Distributed Systems Management
Software-in-the-Loop

Bernd Krämer and Thomas Koch
FernUniversität
Faculty of Electrical Engineering
58084 Hagen, Germany

Abstract

IT experts expect open distributed processing to become the predominant computing infrastructure in the late nineties. All computer supported work places of large enterprises and organizations will then be networked and will be integrated into cross-regional and cross-sector business and information processes. The size and complexity of such applications, the local autonomy, distribution and heterogeneity of participating subsystems, and their asynchronous interaction, however, require new architectures, strategies, and tools for their technical management. In previous work we placed a production rule interpreter into the monitoring, decision, control action loop to provide a flexible, operational semantics of well-understood management policies. In this article we extend this work in two directions. First we map the structure and dynamic behavior of policies into a graph representation. This semantic representation enables a systematic prediction of the effects of policy executions and allows for a better impact analysis in case of policy changes. Then we introduce a declarative event definition mechanism. It supports a causal and temporal correlation of individual events and serves to instantiate and adapt a predefined generic event handler to the specific needs of the actual management application. Such event handlers join in the interaction between monitoring agents and policy interpreter. By event correlation they may reduce the number of events triggering management actions significantly and help to filter secondary events.

Keywords: Distributed systems, technical management, management policy, event handling

1 Introduction

In a growing number of application areas we observe a trend to integrate existing data processing infrastructures into computer networks supporting client-server cooperation of autonomous component systems. These areas include business and office data processing, information and communication systems, process automation, and scientific computing, to name just a few. By the late nineties it is expected that all computer supported work places of large enterprises will
be networked through LANs, MANs or WANs. The motivations behind this trend are a better integration of business processes and information systems and an increased flexibility in (re-) configuring application building blocks.

From the perspective of the application developer, the heterogeneity of the underlying hardware, operating system, and communication system will be masked by a variety of middleware systems such as ANS Aware [1], CORBA implementations [12], OSF's DCE [14], Microsoft's OLE/DCOM, and others. These middleware platforms will also support the required openness of the network, the autonomy and interoperability of its component systems, a seamless integration of legacy systems, and various types of transparency.

On the side of the technical management of distributed applications, however, the situation looks completely different. Management activities traditionally include installation, monitoring, control, time and security management, spooling, accounting, alarm forwarding, and reconfiguration of a system's hardware and software resources. A distributed environment adds new tasks including

- adaptive load balancing [9],
- allocation, replication and migration of data and services,
- dynamic reconfiguration,
- cooperation control,
- reliability management, or
- distributed time and cross-platform security management [14].

Conventional management tools are unable to cope with these new requirements, and recent workstation-based management tools suffer from a number of limitations and weaknesses. Hewlett Packard's HP Open View, for example, or Solstice from SunSoft offer no or only rudimentary mechanisms, respectively, for defining and implementing application specific management policies. They perform monitoring tasks but are unable to automatically react to critical situations according to predefined policies. Rather, an alarm is sent to the human manager who has to take the appropriate management decision. To our knowledge, none of the systems allows the correlation of events to minimize their number by abstraction and filtering in case of secondary and tertiary events referring to situations that were already signaled. These platforms have limitations with respect to interoperation with other management tools; they are also not operational on all platforms and they typically rely on the relatively simple network management protocol SNMP.

Distributed systems require new management architectures that provide an integrated management concept supporting corporation-wide or business process-specific management policies. These architectures must allow for the sharing of a common management information base and enable the seamless integration of specialized and well-tried management tools. A homogeneous interpretation of policies that is independent of a particular management view or resource it
refers to requires a uniform policy model. This is also a necessary prerequisite for the automated detection of local and non-local policy inconsistencies and inferences and a computer supported enforcement of management policies. Integration further requires that management information about heterogeneous resources, workstations and servers is collected and maintained homogeneously. Management related events need to be detected and sometimes related to each other independent of the platform they originate from. A decentralized management architecture and well-designed organizational concepts serve scalability and the flexible adaptation to the changing needs of the distributed computing infrastructure. They also help prevent performance bottlenecks of management components. The integration of existing management tools requires standardized management interfaces and uniform communication mechanisms.

