Policy Definition Language for Automated Management of Distributed Systems*

Thomas Koch, Christoph Krell, Bernd Krämer
FernUniversität Hagen
58084 Hagen, Germany
Phone: (++49) 2331 987 4538
Fax: (++49) 2331 987 375
EMail: thomas.koch, christoph.krell, bernd.kraemer@fernuni-hagen.de

Abstract

The heterogeneity, increasing size and complexity of distributed systems requires new architectures, strategies, and tools for their technical management. In this paper we propose a policy based approach to distributed systems management. The use of different abstraction levels allows a stepwise refinement from an informal strategic level to a formalized operational level. On the lowest level, we use a formal language for separate definition of policies and events, that enables the computer to check the syntax of a given policy description and to translate policies into executable rules.

To increase the capability for reasoning on a given set of policies, we extended the architecture by a graph model of the process semantics of operational policy and event specifications. The graph model is supported by a compiler mapping operational specifications into their semantic graphs, and performing analysis and manipulation functions on such graphs.

Keywords: Automated management, Graph system, CORBA

1 Introduction

The goal of system management is the provision of the highest possible quality of service (QoS) to the user [6]. This requires first the definition of the desired QoS in terms of measurable properties and second the description of suitable procedures (policies) to ensure the desired QoS.

According to a US market study administrators of large distributed systems spend 60%-80% of their time for routine tasks and urgent emergency repairs. Only the remaining fraction of time can be used for pretentious tasks like strategic planning or system optimization. A significant improvement of this situation requires the automation of routine tasks and an automated solution for typical problems.

Currently most management knowledge is memorized in the heads of system administrators. They know the demands of system users and their quality requirements. Management is performed by monitoring the system status, analyzing the actual situation and giving appropriate commands for error correction or general improvement of the system behavior. Therefore the administrators must transform the demands into appropriate commands. Most of the time this transformation occurs on the fly when an error requires instantaneous activity. This management style raises several problems:

- Similar situations may be interpreted differently by different managers with the result of varying reactions to the same problem.
- Management experience and knowledge is not documented. Therefore every administrator has to invent his own solution to a problem which may have been solved already by a colleague.
- The potential for automation is low with the result that administrators spent most of their time with reactive action to urgent problems.

Improvement of this situation requires the automation of standard management tasks according to a predefined strategy. Yet a prerequisite for automation is the detailed documentation of diagnostic knowledge and appropriate procedures, including a translation into a machine readable description.

2 Management model

We chose an object based approach for our management model. All components of a distributed system are viewed as instances of different object classes. Any instance which is the target of a management activity is termed **managed object** and an object that initiates management activities is called a **managing object**. The basic difference between both groups of objects is that managing objects have a degree of authority to ask for information and to perform management tasks. Depending on the task being executed, one object can be in both groups at different times. Two objects can interact through a communication link established between the management interface of a managing and a managed object. An object can have multiple managed and multiple managing interfaces [2].

*Research reported herein is partly sponsored by the BMFT, Project No. INF 26

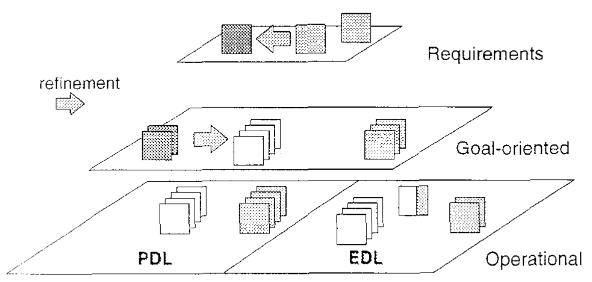

Figure 1: Policy hierarchy

To deal with the size of large distributed systems, the management model groups objects into **domains** [10]. Domains define projections on the set of existing objects. They 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.

Any command or method invocation issued with the purpose of management related action is called a **management activity**. The term management activity is therefore not restricted to commands that will change the behavior of an entity. Reading the value of parameters is also called a management activity as well as a notification about important changes in the system.

A list of related management activities, created to provide a solution to a management related problem, is called a **management task**. A management task could be simple or complex depending on the abstraction level used for the description. Therefore the number of required management activities for a task can vary in a wide range. On the highest abstraction level, a task may be described by the definition of desired effects without a detailed list of activities. Then further refinement is needed to break down the task into subtasks until the required degree of detail is reached. Usually the required management activities for a certain task must be performed in a predefined order. Such a sequence of activities is called a **chain**.

