20

Towards a Comprehensive Distributed Systems Management[1]

Thomas Koch and Bernd Krämer
FernUniversität
58084 Hagen, Germany
Phone: (++49) 2331 987 371, Fax: (++49) 2331 987 375
E-Mail: {thomas.koch, bernd.kraemer}@fernuni-hagen.de

Abstract

This paper describes a hybrid approach towards distributed systems management. The rule-based software development environment Marvel is combined with conventional management tools and a new adaptive resource management approach. This approach improves distributed system management in at least two aspects: Formalized management policies are automatically enforced and standard management tasks can be delegated to the system. The encapsulation feature of Marvel allows an easy integration of separate management tools under a common policy. The description of a resource management application illustrates our approach.

Keyword Codes: C.2.4; K.6.4; D.2.9
Keywords: Distributed systems management, production rules, policy enforcement, ANSAware

1 INTRODUCTION

The availability of more and more distributed processing environments, like DCE[1], CORBA[2], CORDS[3] or ANSAware[4], has disclosed the limitations of conventional system management strategies. Especially the heterogeneity of nowadays distributed systems imposes a great challenge to system administrators. Two reasons for management heterogeneity can be identified: One is the heterogeneity of the system components and the other lies in the specialization of the available management tools. Most commercial management tools are limited in scope (e.g. networking or configuration only) and in the number of supported platforms. This situation will become even worse in the future, because distributed systems are expected to grow rapidly.

To keep large distributed systems manageable, the system should support the human administrator with the autonomous execution of simple management tasks according to a given policy. This "management by delegation" approach requires the formalization

[1]This research was sponsored by the European Union under contract no. ECAUS003 as an European-Australian collaborative project on Information Systems Interoperability.

K. Raymond et al. (eds.), *Open Distributed Processing*

of management policies with the additional advantage that policies can be checked for consistency.

In this paper we review an experimental study which we recently undertook with a flexible and partially automated approach to distributed systems management. In particular, we investigate the application of an existing process-centered environment, Marvel [14], to provide management support in the domain of distributed processing environments. Marvel was originally designed as an environment that assists software development and evolution by guiding and aiding individual programmers and helping coordinate software development teams. It uses a variant of production rules to model development tasks or design methodologies including knowledge about design data and the role of tools in individual development steps. Task specifications are executable due to a dynamic forward and backward chaining of rules based on an interpretation of their pre- and postconditions. Activities are automatically performed by the environment when it knows that their results will be required by the user.

The paper is organized as follows: Section 2 describes a management concept for distributed systems. In Section 3 we introduce Marvel and its specification language MSL. This language is used in Section 4 to define an executable model of the management process including the consideration of different management policies.

2 MANAGEMENT MODEL

Management of any component of a processing system helps to control the component in such a way, that it can meet its requirements in service provision. The management of stand-alone systems is usually performed in an informal style, where the administrator changes the system or any component whenever needs arise. With the interconnection of individual components to distributed systems the informal management style is no longer appropriate due to the openness of the whole system and the partial autonomy of individual components. A software upgrade on one system, for example, could create problems on a remote component. The scale of complexity soon reaches a level where the conventional ad hoc style produces disastrous results.

Therefore the activity of distributed systems management needs to be organized in a more systematic way to ensure a consistent management strategy. We chose an object based approach for our management model where all the components of a distributed system are defined as instantiations of different object classes. This object oriented approach is most widely used and serves also as a basis for evolving international standards [5, 6, 7].

2.1 Object concept

Any entity which is a target of management activity is termed **managed object** and the object that initiates the management activity is called **manager**. The basic difference between both classes of objects is that manager objects have a degree of authority to ask for information and to perform management tasks. One object can be in both groups at different times, e.g., a scheduler acts as manager when it asks for load information, but it is a managed object when the administrator asks for statistical data about the scheduling

activity.

The types of interaction between managers and managed objects can be organized in three groups [8]:

- *control* actions,
- *requests* for information and
- *notifications* of events by the managed object.

2.2 Domains and policies

To deal with the size of large distributed systems, the management model groups objects into **domains**. Domains are used for modularization. They also provide different views on the same system. One object can be in different domains at the same time, depending on the viewpoint [9].

As already mentioned, we propose to use formal rules to organize the management process. A rule that describes a management activity is called a **policy**. A policy can give either an **authorization** or a **motivation** for an activity. Motivations can be both, positive or negative. Any management activity must be motivated and the manager must be authorized to initiate an action [10].

