Generic Policy Management for Open Service Markets

M.T. Tu, F. Griffel, M. Merz, W. Lamersdorf
Distributed Systems Group, Computer Science Department, University of Hamburg
Vogt–Kölln–Str. 30, D–22527 Hamburg, Germany
[tu|griffel|merz|lamersd]@informatik.uni-hamburg.de

Abstract

The dynamic and evolutionary character of open electronic service markets with respect to both the application and the system infrastructure level requires appropriate system support mechanisms in order to dynamically configure business transactions and provide the required support services in flexible and individual ways.

This paper proposes the use of *policy management* mechanisms to support flexible cooperation of applications and dynamic configuration of support services in electronic service markets. It argues that various such policy management mechanisms can be efficiently supported *generically* by a respective common policy management service. Accordingly, the paper presents a generic approach to both policy *modeling* and policy *processing* aspects of an open system support platform for electronic service markets.

Keywords

Policy Management, Interoperability, Dynamic Configuration, Electronic Markets

1 INTRODUCTION

Open electronic service markets (EM) implemented on top of a heterogeneous medium such as the Internet have to be able to support a wide and constantly changing variety of on-line services offered by providers in many different business fields (Merz 1996). System support needed to perform electronic business transactions effectively as well as safely might range, for example, from authentication and notary to billing, payment, trading and negotiation services. Such "third party" services themselves can be offered by several providers with a variety of prices and qualities, using potentially different protocols, e.g. Ecash vs. NetBill. Thus, a system infrastructure,

which allows different and also not yet existing third party service providers to be able to participate (and compete) in the EM, cannot "hard-wire" such services into it, but rather has to integrate them flexibly as "plug-in" components into the overall EM architecture.

In this architecture, the set of support services for electronic market transactions is determined and integrated *dynamically* according to the requirements specified by customer and supplier - called *transaction parties* - just at the time they decide to carry out the transaction. Possible realizations of such a flexibly configurable EM architecture need not only *dynamic invocation mechanisms* to access arbitrary conforming support services at run-time, but also mechanisms to *specify* and *match* the requirements of respective transaction parties.

Moreover, the requirements of transaction parties are not only restricted to the specification of support services but can generally refer to any aspects of the impending cooperation. In general, before the actual exchange between customer and provider takes place, the overall setting of the transaction has to be determined and activated. This setting is defined both by the integration of external components like support services and by the specific *behavior* of the involved parties with respect to this transaction. Therefore, dynamic generic mechanisms are needed to specify the requirements and expected behavior of cooperating parties.

In several application fields and current standardization efforts, e.g. in the fields of *trading* (OMG 1996*b*, ISO 1995), *security* (ISO 1994*a*) and *performance management* (Meyer & Popien 1994), *policy management* is proposed as a mechanism to describe and influence the behavior of distributed applications, where a *policy* is a formal description specifying the requirements / expected behavior of an application. The notion of policy management for distributed systems has originally been used in the field of systems and network management in which the main concern is enforcing a common policy for a set of objects - called a *domain* - whereas the main focus of this contribution is on matching different policies of transaction parties to enable a cooperation.

The rest of the paper is organized as follows: In section 2, the problem of *modeling* policies in an application independent way is discussed and a formal specification format of policies is proposed. Section 3 focuses on *generic methods* to process policies, especially those to match and resolve conflicts between different policies. The architecture of the ongoing prototype implementation of a *generic policy management component* on a CORBA platform is described in section 4. The last section gives an overview of some open questions.

2 MODELING POLICIES FOR DISTRIBUTED APPLICATIONS

Policy management has already been proposed and used for a wide range of different applications separately in order to influence or control various aspects of an application. In order to do that more efficiently in advanced integrated open system environments, however, there is clearly a need to model policies in a *general*, ap-

plication *independent* way. This section first examines general modeling concepts associated with policies in distributed systems and their formalization.

2.1 Policy Composition

Although the need for applying policies has been identified in several standardization activities and related work, there seems to be no clear conception yet on what a policy really denotes and which components and attributes it consists of, as has been pointed out by, e.g., (Wies 1995). Often, a policy is seen as a *rule* constraining the behavior of an object, but neither the formal representation of such a rule nor that of the object behavior is specified (ISO 1994*b*). In one of the most comprehensive conceptual works on modeling policies, Moffett (Moffett 1994) identifies management policies as objects with the following attributes:

- Modality: motivation or authorization. *Motivation* means that the policy is intended to motivate actions to take place and *authorization* means that the policy gives or denies power for actions to take place
- Policy Subjects: Those who are in charge of achieving the policy goals
- Policy Target Objects: The set of objects at which the policy is directed
- Policy Goals: This attribute defines either *high-level goals* or *actions* (to achieve some goal)
- Policy Constraints: This attribute defines constraints on the applicability of the policy, e.g. the times of activation

On the whole, Moffett's policy structure (and similar ones from the systems and network management field) aims at managing a large number of network objects classified into different domains, whereas the main goal of our approach is modeling and matching the dynamic external behavior of application objects cooperating in an individual way to perform business transactions. Therefore, we propose the following policy structure:

A *policy* consists of a *goal*, a *role*, and an *action*.

- Goals: A *goal* can be a *constraint* or a logical combination of constraints by the operators AND, OR, NOT.
  - Constraints: A *constraint* either states that a *property* exists or expresses a relation on the value of a property. To reduce the complexity of the policy processing algorithms (discussed below) we assume that this is a canonical relation between the property value and a literal, e.g. total_cost < 90.
  - Properties: Properties are used to model the behavior of an object explicitly. A *property* is a name-value pair, where name is a string and value can be of any type and type safety is checked at run-time. The name of a property (i.e. its semantics) has to be standardized if it is interpreted by more than one role. For

example, if both customer and supplier can specify the `payment_protocol` property, they need to have a common understanding of the possible values.