In the main body of this article we introduce a management architecture that includes objects, interfaces and domains as conceptual building blocks. The architecture offers a core of generic management service objects for monitoring, global and local event handling, domain management, and policy execution. A novel event definition language provides a declarative mechanism to configure generic monitoring agents and event handlers to the needs of specific management applications. Policies are formalized and automatically executed by means of a policy definition language and enforcement objects, respectively. Being defined separately for the sake of readability and ease of modification, logically related policies and events are glued together on the operational level through a predefined triggering mechanism. An abstract view of the automated management loop implemented by this architecture is depicted in Fig. 1. Before presenting technical details of this architecture, we provide some further motivation and briefly review earlier results in the following subsections.
1.1 Networked Management Activities

To better understand management issues in a distributed setting, consider a client-server application in which service provision may be interrupted shortly, e.g., to install a software upgrade for a particular program server. Assume further that this requires a server shutdown. A typical management policy might require that no data must be lost during server upgrades. To satisfy this policy, the system administrator on the server site may want to unbind all clients of that server and withdraw outdated service offers prior to server shutdown. Before unbinding a client, the manager ought to check whether its users are logged out, no references to service objects exist and no file systems are exported. After successful installation of the software update, the server must be restarted again, the new services must offered, and a re-activation notification should be sent to all clients.

Figure 2 shows the dependency graph for the described software upgrade process and two other management tasks - server Migration and network Maintenance - sharing management activities with the former. The overlap of the three different management tasks is illustrated with shaded boxes inside every activity.

![Figure 2: Dependency graph of different management activities](image-url)
1.2 Rule-Based Management

Automation of such management tasks requires a mechanism that is capable of dynamically creating proper activity chains taking both relevant parts of the actual system state and the semantics of individual policy rules into account. These characteristics suggested to consider production rules as appropriate candidates for modeling management activities. We chose the process-centered software engineering environment Marvel [7] as implementation platform for our distributed management architecture. Marvel is particularly suited because:

- it uses a precondition/activity/postcondition paradigm in connection with an object base; both allow the policy designer to make external effects of an activity execution visible and separate the maintenance of management data from the actual implementation of an activity;

- it offers an overloading mechanism that allows a rule name to be used for semantically similar tasks on different object types;

- it enables both manual and automatic activity invocation (e.g., by a system administrator or external monitoring devices);

- it provides a flexible tool inclusion mechanism by which existing management tools and scripts can be used to implement activities; and

- it supports the dynamic creation of forward and backward chains of management activities, depending on (relevant parts of) the system state and given input data.

Extensive experimentation with our Marvel-based prototype architecture has, however, revealed some deficiencies including:

1. insufficient validation support for large rule systems,

2. lack of predictive mechanisms answering “what-if” type of questions prior to executing particular rule chains,

3. lack of analysis techniques that allow one to predict the impact of planned policy changes.

1.3 Adding Graph-Based Semantic Modeling and Analysis

To cope with these problems, we decided to go a step further towards knowledge-based system management and adapt semantic network techniques to our management architecture. We propose to map both structure and dynamic behavior of policies into a dedicated graph model. The graph model was designed and prototyped with the aid of the visual and rule-based graph rewriting system PROGRES [15]. It comes with a number of powerful tools supporting graph rewriting, analysis, transaction, and query mechanisms.
In the following section we sketch the management model underlying our architecture and its functionality. Section 3 reviews the role and use of Marvel as a kernel component of the prototype implementation of this architecture. Section 4 briefly presents a new mechanism, called “monitoring agents”, which serves to monitor distributed resources autonomously and raise signals or trigger policy rules whenever predefined threshold values or deadlines are reached. In Section 5 we introduce the graph model and discuss its potential to overcome the weaknesses of the current management prototype. Section 6 finally introduces linguistic and operational means for specifying and rapidly implementing complex event handlers.

2 Management Model

Following the example of a number evolving international standards [5, 6], we rely on an object based approach to organize our management model.

