Enforcing Management Policies in Distributed Systems

Wolfgang A. Halang, S.K. Jung and Bernd Kramer
1FernUniversität Hagen, 58084 Hagen, Germany
2Chung Buk National University, 360–763 Cheongju, Chungbuk, Korea
Phone: (++49) 2331 987 371
Fax: (++49) 2331 987 375
EMail: bernd.kraemer@fernuni-hagen.de

Abstract

The objective of our work is to develop a platform independent management architecture for open distributed systems. In this paper we describe one component of this architecture which enforces the observance of formally definable management policies and executes clerical management tasks automatically. Management components are defined in terms of an object-oriented class structure and policies are represented by a variant of production rules. Rule execution and automated management tool invocation is controlled by a Marvel-based policy management environment.

Keywords: Automated policy management, distributed systems management, middleware platforms

1 Introduction

In open distributed computing environments the activities in different management domains and local changes to management policies may lead to inconsistencies in larger context and must therefore be consistently surveyed and coordinated. Platform, management domain specific, regional and network-wide policies interfere with each other and are subject to constant change. This situation is impeded by the usage of heterogeneous management tools and runtime environments available on the different types of platforms that are typically interconnected.

To support the early detection or even avoidance of erroneous management decisions in large distributed systems, automated policy checking and enforcement is desirable. The delegation of simple management tasks to automated tools adhering to formalized policies would also relieve much of the burden from the human system administrator.

This paper describes a first experiment with a flexible and partially automated approach to distributed systems management. A hypothesis underlying this experiment was that an existing process-centered software development environment, Marvel [4], can be of use in the provision of management support in the domain of distributed processing environments. Different management tools, e.g. network management tools and an adaptive load balancing architecture, are embedded in a Marvel application to provide a comprehensive management environment.

The kernel management model underlying our approach is described in Section 2. Section 3 then sketches the architecture and functionality of Marvel. A simple sample application is then presented in Section 4 to demonstrate the cooperation between different management tools.

2 Management Model

Our management model relies on an object based approach, i.e., all components of a distributed system are viewed as instances of different object classes.

2.1 Objects and Domains

Any instance which is target of management activity is termed managed object and an object that initiates management activities is called a manager. Depending on the task being executed one object can be in both groups at different times. Management activities are invoked at management interfaces. This is different from the OSI management approach where a managed object is modeled as a separate object from the resource it represents.

Management objects are grouped into domains [12]. Domains are used for modularization as well as for the provision of different views on the same system. One object can be in different domains at the same time, depending on the viewpoint.

2.2 Policies

The goals of system management as well as the necessary activities to reach the goals are described with policies [7, 10]. The management of large organizations is typically hierarchically organized. Therefore global policies, defined by the head of the organization, are used for the deduction of domain policies. Domain policies are further refined to create policies for the objects within a particular domain. Inter-domain relationships are also described with policies and they should be consistent with global policies as well as with local policies. In this context the formalization of policies has two advantages: consistency can be verified and policy enforcement can be delegated to the system.

Every policy has a modality, that either represents an authorization or a motivation for an activity. The modality could be implicit in global policies, but
when a global policy is refined to describe certain management activities, the modality will be expressed explicitly. Therefore the management system must ensure that every activity is motivated and that the manager is authorized to perform this action. If the management activity is initiated by a human administrator, the motivation is generally not an issue for the system. In this case only the authorization will be checked.

Further concepts of policies include: subjects that are the objects which the policy applies to; a target object that states which object class the policy is directed to; a goal that defines corporate-wide or project-specific goals or specific actions of the policy; and constraints that specific conditions that must all be satisfied before the policy can become active.

2.3 Automated management

The complexity of large distributed systems requires at least a partly automated management environment that provides the human administrator with more support from the system than just collection, filtering and presentation of management data. It is quite natural in a hierarchical structure that the assignment of tasks becomes more specialized and less comprehensive on the way from the top to the bottom of the structure. Especially highly specialized tasks with a strictly limited scope are promising candidates for automation.

In a rule based environment the potential for automation is not limited to some kind of watchdog functionality. The system can actively support the human administrator by using the knowledge, provided by the formalized policies. That is, every command issued by a human administrator can be checked for applicability to the system in its currently recorded state and further activity can be initiated automatically, if necessary. An example for this type of support is given in Section 4.

2.4 Rule based management

Management activities are determined by policies. Informal policy statements must be translated into a machine readable format. The result is a set of rules that describes the management activities unambiguously. The transformation of verbal policy statements into rules is currently done by hand. Policies that define detailed instructions for a special object type can easily be transformed into a rule. Examples for this type will be given in Section 4.