3 Policy hierarchy

The biggest problem with the formalization of management knowledge is the great semantic gap between the form in which demands of system users are expressed, usually given in plain text on a very abstract level, and the formal description of appropriate procedures in terms of machine executable commands. The use of different abstraction levels allows a stepwise refinement from an informal strategic level to a formalized operational level. The number of abstraction levels varies greatly in the literature on management policies. Wies [12] suggests the use of four levels, while Sloman-EtAl [9] provides a more flexible policy language to encompass the whole abstraction range within one level.

The development of our management hierarchy was guided by the following considerations: The description of the hierarchy should clearly define the usage of each conceptual level and the mapping of policies to different levels should be unambiguous. Furthermore we require the usage of different language constructs for each abstraction level in order to provide a most useful mechanism for the refinement process. On the other hand we clearly see the need for more than two levels in order to bridge the semantical gap between abstract management strategies and formalized policies. As a result of these considerations we suggest a three level hierarchy as illustrated in Figure 1. This approach matches well with the viewpoints of the RM-ODP [1]. The following correspondence of policy levels to ODP viewpoints can be identified:

Requirements level	$\Longleftrightarrow$	Enterprise viewpoint
Goal-oriented level	$\Longleftrightarrow$	Information viewpoint
Operational level	$\Longleftrightarrow$	Computational viewpoint

Since the implementation and distribution of objects is supported by an appropriate middleware (CORBA, DCE, ANSAware, etc), the engineering and technical viewpoints are of no concern for the definition of policies.

Our policy hierarchy allows a stepwise refinement of policies from an informal strategic level to a formalized operational level. Technical aspects and the degree of detail are increasing towards the operational level, while business aspects may no longer be visible for the formalization of policies at the operational level. Refinement takes place between different levels as well as on the same level.

3.1 Requirements level

The requirements level is directed to the needs of the users. On this level the policies are typically described in prose. This kind of policy is usually formulated by the administration of a company or administrative body. The requirements level allows the expression of any desired behavior without technical constraints. To enable a smooth transition to more detailed description levels, relevant keywords, attributes, and relationships among them are made subject to hyper-text structuring and navigation. Policies on the requirements level are refined until a sufficient degree of detail for a transition to the next level is reached.

An example for a policy on the requirements level would be:

> DP-responsibility1:
> "The data-processing department is responsible for sufficient availability of **mass-storage**."

Every policy gets a unique id, here "DP-responsibility1", and is embedded in a directed acyclic graph of policy statements through ancestor and descendant links. These links are established by pointing and mouse clicking when the manager enters the policy into the system. The boldface printing of the phrase "mass-storage" indicates that an additional definition is available for this phrase.

DP-responsibility1 is a very general policy and therefore further refinement into more detailed specifications is required. In this example several different policies for the management of mass-storage will result from a thorough refinement process, but we will restrict our considerations to house-keeping problems as one possible refinement branch. The refinement process within one level should be finished as soon as the resulting policies provide sufficient detail for a simple transition to the next level. Two possible refinements are presented as examples:

> Tempfiles:
> "The **administrator** should remove all **temporary files** and **core dumps** older than 24 hours once a day."
>
> Garbage:
> "The **administrator** should immediately remove all **garbage-files** if more than 80% of the **available disk-space** on a **home partition** are used."

Here especially the phrase "garbage-files" needs further explanation, given by the following statement:

> *garbage-files*: **core dumps**, **temporary files** and **object-files** that are not used for at least three days.

The provision of separate definitions for certain terms and phrases allows a well structured policy specification with a high potential for reuse and readability even for nonspecialists. Additionally consistent usage of these general terms is ensured. This approach is similar to the usage of declarations in programming languages.

3.2 Goal-oriented level

The goal-oriented level serves as an intermediate level between the informal description on the requirements level and the formal definition on the operational level. Here the roles of the involved objects are defined in terms of constraints and actions. The goal-oriented level reflects the fact, that management is *event-driven* and performed with *discrete* activities. The continuous service-provision with a certain quality, as it might be expressed on the requirements level, must be translated into discrete management activities performed by managing objects on managed objects. For any management activity the managing object must be **motivated** and **authorized**. Both aspects are generally independent and must be described separately.

To support a stepwise formulization of policies when moving on to the goal-oriented level, a template is provided for the policy description. Every policy must have a core set of attributes[7], and may have additional optional attributes. The following attributes are mandatory for each policy:

Name: A unique identifier.

Subject: Definition of the managing object. A single object, a set of explicitly named objects or a domain can be specified.

Name	Goal02
Ancestor	Garbage
Descendant	Goal03
Subject	administrator
Target Object	home partition
Action	remove garbage-files
Constraint	none
Event	disk-usage > 80%
Modality	Obligation
Status	entry
Author	Thomas Koch

(Mandatory attributes are printed in **bold**)

Figure 2: Goal-oriented policy

Target object: Definition of the managed object, that is the object which is the target of the management activity. A single object, a set of explicitly named objects or a domain can be specified.