We distinguish two elementary kinds of objects: managed and managing objects. The latter designate objects that may initiate management activities because they possess a degree of authority to ask for information and to perform particular management tasks. Managed objects are simply the target of management activities. Depending on the task being executed, one object can be both in the managed and managing role at different times. Two objects can interact through a link established between the managing interface of one and the managed interface of the other object. Objects may have multiple managed and managing interfaces [6].

To deal with the size of large distributed systems, the management model groups objects into domains. Domains define projections on the set of co-existing objects. Domains support modularization and the representation of different views on the same system. An object can be a member of different domains at the same time.

Most management knowledge is currently memorized in the heads of system administrators. Automation of management activities clearly requires machine readable descriptions of diagnostic and procedural management knowledge. We call a rule describing a management activity together with conditions under which it can be employed and desired effects of its execution a policy [11]. In [10] we propose a three-level policy structure to document and formalize management knowledge at a strategic, a goal-oriented and an operational level.

On the strategic level policies are typically described in prose. A simple example would be:

The administrator should remove all temporary files and core dumps older than 24 hours once a day.

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. The goal-oriented level offers templates. Parts of
their contents are automatically derived from the strategic level description. On the operational level a Policy Description Language (PDL) and an Event Definition Language (EDL) are offered. PDL descriptions are translated into executable rules whose activity part may invoke management tools. EDL parts serve the definition of system events which signal significant changes in the state of a managed object and trigger the execution of appropriate policy rules. Events can be caused by: a) the attainment of threshold values or deadlines in the process of resource monitoring, b) activity executions, and c) human interaction (e.g., by means of system administrator commands). The separation of policies and events has the advantage to allow dynamic adjustments of polling frequencies, monitoring filters, etc. while preserving policy rules.

3 Using Marvel to Implement the Management Model

To construct a Marvel-based management environment as schematically illustrated in Fig. 3, the system manager has to define a management data model, a management process model, and tool envelopes.

Management data model. This part of an environment definition must reflect the object types, attributes, interfaces, and relationships described in the previous section. The structure of management data is defined by an object-oriented class hierarchy. Its instances are maintained persistently in an object base which is part of the Marvel environment. Part of a data model is depicted in Fig. 4 for illustration purposes. Object class G_OBJECT represents the most
G_OBJECT :: superclass ENTITY;
  init_status : (Initialized, Plain) = Plain;
  configure  : (Enabled, Done, Problems) = Enabled;
end

G_HOST :: superclass G_OBJECT;
  status     : (Up, Down, NotChecked) = NotChecked;
  agents     : set_of G_MON_AGENT;
end

G_MON_AGENT :: superclass G_OBJECT;
  status     : (Up, Down, NotChecked) = NotChecked;
  conf_file  : string = "";
end

SMART_AGENT :: superclass G_MON_AGENT;
  type       : (ANSA, CORBA) = ANSA;
  ev_type    : (Polling, Notification, Timing) = Polling;
end

FS :: superclass G_OBJECT;
  status     : (OK, Search, Garbage, Umount) = OK;
end

HOST :: superclass G_HOST;
  opsys      : string = "";
  ...
  fs         : set_of FS;
end

Figure 4: Parts of a data model

general definition; it is inherited by all the other classes. Class ENTITY is provided by Marvel and serves as a root for the whole data model. Besides the attributes representing common characteristics of managed objects, each class may contain additional attributes to control the coordination of rule activation. The configure attribute of class G_OBJECT is an example for this kind of attributes.

Management process model. As mentioned before process knowledge is derived automatically from policy and event definitions given at the operational level. Policies are represented as precondition/activity/postcondition-rules, while events definitions are compiled into monitoring agents, which were not part of the original Marvel environment. A Marvel-based environment acts as a server which executes the process model on request of its clients which can ask for the activation of a particular rule. In the process of matching the precondition of the selected rule with the current state of relevant management data, the rule interpreter of Marvel may construct a forward or backward chain of management activities and automatically executes a successfully built chain. To avoid inconsistent object states due to concurrent rule activation, every object is locked while being used by an active rule. This concurrency control mechanism is extremely useful to cope with concurrent management requests by autonomous clients located on different nodes.