General policies, describing the management tasks at an abstract level, are more difficult to deal with. A statement like

"At least one printer must be available at any time"

can not be translated into a single rule. The policy must be refined before it can be expressed in terms of rules. In this example, several policies about maintenance, redundancy and error recovery are required to ensure a high availability of the printing service.

Another approach, currently under investigation, translates abstract policies into special rules. These rules do not activate a management tool, but the post condition of the rule changes the object base in a way that the preconditions for other rules are affected. This seems to be particularly well suited for supervisory functions on a medium grained management level.

2.5 Consistency Checking

Conflicting policies are very likely to appear in a large distributed system. Conflicts will arise between rules on different hierarchies as well as on the same hierarchy. The resolution of conflicts on the same hierarchy level will require a modification of at least one of the conflicting policies. Conflicts between different hierarchy levels can be solved by over-ruling without a modification.

The formalization of policies opens the possibility for a systematic consistency check. Proving the consistency for a set of rules is still an active research area where different approaches are investigated [2, 11]. One approach currently under investigation in our group is the use of predicate logic for this aim. In general, consistency is not provable for an arbitrary set of predicate logic formulae. Therefore we impose several restrictions on the type of formulae. Policies are formulated as Horn clauses with equalities, inequalities and known facts used as atomic formulae [9].

3 Functionality of Marvel

The rule-based software development environment Marvel [4] is used as domain manager. Marvel was originally designed as an environment that assists software development and evolution. To construct a Marvel model, the developer must produce a data model and a process model.

System components are represented in the Marvel environment by data objects which are instantiations of the object oriented data model. The management activities are performed by pre fabricated management tools. They are included into the Marvel system with special envelopes and activated by production rules. From this point of view, the Marvel system combines different management tools into a comprehensive management environment.

3.1 The data model

The data model gives the types, or classes, of the objects involved, their attributes and the relationships between them. At runtime the managed objects are represented in an object base. The object base also maintains history and status of the objects. An example for a data model is given in Figure 1. Here the object class MO (short for "managed object") is
MO :: superclass ENTITY;
  name : string;
  status : (Up, Down, NotChecked) = NotChecked;
end

DOMAIN :: superclass MO;
  hosts : set_of HOST;
  users : set_of USER;
end

HOST :: superclass MO;
  agent : AGENT;
  scheduler : SCHEDULER;
  nodemgr : NODEMGR;
  loc_user : link USER;
end

USER :: superclass MO;
  auth : (True, False, NotChecked) = NotChecked;
end

AGENT :: superclass MO;
  export : (Export, Udrau, NoCheck) = NoCheck;
  chg_user : link USER;
  todo : (Wait, Check, Nothing) = Nothing;
  time : integer;
  tmp : integer;
  timeinput : (TRUE, FALSE) = FALSE;
end

SCHEDULER :: superclass MO;
  algorithm : (SER, LMS) = LMS;
end

NODEMGR :: superclass MO;
  time : integer;
  tmp : integer;
  timeinput : (TRUE, FALSE) = FALSE;
end

Figure 1: Data model

The most generic definition, inherited by all the other classes. The superclass ENTITY is provided by Marvel as a root for the data model. An attribute with the keyword link in the type definition refers to a named and typed relation to an instance or a set of instances of the given class.

3.2 The process model

Marvel uses a set of production rules, the process model, to model computer supported management tasks including knowledge about data and the role of tools in individual steps. Each rule defines

a) object classes affected by the rule,
b) the preconditions which must be satisfied if the activity is to be carried out,
c) the activity, given as an envelope to be executed, and

d) the effects of the activity execution on the object base in terms of an alternative list of assertions reflecting different outcomes of an activity.

The Marvel kernel provides means of enacting the process model. It does so in an "expert-system-like" manner by opportunistic processing. If the precondition of an activity is satisfied that activity will be invoked. This may in turn result in the satisfaction of the precondition of further activities, and by forward chaining, they will be invoked, too. If a particular activity is chosen by a user and is not eligible for invocation, the Marvel kernel will try to build a backward chain of activity whose activation provides the precondition necessary for the selected activity to be performed. Chaining does not introduce a computability problem as the predicates occurring in preconditions of rules are built over attributes of objects, the collection of objects to be checked is always finite, and Marvel verifies the consistency between process and data model prior to enaction.

Figure 2 shows an example for a rule (line numbers are added for reference only!). Line 1 shows the name of the rule, shutdown, and the object class HOST of the formal parameter ?h. The keyword hide indicates that this rule is visible to authorized users only. Hidden rules can be activated either by authorized users or by chaining. The statements in Line 2 and 3 are called ad hoc queries as the object base is searched for two objects that have ?h as an ancestor. All ad hoc queries must be successful to activate the rule.