Action: Definition of the management task to be executed. The complexity of the management task specified generally depends on the refinement level.

Constraint: Definition of conditions that must be fulfilled before an action is initiated.

Event: Definition of the event used to trigger the policy.

Modality: Two modalities are provided:

> **Obligation:** The subject is obliged to initiate the action on the target object if the event happened and the constraint is fulfilled.
>
> **Authorization:** Either **permission** or **prohibition** is given to the subject for initializing the action on the target object. The authorization may be restricted by constraints.

The use of plain text for the expression of attribute values is allowed, but it is advisable to define technically realistic goals, because otherwise the transformation into operational policies could be impossible. This situation is known from program specifications, where some correct specifications are not implementable.

Figure 2 shows the result of the translation from the requirement policy Req03 into the goal-oriented policy Goal02. Req03 provided sufficient detail for a translation without any difficulties, as recommended for the refinement process.

Goal02 as presented in Figure 2 contains mandatory attributes in plain text, which is allowed on this level but not sufficiently detailed for an automated interpretation. Therefore Goal02 needs further refinement, otherwise an automation of this management task is impossible. The next refinement step requires more technical knowledge about the implementation of the involved objects. In our environment a proxy object is created for every file-system to provide the required management interface. Figure 3 shows the CORBA interface definition for such a management interface as an example. We assume the provision of

```
interface FileSystem {
typedef string File;
// the Capacity of a file system
// as displayed with the UNIX df command
readonly attribute unsigned short Usage;                    5
sequence<File> ListFiles ( in string Filter );
void RemoveFile ( in File aFile );
...
};
                                                            10
interface ExtendedFileSystem : FileSystem {
// operation to remove garbage files
void removeGarbage();
...
};                                                          15
```

Figure 3: CORBA interface for an extended file-system

an *ExtendedFileSystem* for our running example. The *ExtendedFileSystem* offers a method for garbage removal (line 13), which simplifies our refinement process slightly. Without this functionality, several basic operations (like reading the directory or removing files) must be combined to perform the required management task.

Furthermore, we assume that a domain service is provided in the management environment. Naming of domains is conceptually similar to a file-system, e.g. the path to a subdomain is denominated by the sequence of ancestor domain-names separated by a slash ("/"). The usage of common wildcards (like "*" for any sequence of characters) is also supported.

The subject of Goal02 in Figure 2 is simply called "administrator". On many operating systems a special user named "root" is provided for administrative purpose. In our management system any automated management task is performed by processes owned by the "root" user to avoid any access restrictions.

With this technical information the policy Goal02 can now be refined into Goal03 as illustrated in Figure 4. The refinement process on the goal-oriented level is finished with policy Goal03, because this policy provides sufficient technical details for a translation into an operational policy as illustrated in the next section.

3.3 Operational level

On the operational level, we use a *Policy Description Language (PDL)* that enables the computer to check the syntax of a given policy description and to translate policies into executable rules. Here the subject and target objects are domain expressions or object descriptions with a well defined syntax. The

Name	Goal03
Ancestor	Goal02
Descendant	disk80
Subject	root
Target Obj	filesys/home/ExtendedFileSystem
Action	removeGarbage()
Constraint	none
Event	disk-usage > 80%
Modality	Obligation
Status	refined
Author	Thomas Koch

Figure 4: Refined goal-oriented policy

domain service is asked for resolution of domain expressions when the policy is activated.

All actions must be valid system commands or method invocations. Furthermore every possible action is associated with an event, where every event must be described in an *Event Definition Language (EDL)* with sufficient technical details to allow an implementation.

Figure 5 illustrates the general structure of an operational obligation policy. The general structure of

```
policy name type obligation for subject {
  <targets>
  action [ eventname ] [ if precondition ] {
    <operation>*
  } [ success postcondition1 ]                              5
    [ nosuccess postcondition2 ]
}
```

Figure 5: Operational policy structure

an authorization policy is similar, details are provided in the BNF-definition in Appendix C. An operational policy may have several **action** parts, triggered by different events. It is important to note that the *eventname* is optional (line 3). This construction allows an action to be triggered by a chaining mechanism only, that is the action is triggered when a postcondition of another policy fulfills the precondition of this action clause. If an event is provided, the chaining mechanism will not be activated.