The rules presented in Fig. 5 implement the strategic policy statement as presented in the previous section. To enable future reuse, we chose to define two
separate rules, \texttt{fssearch} and \texttt{fsclean}, responsible for identifying garbage files and for their removal, respectively. Each rule defines: 1) the object classes affected by the rule, such as \texttt{FS} (cf. also Fig. 4) occurring in parentheses after the name of the rule; 2) an optional ad hoc query through the object base (in our example the query is used to find the \texttt{HOST} object on which the target file system is mounted); 3) the precondition (here rule \texttt{fsclean} requires the host to be \texttt{Up} and the status of the file system to be \texttt{Garbage}; the keyword \texttt{no\_chain} prohibits any participation of this policy rule in a chaining process); 4) the activity, enclosed in curly brackets, and 5) the possible effects of the activity execution on the object base in terms of an alternative list of assertions reflecting different outcomes of an activity.

\begin{verbatim}
fssearch[?f:FS]:
(exists HOST ?h
    suchthat(member(?h.fs ?f))
    :
    (and no\_chain (?h.status = Up)
        (or
            no\_backward (?f.status = Search)
            no\_chain (?f.status = OK)
        )
    )
    {FILESYS search ?f.Name ?h.Name}

(?f.status = OK); # RC 0: nothing found
(?f.status = Garbage); # RC 1: found garbage!

fsclean[?f:FS]:
(exists HOST ?h
    suchthat(member(?h.fs ?f))
    :
    (and no\_chain (?h.status = Up)
        (?f.status = Garbage)
    )
    {FILESYS clean ?f.Name ?h.Name}

(?f.status = OK); # RC 0: cleaned FS
(?f.status = Garbage); # RC 1: problem, didn't clean it!
\end{verbatim}

Figure 5: Rules for searching and cleaning a file system

\textbf{Tool envelopes.} Marvel is open to the integration of prefabricated tools and methods. They are encapsulated in so-called envelopes \cite{3}. An envelope specifies admissible types of inputs and outputs of tool invocations; it uses Unix shell scripts to activate one or more tools in their proper environment; it also matches the tools’ relevant return codes with the postconditions of rules using that tool. Through the envelope mechanism tools can be invoked automatically from within rules. In addition, some form of consistency checking is possible with respect to data and rule definitions.
4 Monitoring Agents

In our management model we associated the notion of *event* with any a significant change in the state of a managed object. Such state changes could, for instance, be detected by implanting suitable decision procedures, event creation and signaling mechanisms into each managed object. This solution has, however, the drawback that the code of the implementation of managed objects needs to be changed. This is not only difficult as resource drivers and operating systems of workstations and servers may be written in different languages; it is also undesirable as the integrity of the driver or systems software may be invalidated or it is even impossible if the source code is not available. As a better alternative we propose generic monitoring objects to be located in the physical neighborhood of the managed objects. These monitoring objects only assume the existence of standardized management interfaces to acquire information about the state of a managed object. Examples of such standards include SNMP, the OSI management model, or more recent HTML based control interfaces of network resources. These monitoring agents are capable to create appropriate events and signal them to the policy enforcement agent. In our Marvel based prototype this is realized by the Marvel rule interpreter. The individual behavior of each agent is defined by means of EDL specifications.

Currently we support three generic types of monitoring events in EDL:

- **Polling** based events require repeated activity of the monitoring objects. Events are created by monitoring objects based on measured values.

- **Notification** events are created by monitoring objects based on notifications received from 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.

These event types are implemented by corresponding agent types that perform monitoring, filtering and event creation independently. The common structure of these agents is illustrated in Fig. 6. In the sequel we shall discuss the behavior of polling agents, only, as it reveals the most interesting functionality. The other two agent types are similar in behavior but less complex.