A typical example for a verbally expressed motivation policy could be:

> "Every source file must be registered in the revision control system."

Policies are considered as instantiations of special object classes. Every member of a class has at least the following attributes:

Modality: describes authorization or motivation. In the quoted example a positive motivation is given.

Policy subjects: defines the objects which the policy applies to, here "source file".

Policy target object: states which object class the policy is directed to, here subject and target object are identical.

Policy goal: defines abstract goals or specific actions of the policy, in our above example "registration in the revision control system" is the objective.

Policy constraints: defines conditions that must all be satisfied before the policy can become active. No constraint is given in the example because the policy applies to "every" source file.

Note that this example just gives the motivation to register a source file in the revision control system. Authorization would be expressed by an additional statement like:

> "Every user may use the revision control system."

The scope of a policy is given by the domain the policy is defined for. Here the object oriented approach gives another advantage: Policies can be specialized to minor domains by means of object inheritance. With this approach, it is automatically ensured that an inherited local policy is not in conflict with a global policy.

2.3 Management by delegation

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 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. The delegation of tasks is well supported by our approach, because the Marvel environment interprets the policies and is able to activate management tools without intervention of human administrators.

3 SOFTWARE PROCESS MODELLING AND MARVEL

Software process modelling has assumed considerable importance in discussions of software engineering. In particular attention has been paid to the use of software process modelling in the construction of software development environments.

3.1 What is software process modelling?

Essentially, software process modelling is the construction of an abstract description of the activities by which software is developed. In the area of software development environments the focus is on models that are enactable, that is executable, interpretable or amenable to automated reasoning. A particular "instance" of the software development process – the development of a particular piece of software – can be seen as the "enaction" of a process model. That model can be used to control tool invocation and cooperation. A software development environment for a particular development is thus built up around (or generated from) an environment kernel which is essentially a vehicle for constructing and enacting such software process models.

3.2 Functionality of marvel

Marvel is a rule-based environment that was designed to assist users with the software development process automating well understood and formalized tasks. To construct a Marvel model, the developer must produce a data model and a process model. The data model describes the objects to be managed during the process of software development such as specifications, design documents, test data, code or project data. The process model describes the activities carried out on those objects by the developers and tools involved in the specified development.

Marvel uses the data model to generate an objectbase in which all artifacts created during a development are held. The objectbase also maintains history and status of the objects. The data model gives the types, or classes, of the objects involved, their attributes and the relationships between them. The objectbase is implemented straightforwardly as a Unix file structure. Each object instance has associated with it a unique directory, and directories are structured according to the relationships between the object instances.

```
ENVELOPE withdraw;
SHELL    sh;
INPUT    string : HOSTNAME;
OUTPUT   none;
BEGIN
  prog_answer='/home/TUNIX/gerald/ANSA/ChangeExport -H $HOSTNAME -e'
  prog_status=$?;
  if [ prog_staus -ne 0 ]
  then
    echo "Problems with ChangeExport!"
    RETURN "1";
  else
    echo "Exportpolicy on $HOSTNAME changed."
    RETURN "0";
  fi
  RETURN "1";
END
```

Figure 1: Tool Envelope

The process model is given in the form of production rules. Each rule defines a) object classes affected by the rule, b) the precondition which must be satisfied if the activity is to be carried out, c) the activity, and d) the effects of the activity execution on the objectbase in terms of an alternative list of assertions reflecting different outcomes of an activity. Exactly one effect becomes true when an activity is completed. Activities are carried out by tools made known to the objectbase via tool envelopes.

The Marvel kernel provides a 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.

Both data and process model are expressed as "strategies" in the Marvel Strategy Language. Strategies can be imported into a main strategy. It is standard to define the data model in a single strategy but have multiple process model strategies for related tool sets.