The operational refinement of our goal-oriented example Goal03 is illustrated in Figure 6. Here line 1 gives name, type and subject of the policy, *~dvt* defines the domain of our department. The policy-object is specified in line 2, where the optional keyword **every** defines that all objects of class *ExtendedFileSystem* in the domain *filesys/home* are a target of this policy. This simple example requires only one action part, which is triggered by *use80*, defined in the event definition *disk_use*. The definition of this event is of no concern for the policy and will be discussed in detail

```
policy disk80 type obligation for ~dvt/root {
  for every ExtendedFileSystem in filesys/home
  action disk_use.use80 {
    removeGarbage()
  } nosuccess (~dvt/status.filesys == ALARM)
}

policy messages type obligation for ~dvt/root {
  for ~dvt/console
  action if (~dvt/status.filesys == ALARM) {
    notify(admin, file_alarm)
  }

}
```

Figure 6: Operational policy "disk80"

in the next section. If the action is triggered, the method *removeGarbage()* will be invoked on the target object. No further activity is triggered if the garbage removal was successful. If the execution of *removeGarbage()* was not successful, an appropriate postcondition is used to trigger a chained action to inform the human administrator of a potential problem. A special *messages* policy (defined in line 8–14) is used to provide chained action clauses for message forwarding to the human manager.

The example discussed so far does not deal with authorization policies. In general every activity described by an obligation policy must be authorized. In our environment this constraint is slightly weakened by providing a general authorization for the "root" object. We therefore do not require special authorization for automated management activities as long as the tasks are carried out by the root object.

4 Events

The automation of management tasks requires the formal definition of events. An event can result from three different activities:

a) monitoring of the system,

b) result of a policy activity, and

c) commands issued by human administrators.

The Event Definition Language we provide deals with events of type a) only, since these require special automated activity.

Monitoring is performed by special monitoring objects, where the behavior of these objects is described in EDL. Three generic types of monitoring events are defined:

- **Polling** based events require repeated activity of the monitoring objects. Events are created by the monitoring objects, based on the measured values.
- **Notification** events are created by the monitoring objects, based on notifications received from the managed objects.
- **Timing** events are created by timing objects. Conceptually, timing events can be considered as special polling events, but the provision of a separate type simplifies definition and implementation.

The general structure of a polling event definition is displayed in Figure 7. Here *name* defines the name of

```
event name type polling {
  operation ( parameter_1 , ... , parameter_n )
  every cycle
  [ filter ({median | medium | none}, window) ]
  on { == | != | <= | < | >= | > } threshold
  [ mode { static | dynamic } ]
  [ delay intervall ]
  [ single ]
  trigger eventname
  on ...
}
```

Figure 7: Event structure

the event definition, while the name of the event used to trigger a policy is defined in line 9 as *eventname*. The "def.event" notation in the PDL (see Figure 6, line 3) reflects the fact that every event specification can provide many events. The *operation* defines an interface method that provides the monitoring value. The polling rate in seconds is specified with *cycle*, an optional filter can be defined (line 4), followed by the option to define several technical details separately for each event (lines 5–9). The events described in EDL are automatically translated into an initialization procedure for a generic monitoring object, called *smart agent* [4]. The smart agent concept is introduced in Section 5.

Figure 9 displays the event definition *disk_use* for the file-system example. Monitoring is performed by

```
event disk_use type polling {
  unsigned short ExtendedFileSystem->Usage()
  every 300
  filter (median, 3)
  on > 80
    single
    trigger use80
}
```

Figure 9: EDL-example

access to the **readonly attribute** *Usage* of the *ExtendedFileSystem* interface (as defined in Figure 3). The polling period is 300 seconds and the measured values are filtered through a median filter with a window-size of three. If the output of the filter is above 80, an event *use80* is created once. Repeated events are

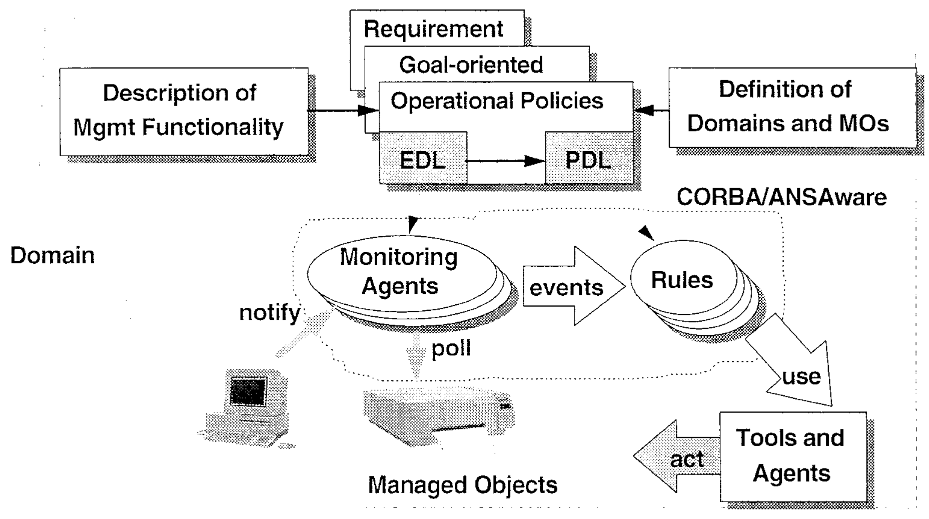

Figure 8: Management Framework

suppressed unless an explicit reset command is send to the monitoring agent. This functionality is indicated by the keyword **single** (line 6).