**Sensor.** Different agents will probably require very different sensors. To provide maximum flexibility, the sensor is not part of the agent but is a separate object that is activated by the agent via a system call. The communication penalty to be paid for this separation can be neglected for most measurement frequencies. Only for monitoring tasks with a high polling frequency ($f_p > 10$ Hz) the use of a specialized and performance optimized agent is recommended. The sensor can be any executable program that provides a useful value. 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 exe-
Figure 6: Structure of a monitoring agent

cutable without user interaction; 2) a call to another ANSAware interface, and 3) a call to a CORBA object.

**Organizer.** This component coordinates the internal activity of an agent and provides the functionality for both service and management interface. At start-up 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 Fig. 6). Reconfiguration can be requested at the management interface, either by provision of another configuration file or by an explicit Set_Parameter function. 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 \( f_k \) of the filter for access via the service interface and calls an appropriate trigger function with value \( f_k \). 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 (\( f_k \)) according to a number of predefined filter functions. They include identity, median with window size, and medium with window size.

**Trigger.** The trigger function performs a call to the Marvel system whenever a predefined condition occurs on the filtered data stream. An arbitrary set of threshold values with different characteristics for each threshold can be programmed based on a predefined set of parameters.

**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.

Generic agents are currently implemented both as ANSAware capsules and Or-
bix objects [4]. Monitoring agents must also be interfaced with the Marvel system. This is achieved by a collection of six generic rules, an object class SMART_AGENT, a specialization of class G_MON_AGENT (cf. Fig. 4), and tool envelopes encapsulating ANSAware capsules or Orbix objects.

4.1 Polling events

Polling events are created by generic monitoring agents. For initialization of these agents the event definition must contain behavioral instructions. EDL is considered to be an open language, where the set of keywords can be enhanced by new agent types with different features. The current implementation supports basic filtering and triggering mechanisms as well as the adaptation functionality described in the previous section. The events described in EDL are automatically translated into an initialization procedure for a generic monitoring agent.

Figure 7 displays the event type definition *thr-example* for a threshold based adaptation of a given polling frequency. Monitoring is performed by access to the **readonly attribute** *Usage* of an arbitrary *ex_interface*. The basic polling period is defined to be 10 seconds and the measured values will be filtered through a median filter with a window-size of three. Any exception raised during an invocation of the *Usage()* method will trigger a general Alarm event (line 5-7) named *thr-example.Alarm*. If the filtered value exceeds 60 or 80, an event named *thr-example.T1* or *thr-example.T2*, respectively, is created. The range for an increased polling frequency is defined as follows: between 0 and 50 the basic polling period of 10s is used. Beginning at 50 (line 11) the frequency is linearly increased until it reaches the value 60. Beyond that value, the increased

```plaintext
monitor thr_example type range
for all ex_interface in / {
  unsigned short ex_interface->Usage()
  every 10
  filter (median, 3)
  exception ANY
  single
  trigger Alarm
  on > 60
    factor 1.5
    low 50
    trigger T1
  on > 80
    factor 3
    low 60
    trigger T2
}
```

Figure 7: Example for a monitoring event definition
frequency is left unchanged because no high value is defined for T1. From 65 the frequency increase for T2 becomes active, resulting in a polling period of 3.33s when the value 80 is reached.

5 Graph Model of Operational Management Rules

Extensive experimentation with various prototypes of Marvel-based management environments confirmed the use of our management model. Marvel also satisfied many of our expectations into the rapid prototyping of a management environment implementing this model. For larger management applications, however, the current design of Marvel revealed some weaknesses including the lack of:

- semantic checking functions enabling the detection of ill-behaved rule sets that incorporate, 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;
- analysis features to predict the impact of planned changes to an existing policy system.

As we have no access to the internal representation of Marvel rules and the implementation of the rule interpreter, we decided to extend the architecture depicted in Fig. 3 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. an implementation of analysis and manipulation functions on such graphs.

5.1 Semantic Graph Model

The design of a semantic model for operational policy and event specifications is relatively straight-forward. 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 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. 8.
Event definitions are also represented in the graph model. We distinguish between different types of events, triggers, filters, and sensors using 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 complete semantic graph for two logically related policies and a polling event is depicted in Fig. 9. The structure of one of the policies was shown in Fig. 8 already.