Line 4 contains one precondition, the attribute status of object ?h must be set to Up. The keyword no_chain states that backward-chaining is not allowed for this precondition. Line 5 gives the name of a tool class, here COMMANDS, displayed in Figure 3. The attributes of the tool class contain the names of envelopes as strings. Envelope shutdown (cf. Figure 4) will be executed if the rule is fired with ?h.hostname given as parameter to the envelope.

Figure 2: “shutdown” rule
Lines 6 to 8 contain two postconditions stating alternative effects of rule execution. Exactly one effect becomes true when an activity is completed. In case of success the status of the objects a and s is set to Down, otherwise nothing is changed indicated by the keyword no.assertion in line 8.

The Marvel environment consists of a server enacting the process model on behalf of several clients. Every client can ask the server for the activation of a rule and the server will try to fire this rule. Chaining is automatically applied if necessary and possible. To avoid inconsistent object states due to concurrent rule activation, every object is locked as long as it is used by an active rule. This mechanism allows concurrent management activities, activated by different clients, which may be located on different nodes.

3.3 Tool integration

The encapsulation feature of Marvel allows an easy integration of separate management tools under a common set of policies. To integrate a tool into a Marvel environment it is necessary to create an envelope and to add the name of the envelope to a tool class. An envelope specifies types of inputs and outputs of tool invocations; it uses Unix shell scripts whose type is indicated in the head of the envelope) to activate one or more tools in their proper environment and matches the tools' relevant return codes with the effects of the invoking rules. That is, if the tool may, for example, return with three different codes, the corresponding rule must provide three effect specifications listed in the order of the corresponding codes.

Tool envelopes are written in the Shell Envelope Language [3]. An example for a tool envelope is given in Figure 4. The head describes the name of the envelope, shell to be used, input and output parameters. Between the keywords BEGIN and END are the shell commands. The second line in this block performs the tool execution, here the executable program is named ShutDown and is called with the hostname as parameter. The result of the execution is stored in the variable prog.answer and then evaluated in the following if statement. Two possible results are defined, matching the number of postconditions in Figure 2.

As soon as a rule is activated, the Marvel kernel opens a special window (in X-window mode only). This window consists of two parts: the upper part displays the chain of rules that is executed and the lower part is used for communication between the envelopes and the user. Any messages occurring during execution of the envelopes are printed here (output is performed by the echo command in Figure 4), and the user can be asked for additional information if necessary.

4 Implementation

In our experiment we use a scheduling tool as an example management application. The scheduler realizes an adaptive load distribution between several hosts in a workstation cluster. A more detailed description of the scheduling algorithm is given in [6, 5]. ANSAware [8] is used as a midware platform to provide a location and vendor transparent environment.

Figure 5 gives an overview of the architecture. ANSAware components and communication links to ANSAware components are printed with dashed lines. Every node (here a node represents a host) has one Agent and one Scheduler running. Agents gather local information and provide it to their schedulers. Clients ask their local scheduler for a recommendation for the execution of a job.

In an ANSAware environment, every node has a Node Manager and a Factory running. The node manager provides an architectural interface for the creation, simple monitoring and destruction of ANSA services on a single node [1]. It uses the factory for the actual mechanics of creation and destruction. Furthermore, the node manager maintains a database with service descriptions. Here a managed service can be marked as permanent, that is, the node manager will

```plaintext
COMMANDS :: superclass TOOL;
export : string = "export";
withdraw : string = "withdraw";
shutdown : string = "shutdown";
startup : string = "startup";
END

Figure 3: Tool class

ENVELOPE shutdown;
SHELL sh;
INPUT string : HOSTNAME;
OUTPUT none;
BEGIN
  echo "Shutting down host $HOSTNAME"
  prog_answer="/bin/ShutDown -H $HOSTNAME"
  if [ "$prog_answer" = "ShutDown: started!" ]
    then
      echo "Shutdown started."
      RETURN "0";
    else
      echo "Error while computing ShutDown"
      RETURN "1";
  fi
END

Figure 4: "shutdown" envelope
```
automatically restart the service if it terminates. This functionality is used for our scheduling application to ensure a high availability (marked with dotted lines in Figure 5). The schedulers get their information about available agents from the ANSAware trader. Therefore all ANSAware components are required for the scheduling application.

From a management point of view, two aspects arise with the scheduling application:

1. The scheduling architecture is considered as an ANSA application. The system manager should be aware of this fact whenever he performs a management activity with ANSAware. Therefore the policies for the management of the ANSAware environment should reflect the requirements of this application.

2. The scheduling architecture is considered as a management tool. This requires the definition of appropriate policies for the behaviour of the scheduler.

Both aspects will be considered in the following sections.

4.1 Scheduler as ANSA application

The node manager provides a functionality to remove inactive services. This functionality is now used to illustrate a potential dependency between the ANSAware environment and an ANSA application.