The EDL is considered to be an open language, where the set of keywords can be enhanced by new agent types with different features. While the PDL is also still considered to be an evolving language, we do not expect it to change and grow as dynamically as the EDL. This is due to the fact that the PDL describes the internal functionality of our management environment, while the EDL is used to describe the functionality of potentially external monitoring devices and their usage with our system. The openness of the EDL provides maximum flexibility, because improved monitoring agents may replace existing types without any change to the policy definitions as long as the new agent is capable to provide the necessary events. The price for the usage of different agents could be a reduced checking capability if the agent performs complex decisions autonomously. An additional advantage of the separation between policies and events is given by the fact, that technical details (polling frequency, different filters, ...) are invisible at the policy definition level. No changes to the policies are required if details of the event creation are changed.

5 Implementation

The implementation of our management framework consists of several components, as illustrated in Figure 8. Automated support for policy management and refinement is provided on all levels. The use of a hypertextsystem combined with an object oriented database allows the storage and management of policies. The operational level is additionally supported with tools for syntax-check, consistency between EDL and PDL, and translation of EDL and PDL into executable objects or rules.

We are currently investigating two different approaches for the implementation of the rules:

1. The rule-based software development environment Marvel [3] 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*. Policies are translated into a Marvel process-model while the data model contains the description of managed objects [5]. In this approach the distribution transparencies are provided by the middleware platform ANSAware.

2. Policies are translated into a set of CORBA objects. ORBIX is used as middleware platform. This approach allows the distribution of the rule objects as well as the monitoring agents, but chaining with automated locking of participating rules and objects, as provided by Marvel, is currently supported on a less sophisticated level.

5.1 Monitoring agents

Monitoring is performed by generic monitoring objects which are located as close to the managed object as possible. Our monitoring agents are "smart" in the sense that they perform monitoring, filtering and event creation independently. The components of our generic smart monitoring agent are illustrated in Figure 10. The behavior of the agent is completely controlled by an initialization file and invocations of operations provided at the management interface. Therefore recompilation is not required for standard types.

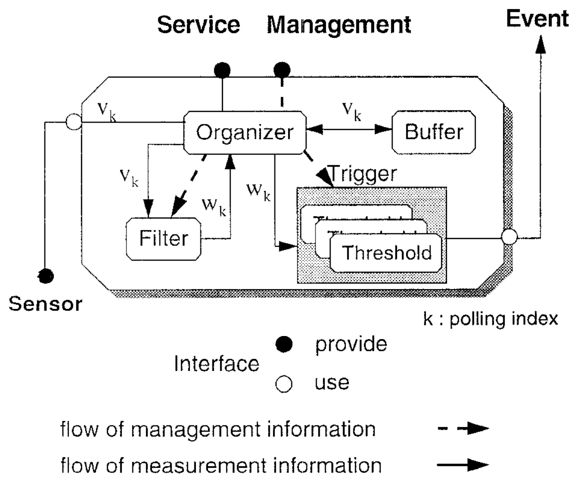

Figure 10: Smart monitoring agent

Sensor. Different agents will probably require very different sensors. To provide maximum flexibility, the sensor is not a part of the agent. The communication penalty that must be paid for this separation can be neglected for most measurement frequencies. Our agent implementation currently supports three types of sensors: 1) any UNIX shell command including a piped sequence of commands, all of which must be executable without user interaction; 2) a call to another ANSAware interface, and 3) a call to a CORBA object. Therefore monitoring is not restricted to a specific management environment (SNMP, CMIP,...), as long as an appropriate sensor is provided.

Organizer. This component coordinates the internal activity of an agent and provides the functionality for both service and management interface. At startup time the organizer reads configuration information from an initialization file and passes the appropriate parameters to the filter and trigger component (indicated with dashed arrows in Figure 10). Reconfiguration can be requested at the management interface. Monitoring is performed automatically with an adjustable measurement frequency. On every tick of the internal clock the organizer a) asks the sensor for a new value v_k and stores it in the buffer; b) calls the appropriate filter function with value v_k; c) stores the output w_k of the filter for access via the service interface and d) calls an appropriate trigger function with value w_k. To ensure data integrity, invocations on any of the interfaces are blocked until a complete measurement and evaluation cycle is finished.