5.2 Creation, Manipulation and Query of Semantic Graphs

The manual implementation of tools compiling policy and event definitions into semantic graphs would require a substantial effort in terms of manpower and time. This is also true for algorithms that inspect such graphs to certify or falsify certain properties and maintain the consistency between the rule base of Marvel and the semantic model in case of changes.

To avoid this effort, we employed a powerful graph rewriting system, PROGRES, that was designed and implemented at the University of Aachen, Germany [15]. 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.

An incomplete PROGRES specification of semantic policy and event graphs together with pertinent creation and manipulation operations is listed in Ap-
Figure 9: Semantic graph for two policies and one event
Appendix A. This specification was used to generate the graphs presented in this section. It also illustrates further features of PROGRES, such as different kinds of associations well-known from relational database models, graph grammar production rules, and transactions, which we cannot discuss in detail here.

Based on the graph-model, a what-if query issued prior to the automatic activation of a management activity could, for example, be answered by:

- computing a feasible forward or backward chain along the graph model,
- visualizing a successfully computed chain as a tree projection on the behavior part of the semantic, and
- allowing the administrator to navigate through the projected subgraph.

Non-exit loops can be detected by means of a query that identifies all cycles that connect preconditions, activities, and postcondition through forward, guards, and success edges only. A reachability query would attempt to find a path from the actual to the designated management state or activity. The prediction of the impacts of an intended change to a given policy model could be supported on two levels: 1) on the object level one could contrast the old and the new version of the semantic graph; 2) provided that all admissible graph rewrite rules had been categorized with respect to their effect on a given graph, impact analysis on a meta level would simply consist in the identification of the requested rewrite rule and a query to its “change impact” attribute.

Although much of this work is still ahead of us, first experiences with PROGRES’s powerful query, visualization, and transaction mechanisms show great promise that behavior prediction, policy checking, and impact analysis functions can be implemented equally rapidly as the Marvel-based kernel. The meta level analysis feature sketched in the previous paragraph requires, however, a substantial amount of further practical experience with the extended management architecture and related work developed elsewhere.

6 Enhanced Event Handling

Management of distributed systems often requires the evaluation of complex situations usually based on incomplete knowledge on the overall system state. In Sections 3 and 5.1 we have presented a policy and event model in which individual events may trigger a chain of policy actions. This approach is somewhat simplistic as it cannot deal with causal and temporal relationships between individual events appropriately. To obtain a proper interpretation of the situation at hand, it is often necessary to correlate several events and generate an event that embodies a higher semantic content than the individual events bear. Event correlation thus reduces the number of events propagated to the management executive, be it a human manager or a managing object.
Event specification and correlation is specified in terms of a declarative mechanism, EDL. A generic event handler is then able to evaluate such specifications and generate new events derived from correlating other events. The introduction of event handlers leads to a modification of our management loop as depicted in Fig. 10.

![Diagram of Extended Management Loop](image)

**Figure 10:** Extended management loop

### 6.1 Abstract Operators for Event Correlation

The abstract syntax of the operators supporting event correlation is defined below. EDL includes, of course, more syntactic sugar to ease the administrator’s task [10]. A simple example of an EDL specification is discussed in the following subsection.

Events are inductively defined as follows:

1. basic events resulting from system monitoring, a policy action or a command issued by a human administrator are events;
2. if $a$ and $b$ are events, then $a\&b$ denotes an event that occurs when both $a$ and $b$ occurred;
3. if $a$ and $b$ are events, then $a|b$ denotes an event that occurs when $a$ or $b$ occurred;
4. if $a$ and $b$ are events, then $a;b$ denotes an event that occurs if $b$ occurred after $a$;
5. no other events exist.