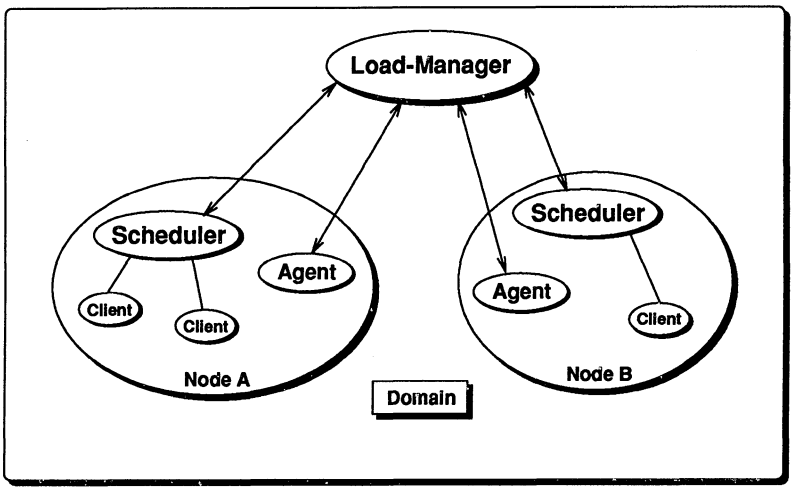

Figure 2: Resource Management Architecture

3.3 Tool integration

Typically, the activities of a Marvel model are performed by tools. Marvel is an open environment that allows the appropriate tools for the development to be added to the environment by the users. To integrate a tool into a Marvel environment it is necessary to create an "envelope". Tool envelopes are written in the Shell Envelope Language [15]. An envelope specifies types of inputs and outputs of tool invocations; in its kernel it uses Unix shell script (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 return with two different codes, like the example in Figure 1, the corresponding rule must provide two postconditions listed in the order of the corresponding codes.

4 AN EXECUTABLE MODEL OF MANAGEMENT POLICIES

Our experiment started from the hypothesis that Marvel's capabilities are suited for maintaining management tools of a distributed environment based on policies that are formalized in terms of Marvel rules and are then enforced by the Marvel kernel. To test the viability of our approach we built and used a Marvel environment for some sample management tools.

We shall now introduce a resource management application to illustrate our approach. Figure 2 shows a generic architecture for an adaptive resource management tool. We use a multi level approach to handle the complexity of large distributed systems as proposed

in [11], but it is sufficient for the purpose of this paper to restrict our description to the lowest level which corresponds to a single domain, as depicted in Figure 2.

Every node of a distributed system has a scheduler, giving recommendations to potential clients, and an agent to collect the required data. One centralized load-manager serves as management interface for all schedulers and agents of a domain, that is, the load-manager is not involved in the scheduling business but is responsible for all management tasks. A more detailed description of the architecture as well as the scheduling algorithm can be found in [12].

This management architecture is now embedded in a Marvel environment. As a consequence, all management commands will be issued to the Marvel interpreter and then Marvel decides which tool to use in what situation, based on the formalized policy and on the current state of the system.

4.1 Product model

As explained in Section 3, the managed objects of the system are described in terms of object classes, their attributes, and relations. Figure 3 shows the class definitions for our example. The most generic class is named *MO* (short for Managed Object). Every object in the system inherits the standard attributes from *MO*. Multiple inheritance is supported in Marvel, but not used in this example. 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.

During a session and beyond, the Marvel object base maintains a set of uniquely identifiable objects whose attributes are determined by the corresponding class definition. Possibly initial attribute values can be declared in the class definition, e.g. `NotChecked` in Figure 3. For every process in our example, Marvel maintains an object instantiation of the corresponding class.

4.2 Process model

Every management policy has to be encoded by Marvel rules. All rules together build the model implementing the automated management assistant. 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."

According to the object model described in Section 2.1, we can identify the following attributes:

Modality: Permission

Subjects: Every user

Target object: Resource agents (the command is given to the load-manager, but the agents are affected)

Goal: Withdraw target host from the pool

```
MO :: superclass ENTITY;
  name   : string;
  status : (Up, Down, NotChecked) = NotChecked;
end

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

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

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

AGENT :: superclass MO;
  export   : (Exported, Withdrawn, NotChecked) = NotChecked;
  chg_user : link USER;
  todo     : (Wait, Check, Nothing) = Nothing;
end

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

LOADMANAGER :: superclass MO;
  export : (Export, Withdrawn) = Export;
  host   : link HOST;
end
```

Figure 3: Description of managed objects

Constraints: User must be authorized and logged on the target host

The translation of a policy object into the Marvel environment can be done in three steps:

1. The constraints must be transformed into preconditions. The check if a constraint is satisfied or not, could require the use of a special tool. In this case a separate rule must be provided for the activation of the checking tool.