Filter. This mechanism transforms an input stream of values (here v_k) into an output stream (w_k) according to a number of predefined filter functions. They include identity, median with window size N, and medium with window size N.

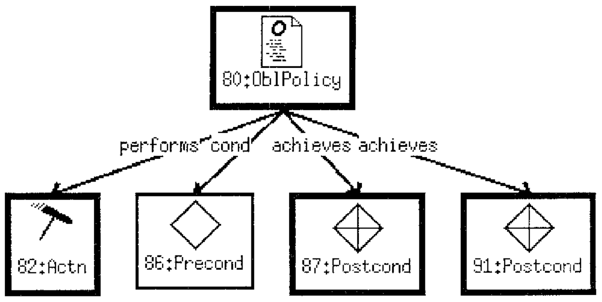

Figure 11: Structure subgraph for an obligation policy

Trigger. The trigger function performs an appropriate call whenever a predefined condition occurs on the filtered data stream.

Buffer. This component stores up to N previous measurements, where the buffer size N is adjustable through the management interface. The buffer is organized as a ring buffer, that is, as soon as the buffer is filled for the first time, it starts to override old values according to a FIFO strategy

6 Graph model of operational policies

Experimentation with our management environment soon demonstrated the potential complexity of chained policies. Especially for larger management applications we found the need for:

- semantic checking functions enabling the detection of ill-behaved policy sets incorporating, for example, non-exit loops, omitted or undesired logical connections, or unsatisfiable preconditions;
- "what-if"-type of analysis features that would allow an environment user to determine possible effects of a backward or forward chain of management activities prior to its execution;
- impact analysis features to predict the impact of planned changes to an existing policy system.

Based on these requirements we decided to extend the architecture depicted in Fig. 8 by three further components:

1. a graph model of the process semantics of operational policy and event specifications,
2. a compiler mapping operational specifications into their semantic graphs, and
3. implementation of analysis and manipulation functions on such graphs.

6.1 Semantic Graph Model

The design of a semantic model for operational policy and event specifications is relatively straightforward. The *structure* of a policy is represented by different **policy node types** and **policy edge types**. The node types are: *obligation policy*, *permission policy*, *precondition*, *activity*, and *postcondition*. The

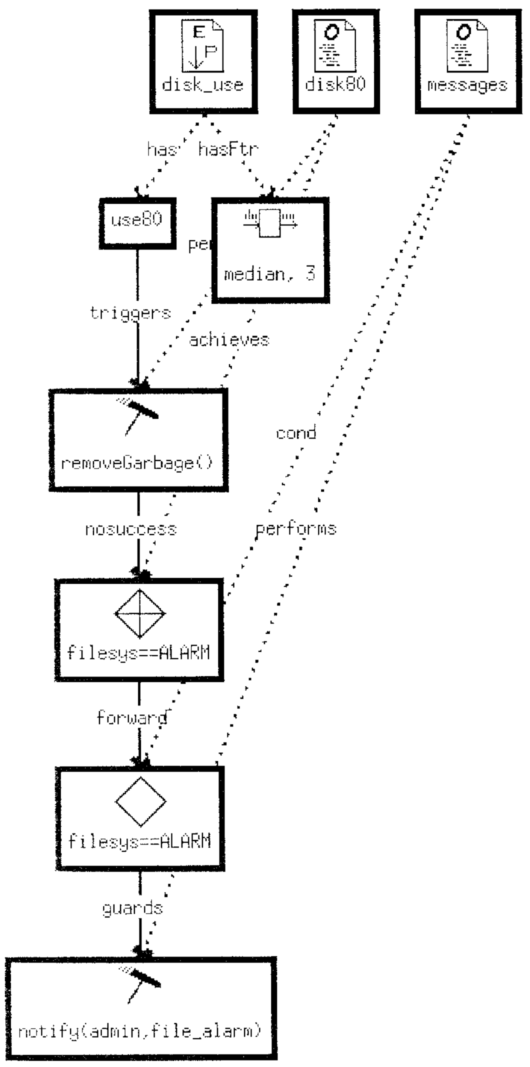

Figure 12: Graph representation of chained policies

edge types are *cond* (relating a policy with its precondition), *perform* (relating a policy with a management activity), and *achieves* (relating a policy with its possible postconditions). An example of a policy structure graph is shown in Fig. 11.

Similar to the structure of policy definitions, event definitions are semantically represented in the graph model. Again we represent the different types of events, triggers, filters, and sensors by different **event node types**. The various consists-of relationships between an event and its constituents are modeled by different edge types.