This definition suggests that we assume a notion of time as it does not seem useful to consider the occurrences of events over an unlimited observation period.
In addition, the *after* operator ";;" relies on an ordering of event occurrences to be meaningful. In distributed systems, however, an objective linear ordering of events created independently at different locations is unrealistic. To escape from this gap we could rely on a special protocol for order preserving communication between distributed objects [2]. Another option would be to rely on synchronized distributed clocks as provided, e.g., by DCE. We use a third approach by associating a locally unique time-stamp with each event as it is recognized by a specific event handler. Based on these associated time-stamps we can now extend our definition of event correlation by the specification of a time period $T$ within which the intended event correlation must occur. For example, the event expression $(a & b) \mid (b; c)[T]$ means that a new event is generated at time $t_3$ if $a$ occurred at time $t_1$ and $b$ occurred at time $t_2$ or if $b$ occurred at time $t_1$ and $a$ occurred at time $t_2$ or $c$ occurred at time $t_2$. In all cases we have that $t_1 < t_2 \leq t_1 + T$ and $t_2 \leq t_3$, i.e., the new event can be generated with some delay after the last relevant event has been received.

The correlation of events not only extends the functionality of the management architecture by new capabilities but also introduces some freedom in the choice of linguistic and operational mechanisms. Some preconditions of policy rules can now be evaluated by means of event correlation. This may significantly relieve the rule interpreter of constraint evaluation. Long activity chains can also be broken into short activity loops with interspersed event correlations.

### 6.2 Example

```plaintext
event comp.example {  

    thr.example.Alarm
    delay 5m trigger AlarmReset

    x:thr.example.Alarm ; y:thr.example.Alarm [2h]
    if (?x == ?y)
        trigger Message6

    x:thr.example.Alarm ; y:thr.example.Alarm [30m]
    if ((?x == ?y) && ((?y < 8.00) || (?y > 17.00)))
        trigger Pager1

    x:comp.example.Message6 & y:comp.example.Message6 [30m]
    if (?x != ?y)
        trigger Pager2

    x:thr.example.T2 & y:thr.example.T2 [15m]
    if (?x != ?y)
        trigger Mail5

}
```

Figure 11: Example for composite events
Figure 11 shows an example for event correlation as it is expressed in EDL. The events used in this example are created by a monitor object whose properties were defined in Fig. 7. If the monitor object encounters any problems with accessing its corresponding management interface, that is, if any exception is raised by the middleware environment, an event of type \texttt{thr.example.Alarm} is triggered exactly once (indicated by the keyword \texttt{single} in Fig. 7). Whenever the event handler receives such an event, it will wait for 5 minutes (lines 3, 4) and then trigger a reset alarm. This alarm will activate a corresponding policy that, in turn, sends a command to the monitoring agent to reactivate the \texttt{Alarm} event.

If an exception occurred twice on the same interface within a time-frame of two hours (lines 6–8), the event handler will trigger \texttt{comp.example.Message6} to notify an administrator about a potential problem. If an exception occurred twice on the same interface within a time-frame of 30 minutes and the timestamp of the second event is not at office hours, the administrator will receive a message on his pager (lines 10–12). A different pager is used in a more urgent scenario when two different interfaces have problems within a time period of 30 minutes (lines 14–16). Finally a mail is sent to the administrator if two different interfaces signal the highest threshold event ($T2$) within 15 minutes (lines 18–20).

In this example we observe that the composite event type \texttt{comp.example} offers four different alternatives of pairwise event correlation, namely two sequential and two conjunctive correlations, to cope with different combinations of external triggers.

It should be noted here that the event handlers generated from such event definitions do not initiate any activity but rather notify an appropriate enforcement agent if a matching policy definition exists. This is performed both for composite and simple events. Therefore no composite event is required for the threshold \texttt{thr.example.$T1$} in Fig. 7 if a matching policy deals directly with the simple monitoring event.

### 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.

The Marvel-based management architecture and the concept of agents presented in the main body of the paper provide flexible means to automate management policies. This was achieved by formalizing policies in the form of precondition/activity/postcondition rules and event specifications. Rules effectively imple-
ment computer-supported chaining and execution of logically related management activities, including the invocation of prefabricated management tools. Generic monitoring agents that are capable to create events caused by resource monitoring and issue activation requests to the rule interpreter enhance the functionality and performance the rule-based architecture.