2. At least one rule is needed to reach the policy goal. The activation of a tool in order to reach the goal is not mandatory, although it is necessary in most cases. Some policies could reach their goal just by affecting the attributes of other objects, without tool invocation.

   If the target consists of instances of differently specialized managed objects, it could be necessary to create several similar rules taking different attribute sets into account to reach the goal.

3. The attributes must be changed after the rule was activated. Otherwise the Marvel interpreter would fire the same rule again and again.

One way to formalize our example policy is given by the following rule (line numbers are added for reference only!):

```
 1 withdraw[?h:HOST]:
 2  (and (exists USER ?u suchthat (linkto [?h.loc_user ?u]))
 3       (exists AGENT ?a suchthat (ancestor [?h ?a]))):
 4  (and          (?u.auth = True)
 5                (?a.status = Up)
 6       no_chain(?a.export = Exported))
 7  {EXPORTTOOL withdraw ?h.name}
 8  (and          (?a.todo = Check)
 9                (?u.auth = NotChecked)
10       no_forward(link [?a.chg_user ?u])
11       no_forward(?a.export = Withdrawn));
12  no_assertion;
```

Line 1 shows the name of the rule (*withdraw*) and the formal parameter object with its class. The statements in Line 2 and 3 identify the local user and the corresponding agent. In Marvel such objects are called derived parameters. They can be determined through existentially and all-quantified predicates inspecting the object graph along attribute links. From Line 3 to Line 6 list the precondition of the rule. The keyword `no_chain` in Line 6 ensures that the interpreter does not try to satisfy this precondition by firing other rules, as otherwise the interpreter could try to export an already withdrawn agent because this action would fulfill the precondition. If the precondition is satisfied, the activity in Line 7 will be activated, here the envelope `withdraw` in Figure 1 is executed. In this example the load-manager is instructed to withdraw the agent, given as parameter in the command. Two possible outcomes are then defined: In the case of a successful tool invocation, Line 8 to 11 will adjust the attributes of the involved objects, according to the new situation.

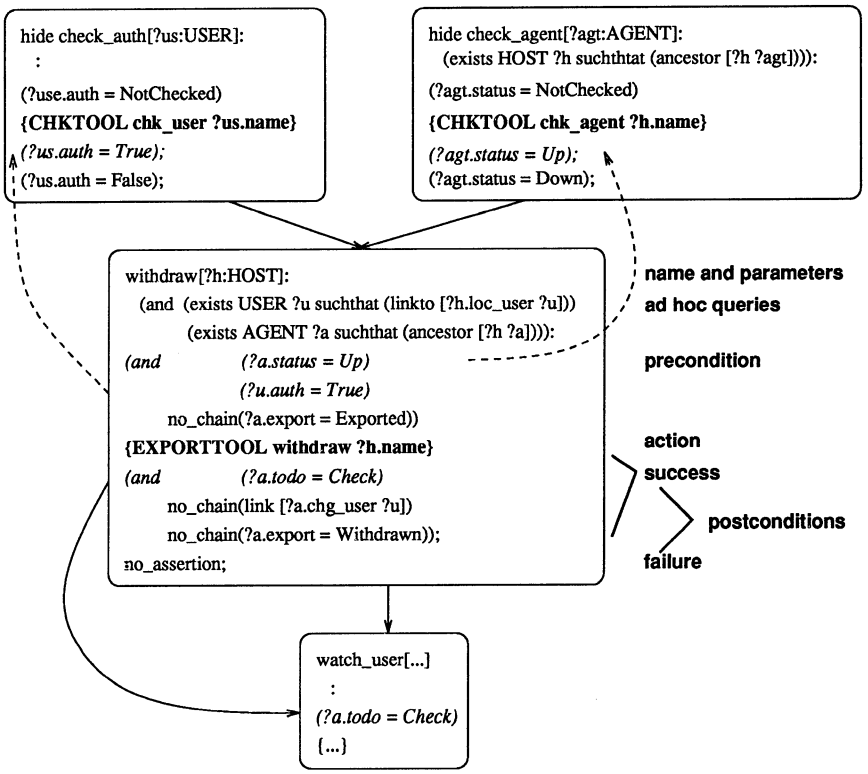

Figure 4: Rule Chaining

These changes may trigger other rules by the use of forward chaining. If the tool returns an error code, no attributes are changed according to Line 12.