Apart from the structure of policies and events, the graph model also captures their *dynamic behavior* using the same node types but different edge types. For interconnecting policies we have the edge types *forward*, *backward* and others, relating a postcondition node of one policy with the precondition node of another or the same policy, respectively. To interconnect events and policies, we use the edge types *may_trigger* and *activate* relating the trigger of an event with the precondition of a policy.

The example policies, introduced in Section 3.3, are displayed as policy-graph in Figure 12. Here the edges for representation of structural information are drawn with dotted lines. Edges used for representation of dynamic behavior are drawn with solid lines. The chain of activities, starting with a trigger and potentially resulting in a notification to the administrator, is clearly visible. For more complex applications parts of the graph may be suppressed (e.g. no structural information) to increase the readability. Compression of pre- and postconditions into one node is also supported.

6.2 Creation, Manipulation and Query of Semantic Graphs

The manual implementation of tools compiling policy and event definitions into semantic graphs and inspecting such graphs for certain properties would require a substantial effort in terms of manpower and time. To avoid this effort, we employed a powerful graph rewriting system, PROGRES, that was designed and implemented at the University of Aachen, Germany [8]. This system provides a visual rule-oriented language for graph-based data definitions and for the declarative specification of graph manipulation functions. The language is part of a programming environment that provides assistance for the creation, semantic analysis, debugging, and consistent manipulation of graph models relying on a graph grammar paradigm and graph rewrite rules.

```
production AddForward ( post : Postcondition ) =
   begin
      obl_node '1 = post;
      obl_node '2 : Precondition;
      '1 +> '2 : forward;
   end
   ::=
   begin
      obl_node 1' = '1;
      obl_node 2' = '2;
      1' -> 2' : forward;
   end
   condition '1.PostExpr = '2.PreExpr;
end;
```

Figure 13: Graph production rule

In our management environment this graph language is used for consistent creation, manipulation and query of policy graphs. Figure 13 gives an example for a production rule that adds a *forward* edge to an existing graph if a matching *Precondition* is found for a given *Postcondition*. In lines 3–5 the graphical requirements are specified, here a *Precondition* node without a *forward* edge to the given *Postcondition* node is required (the "+ >" notation represents a crossed arrow "− >"). An edge between both nodes is added (line 11) if the condition is fulfilled (line 13), that is the expression attributes of both nodes are equal.

```
transaction AddPostcond ( po_id : string;
        success : boolean; status : string )
=
  use id : Postcondition
  do                                                    5
     choose
      when (success = true)
      then
        AddSuccess ( po_id, status, out id )
     else                                               10
      when (success = false)
      then
        AddNoSuccess ( po_id, status, out id )
     end
   & choose                                             15
      loop
        AddForward ( id )
      end
     end
  end                                                   20
end;
```

Figure 14: Graph transaction rule

Based on this production a transaction is defined (Figure 14), that adds a postcondition node (line 9 or 13) to an existing policy and checks automatically for possible *forward* edges (line 15–19). If, for example, the postcondition "filesys==ALARM" in Figure 12 is added to the policy "disk80" the system will automatically query all preconditions for a potential match and add appropriate edges. The edge "forward" is therefore created automatically when the postcondition "filesys==ALARM" is entered. The potential for erroneous policy design is significantly reduced with this kind of automatic support.

7 Conclusions

Main objectives of the research work described in this paper were to design a management environment that respects fundamental characteristics of distributed applications and supports the human administrator in two ways: a) clerical management tasks should be performed automatically and b) the environment should provide comprehensive guidance, insight, construction and analysis support to human administrators bearing responsibility for technical systems management.

We presented a three level policy hierarchy to support a stepwise refinement from an informal strategic level to a formalized operational level. A separate policy and event definition on the lowest level provides maximum flexibility for design and inclusion of different monitoring agents.

A recent extension of our management architecture proposed the use of a graph-based semantic model for policy and event definitions. The graph model supports design and maintenance of complex policy sets. Further experimentation with the extended management environment will provide experience with the application of our graph model to different applications. Further improvement of the graph grammar can be expected from these insights.

References

[1] ISO/IEC. *Information technology – Basic reference model of open distributed processing: Prescriptive model*, 1995.

[2] ISO/IEC JTC1/SC21. *Open Distributed Management Architecture, Working Draft 3*, Oct 1995.

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

[4] T. Koch. Rule Based Management Architecture with Smart Agents. In M. Sloman and T. Usländer, editors, *Proc. of the Int. Workshop on Services for Managing Distributed Systems*, Karlsruhe, September 1995. Fraunhofer IITB.