Experimentation with the Marvel-based management prototype approved the use of the general concept. But it also revealed some limitations due to Marvel's current design. Ill-designed management policies may have severe, often non-recoverable, impacts on a distributed system. Therefore, we found it desirable to inspect a rule chain before it is actually executed. But the Marvel interpreter offers no such feedback mechanism. Another problem with Marvel is the difficulty to find logical defects in a rule base. Especially indirect dependencies of conflicting policies turned out to be difficult to handle. One such case is policy rules that use and affect different target objects which are logically related through the use of a common resource, only. Moreover, the advantage of rules to be easy to change turns into a drawback when the impact of a planned change to a given policy model must be determined.

A recent extension of our management architecture includes a graph-based semantic model for policy and event definitions as a supplement to the Marvel-based implementation. We have presented this model and argued that it provides firm grounds for the implementation of mechanisms and operations balancing the deficits of the Marvel-based management architecture. Further experimentation with the extended management environment will show whether the redundant maintenance of a rule set and a graph model are practical, especially can be kept consistent, or whether we better re-implement the rule interpreter on the basis of the graph model.

The final introduction of an elaborate event handling and correlation mechanism pays tribute to the fact that event processing belongs to the most essential aspects of distributed systems management, which includes, to a large extent, event driven tasks. The described management architecture supports a recursive approach to event correlation in the presence of timing constraints. This allows for the formation of events that embody a richer semantic content than simple monitoring events and contribute to a significant reduction in the number of events signaled to the automated or human manager. We are currently exploring an Orbix based prototype implementation of the extended management loop in which the main focus is on monitoring and event handling, while activity chaining is simplified to forward chaining. Subsequent activities are linked by intermediate events, which obviates the need for a rule interpreter. Significant comparative results of the different prototype implementations of our management architecture require, however, further experimentation.

A more detailed description of this management approach can be found in [8].
Acknowledgments

The research reported in this article is partly sponsored by the European Union under the European-Australian cooperation project ISI (Information Systems Interoperability, project number ECAUS 003) and by the German Minister of Research and Technology under project number KAN INF 26, Distributed Systems Management.

References


A Incomplete PROGRES Specification of Policy Graphs and Pertinent Manipulation Operations

```plaintext
spec OperationalPolicySemantics

section GraphScheme
  node_class Generic
    intrinsic
      index Name : string;
    end;

e_type may_trigger : Trigger -> Precondition;
e_type forward : Postcondition -> Precondition;
e_type backward : Precondition -> Postcondition;
end;

section Policies
  node_class Policy is a Generic
    intrinsic
      Subject : string;
      Target : string;
    end;

  node_type OblPolicy : Policy end;

  node_class Postcondition
    intrinsic
      PostExpr : string;
    end;

  node_type Postcond : Postcondition end;

  edge_type cond : Policy [1:1] -> Precondition [0:1];
  (* Each Policy may have at most one Precondition. *)
  ...
end;

section Events

  node_class Trigger is a Generic
    intrinsic
      Op : string;
      Threshold : real;
      DynamicMode : boolean := false;
      Delay : integer;
    end;

  node_type SingleTr : Trigger end;
```
node_type RepeatedTr : Trigger end;

edge_type has : Event [1:1] -> Trigger [1:n];
(* Each Trigger belongs to exactly one Event, *)
(* but a PoEvent could have more than one Trigger. *)

edge_type hasFtr : PollingEv [1:1] -> Filter [0:1];
(* A PollingEvent could have at most one Filter. *)
end;

(* Operations on Graphs *)

section CreateOperations
production AddFilter ( evid : string ;
                      FType : type_in Filter ;
                      win : integer [0:1] ) =

    begin
        obl_node '1' : PoEv;
        not_node '2' : Filter;
        '1' -> '2' : hasFtr;
    end

    :=

    begin
        obl_node 1' = '1;
        2' : FType;
        1' -> 2' : hasFtr;
    end

    condition '1'.Name = evid;
    transfer 2'.Window := [ win | 3 ];
end;
...

end.