Next we will assume that the policy has changed to:

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

The new situation is illustrated in Figure 4. The *withdraw* rule checks the user authentication, 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. 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 succesfully, forward chaining is automatically applied to initiate a watchdog mechanism. The box *watch_user* in Figure 4 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.

Note that the example policy just gives authorization to withdraw the host, here the motivation comes from the fact that the user issued a command to the interpreter. As soon as the user becomes inactive, *watch_user* is motivated to export the host. The authorization is implicitly given.

This scenario shows another important advantage of our management environment: 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.

5 CONCLUSIONS AND FUTURE RESEARCH

We described the application of the rule-based software development environment Marvel as a management environment for distributed systems. 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. Our approach improves distributed systems management in several areas: The automation of simple management tasks is supported and the observance of policies is enforced. Another important feature of our approach lies in the fact that existing management tools can be embedded in the new environment. The possibility to ask the system for preconditions or consequences without actually executing the command reduces the potential for erroneous activities drastically. 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 protoype implementation, further research is still needed, especially in the area of policy transformation into appropriate rules. The possibility for automated negotiations seems to be an indispensable feature for open distributed systems.

General policies, describing the management tasks on an abstract level, are more difficult to deal with than local policies. One approach currently under investigation translates abstract policies into special rules. They do not activate a management tool, but the postcondition of the rule changes the objectbase in such a way that the preconditions for other rules are affected.

Another problem with global policies is the potential for inconsistencies between global and local policies. A promising approach to this problem is described by Finkelstein et al. in [13]. Here a common logical representation of objects, policies and consistency rules is used to detect and identify inconsistencies. Meta-level rules may then be used to perform predefined actions on inconsistencies.

References

[1] A. Schill. *DCE Einführung und Grundlagen.* Springer Verlag, 1993. in German.

[2] The Common Object Request Broker: Architecture and Specification. Technical Report 91.12.1, OMG , December 1991.

[3] M. Bauer, N. Coburn, D. Erickson, P. Finnigan, J. Hong, P. Larson, J. Pachl, J. Slonim, D. Taylor, and T. Teorey. A distributed system architecture for a distributed application environment. *IBM SYSTEMS JOURNAL*, 33(3):399–425, 1994.

[4] Architecture Projects Management Limited, Castle Park, Cambridge. *Application Programming in ANSAware*, release 4.1 edition, February 1993.

[5] Reference Model - Open Distributed Processing - Part 2: Descriptive Model. Technical Report JTC 1/SC 21 N 8538, ISO/IEC, April 1994.

[6] Reference Model - Open Distributed Processing - Part 3: Prescriptive Model. Technical Report JTC 1/SC 21 N 8540, ISO/IEC, April 1994.

[7] Open Distributed Management Architecture - First Working Draft. Technical Report JTC 1/SC 21 N 8801, ISO/IEC, August 1994.

[8] A. Langsford and J.D. Moffett. *Distributed Systems Management.* Addison-Wesley, 1993.

[9] M. Sloman and K. Twidle. Domains: A Framework for Structuring Management Policy. In Morris Sloman, editor, *Network and Distributed Systems Management*, chapter 16, pages 433–453. Addison-Wesley, 1994.

[10] J. D. Moffet. Specification of Management Policies and Discretionary Access Control. In Morris Sloman, editor, *Network and Distributed Systems Management*, chapter 17, pages 455–480. Addison-Wesley, 1994.

[11] A. Goscinski and M. Bearman. Resource Management in Large Distributed Systems. *Operating Systems Review*, 24(4):7–25, October 1990.

[12] T. Koch, G. Rohde, and B. Krämer. Adaptive Load Balancing in a Distributed Environment. In *Proceedings of SDNE'94*, pages 115–121. IEEE, June 1994.

[13] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency Handling in Multi-Perspective Specifications. *IEEE Transactions on Software Engineering*, 20(8):569–578, August 1994.

[14] G.E. Kaiser; P.H. Feiler; S.S. Popovich. Intelligent Assistance for Software Development and Maintenance. *IEEE Software*, 40–49, May 1988.

[15] M.A. Gisi and G.E. Kaiser. Extending a tool integration language. In *First International Conference on the Software Process – Manufacturing Complex Systems*, pages 218–227, Redondo Beach, California, October 1991. Computer Society Press.