[5] T. Koch and B.J. Krämer. Rules and agents for automated management of distributed systems. *Special issue of the Distributed Systems Engineering Journal on Distributed Systems Management*, 1996.

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

[7] J.D. Moffet. Specification of Management Policies and Discretionary Access Control. In Sloman [11], chapter 17, pages 455–480.

[8] A. Schürr. Rapid Programming with Graph Rewrite Rules. In *Proc. USENIX Symp. on Very High Level Languages (VHLL)*, pages 83–100, Santa Fee, Oct 1994. USENIX Association.

[9] M. Sloman. Management Policy Service for Distributed Systems. In *Proc. of 3rd Int. Workshop on Services in Distributed and Networked Environments*, Macau, June 1996. IEEE.

[10] M. Sloman and K. Twidle. Domains: A Framework for Structuring Management Policy. In Sloman [11], chapter 16, pages 433–453.

[11] Morris Sloman, editor. *Network and Distributed Systems Management*. Addison-Wesley, 1994.

[12] R. Wies. *Policies in Integrated Network and Systems Management: Methodologies for the Definition, Transformation, and Application of Management Policies*. PhD thesis, University of Munich, Germany, 1995.

A General Language Elements

```
<operation>        ::= <identifier> <parameters>
<parameters>       ::= "(" [ <parameter> { ", "
                               <parameter> }* ]")"
<parameter>        ::= <string_literal>
                   |   <integer_literal>
<integer_literal>  ::= <number>+
<number>           ::= "0" | "1" | "..." | "9"
<string_literal>   ::= "<identifier>"
<identifier>       ::= <alpha><alphanumber>*
<ext_identifier>   ::= ["/"] {<identifier> "/"}*
                         <identifier>
                         ["->" <operation>]
<alpha>            ::= "A" | "B" | ...
                         | "Z" | "a" | ... | "z"
<alphanumber>      ::= <alpha>
                   |   <number>
                   |   "_"
```

B EDL Syntax

```
<event>          ::= <event_header>
                     "{" <event_body "}"
<event_header>   ::= "event" <identifier>
                     "type" <event_type>
<event_type>     ::= "polling"
<event_body>     ::= <operation>
                     "every" <integer_literal>
                     [ "filter"
                       "(" <filter_type> ","
                           <integer_literal> ")"]
                     <event_trigger>*
<filter_type>    ::= "median"
                 |   "medium"
                 |   "none"
<event_trigger>  ::= "on" <operator>
                          <integer_literal>
                     [ "mode" <mode_type> ]
                     [ "delay" <integer_literal>]
                     [ "single" ]
                     "trigger" <identifier>
<mode_type>      ::= "static"
                 |   "dynamic"
<operator>       ::= "=="
                 |   "!="
                 |   "<="
                 |   "<"
                 |   ">="
                 |   ">"
```

Remarks:
The current version supports polling events only.

C PDL Syntax

```
<policy>          ::= <obl_policy>
                  |   <auth_policy>
<obl_policy>      ::= <obl_header>
                      "{" <obl_body> "}"
<auth_policy>     ::= <auth_header>
                      "{" <auth_body> "}"
<obl_header>      ::= "policy" <identifier>
                      "type" "obligation"
                      "for" <ext_identifier>
<auth_header>     ::= "policy" <identifier>
                      "type" <auth_type>
                      "for" <ext_identifier>
<auth_type>       ::= "permission"
                  |   "prohibition"
<obl_body>        ::= <targets>
                      <action>*
<auth_body>       ::= <targets>
                      <identifier>*
<targets>         ::= "for" [ "agent-of" ]
                            <ext_identifier>
                  |   "for" "every" [ "agent-of" ]
                            <ext_identifier>
                            "in" <ext_identifier>
                  |   "for" "all" [ "agent-of" ]
                            <ext_identifier>
                            "in" <ext_identifier>
<action>          ::= <action_header>
                      "{" <action_body "}"
                      <action_trailer>
<action_header>   ::= "action" [ <identifier> ]
                      [ "if" <expression> ]
<action_body>     ::= <operation>*
<action_trailer>  ::= ["success" <expression>]
                      ["nosuccess" <expression>]
<expression>      ::= <operation>
                  |   <string_literal>
                  |   <integer_literal>
                  |   "(" <expression> ")"
                  |   "!" <expression>
                  |   <expression>"&&"<expression>
                  |   <expression>"||"<expression>
                  |   <expression>"=="<expression>
                  |   <expression>"!="<expression>
                  |   <expression>"<="<expression>
                  |   <expression>"<" <expression>
                  |   <expression>">="<expression>
                  |   <expression>">" <expression>
```