We define a set $S_h$ that contains all ANSA services on a particular host $h$:

$$S_h = \{s_1, s_2, \ldots\}$$

The function $\mathrm{status}$ gives the current status of a service:

$$\mathrm{status}(Service) \in S_1$$

The function $\mathrm{sleep}$ determines how long the service is sleeping since its last activity:

$$\mathrm{sleep}(Service) \in \mathbb{N}$$

The node manager database stores a decay value for every managed service. If the decay value is set to $M$, the node manager will remove the service as soon as the capsule sleeps for $M$ seconds, but not otherwise. This functionality can be expressed as:

$$\forall s \in S_h . (\mathrm{sleep}(s) \geq M \Rightarrow \mathrm{status}(s) = \text{dead}) \quad (1)$$

$$\forall s \in S_h . (\mathrm{sleep}(s) < M \Rightarrow \mathrm{status}(s) = \text{active}) \quad (2)$$

with $M \in \mathbb{N}_+$

To keep the example simple, we do not consider other reasons beside a kill from the node manager for a service to die. Additional reasons would extend the premise of formula (2) by a conjunction of assumptions.

One possible strategy for the communication between agents and schedulers requires the agent to send its data periodically to all schedulers with a sleeping phase in between. The period $N$ is adjustable by the system administrator. The following formula describes this behavior:

$$\exists s_a \in S_h . (\mathrm{sleep}(s_a) \leq N) \quad (3)$$

with $N \in \mathbb{N}_+$

In general, conditions on the input variables can be computed which imply a certain desired property of the system. In this example it follows easily from (2) and (3) that an agent $s_a$ is active provided that

$$N < M$$
With the variable \( ?a. \text{timeinput} \) set to \( \text{TRUE} \) all preconditions are provided either for \text{LegalTime} or for \text{IllegalTime}. One of the two is fired automatically, depending on the comparison between the new value and the node manager timeout. If the new value does not fulfill the constraint of \text{Pol}, then \text{IllegalTime} is fired and the administrator gets an error message without any change to the system. Otherwise \text{LegalTime} is fired and the new value is sent to the agent.

In this example three rules are needed to realize one policy. This is due to the fact that the constraint of \text{Pol} is realized as a precondition in \text{LegalTime}. Therefore the value must be entered into the Marvel system before the interpreter can decide if it fulfills all constraints.

### 4.2 Scheduler as management tool

The use of the scheduler as a management tool requires the definition of policies. As an example, we shall investigate the following informally stated policy for our resource management system:

> "Every authorized user is allowed to withdraw his host from the pool as long as the user is active."

This policy statement is translated into several Marvel rules. Figure 7 shows a simplified version of these rules. The withdraw rule needs a positive user authentication as precondition, according to the policy. Rule check-auth is designed to activate the authorization mechanism and check-agent reads the current status of the corresponding agent. If the user is not already checked, the rule check-auth will be fired and the user is asked for authentication. Both rules are hidden for a normal user and automatically fired through backward chaining as indicated with dashed lines. If the withdraw rule was fired successfully, forward chaining is automatically applied to initiate a watchdog mechanism. The box watch-user in Figure 7 indicates a set of rules, designed to check periodically if the user is still active. If the user becomes inactive, the previously withdrawn host is automatically exported.

This delegation of tasks is well supported by our approach, because the Marvel environment interprets the rules and is able to activate management tools without intervention of human administrators. The fact that a Marvel server can enact the process model on behalf of several clients enables the system to perform several delegated tasks concurrently. Concurrency is not supported within a single client.

### 5 Conclusions

To keep large distributed systems manageable, the system should support the human administrator in two ways: Simple management tasks should be executed automatically and the system should provide guidance and support to the human administrator according to the corresponding policies. We described an ongoing experiment towards a rule based management
architecture for distributed systems. The software development environment Marvel is used as a platform for a rule based management approach. The motivation behind this approach was to gain rapid feedback about our management model through the use of existing tools that showed sufficient potential to support this experiment. The encapsulation feature of Marvel allows an easy integration of separate management tools under a common policy. The automation of simple management tasks is supported and the observance of policies is enforced.

As a further improvement Marvel could be asked for all necessary preconditions as well as for all resulting postconditions for a planned activity. These “What is necessary to...” or “What happens if...” questions are extremely useful for the management of complex distributed systems, because the system administrator can learn about the consequences of a planned activity without executing the command. Currently this feature is not included in the Marvel environment, but an approach using an additional set of rules is under investigation.

Despite the encouraging results of our prototype implementation, further research is still needed, especially in the area of policy transformation into appropriate. The transformation should imply a consistency check to reduce the risk of errors. The possibility for automated negotiations seems to be another indispensable feature for open distributed systems.

6 Acknowledgement

We would like to thank Mr. Norbert Völker for reading a draft of this paper and his helpful advice.

References