., Singhal, A., Weiss, G., Litman, D., Patel-Schneider, P., Dvorak, and D., Crawford, J. 1996. R++: Using Rules in Object-Oriented Designs, *Addendum Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA).

Patel-Schneider, P. 1998. The R++ home page: http://www.research.att.com/sw/tools/r++.

Rabinowitz, H., Flamholz, J., Wolin, E., and Euchner, J. 1991. NYNEX MAX: A Telephone Trouble Screening Expert, *Innovative Applications of Artificial Intelligence 3*, Smith, R. & Scott, C., eds., AAAI Press, Menlo Park, CA., pp. 213-230.

Rumbaugh, J., Blaha, M., Premerlani, W., and Eddy, F. 1991. *Object-Modeling and Design*, Prentice Hall.

Sasisekharan, R., Seshadri, V., and Weiss, S. 1996. Data Mining and Forecasting in Large-Scale Telecommunication Networks, *IEEE Expert*, 11(1): 37-43.

Singhal, A., Weiss, G. M., and Ros, J. P. 1996. A Model Based Reasoning Approach to Network Monitoring, *Proceedings of the ACM Workshop on Databases for Active and Real Time Systems* (DART '96), Rockville, Maryland, pp. 41-44.

Weiss, G. M., and Ros, J. P. 1999. Implementing Design Patterns with Object-Oriented Rules, *Journal of Object Oriented Programming.* To be published January, 1999.

Weiss, G. M., and Hirsh, H. 1998. Learning to Predict Rare Events in Categorical Time-Series Data, *Proceedings of the 1998 AAAI/ICML Workshop on Time-Series Analysis*, Madison, Wisconsin.

Weiss, G., Eddy, J., Weiss, S., and Dube, R. 1998. Intelligent Telecommunication Technologies, *Knowledge-Based Intelligent Techniques in Industry (chapter 8)*, L.C. Jain, ed., CRC press.