- Roles: Roles are used to characterize the type of objects which want to enforce the policy. This information is used by meta-policies to resolve conflicts as described below. A *role* is represented by a string.
- Actions: An action can be any kind of procedure that can be activated by the policy management component to achieve the goal if necessary (i.e. if the goal has been evaluated to *false*). In case of a passive policy, the action field is empty. An *action* is represented by a unique identifier.

2.2 Restrictions

In order to enable an effective, generic processing of policy specifications as presented in the next section, we impose some restrictions on the formal representation of policies as presented above:

The first refers to the type of constraints in policy goals: Only relations between a property value and a literal are considered. If relations between values of different properties were also allowed, e.g. `total_cost` $<$ 1/2*`average_speed`, then policies could indeed have richer goals, but this would enhance the complexity of the processing algorithms considerably.

Secondly, the overall complexity can also be reduced by allowing only goals in Disjunctive Normal Form (DNF). This is no semantical restriction, because there are constructive methods to convert a formula of predicate logics into DNF. This representation also provides a conceptual advantage because if a goal is represented in DNF, then each conjunct can be seen as an *alternative* goal in comparison with other conjuncts. Moreover, it is useful to exclude such goals that do not express real alternatives, i.e. contain conjuncts that can be inferred from one another. Therefore, we use a special form of DNF to represent goals, which is defined as follows:

Definition: A formula F is in *lean DNF* iff F is in DNF and for all conjuncts ki, kj of F holds: $\neg(ki \rightarrow kj)$

Further, it is assumed that in general, different properties are semantically independent, i.e. the value of a property can't be inferred from the values of other ones.*

It should be noted that this policy structure does not model the set of policy objects or the policy domain (as introduced in (Moffett 1994)) explicitly. However, the set of objects a policy is defined for can be modeled by the *set of properties* which the respective objects have in common. For example, in order to classify mobile agents into different security domains, following property sets could be defined and appointed explicitly to the agents:

* Where such semantical relationships are relevant, an additional preprocessing should be applied, as proposed in section 3.6.

high_security:	low_security:	no_security:
encryption=RSA	encryption=DES	encryption=no_encryption
key_length=46	key_length=16	key_length=0

In this way, the security behavior of an agent could be adapted dynamically by appointing a different property set to it at run-time. On the other hand, the behavior of all agents sharing the same security property set can be modified at the same time by modifying any property in the set, e.g. the key_length in "high_security".

3 GENERIC PROCESSING OF POLICIES

If we want to commonly control and influence the behavior of many different distributed applications dynamically by means of formal policies as defined in the last section, we need functions to process policies in a way that is independent of the specific application being treated. Some of such generic functions are proposed and presented in this section.

3.1 Evaluation

First, a generic function is needed to tell whether an arbitrary policy is satisfied at some time, i.e. to evaluate the goal of that policy to *true* or *false*. This is done by evaluating the single constraints in the goal and subsequently combining the results according to the semantics of the logical operators AND, OR, NOT.*

The evaluation function should also recognize *inconsistent* goals, for example $(\text{cost} > 20) \wedge (\text{cost} \leq 10)$, which can never be achieved.

3.2 Unification

For a policy driven cooperation between two or more applications it is usually necessary to match the policies of all the involved parties. It is intuitive that the result of the matching process should satisfy each of the input policies. This process can be understood as a *negotiation* of different requirements. For example, this is described in the Security Architecture of the ISO Basic Reference Model (ISO 1994*c*) as follows:

> The provision of the security features during an instance of connection-oriented communication may require the negotiation of the security services that are required. The procedures required for negotiating mechanisms and parameters can either be carried out as a separate procedure or as an integral part of the normal connection establishment procedure. (p. 10)

*It should be noted that if the goal could be any formula of first-order predicate logics, then this problem would be undecidable. Therefore, we have to restrict the expressiveness of goals as described in 2.2.

Using the policy structure presented above, the semantics of such a "separate procedure" - called `unify` - can be formalized by the following definition:

Definition:

For all policy goals G1, G2, R:
R = unify(G1, G2) iff
$R \rightarrow G1$ and
$R \rightarrow G2$ and
$\neg(\exists R')$: $(R' \not\equiv R) \wedge (R' \rightarrow G1) \wedge (R' \rightarrow G2) \wedge (R \rightarrow R')$

That means, the unification of two policies returns the weakest policy that logically implies both inputs.

Examples:

Given
P1 = $((\text{cost} \leq 50) \wedge (\text{speed} \geq 5)) \vee ((\text{cost} \leq 100) \wedge (\text{speed} \geq 10))$
P2 = $((\text{cost} \leq 90) \wedge (\text{speed} \geq 20)) \vee (\text{speed} < 5)$
P3 = $((\text{cost} \leq 90) \wedge (\text{speed} \geq 20)) \vee ((\text{cost} \leq 60) \wedge (\text{speed} \geq 5))$
Then
unify(P1, P2) = $(\text{cost} \leq 90) \wedge (\text{speed} \geq 20)$
unify(P1, P3) = $((\text{cost} \leq 90) \wedge (\text{speed} \geq 20)) \vee ((\text{cost} \leq 50) \wedge (\text{speed} \geq 5))$

The `unify`-operation can be computed by the following algorithm:

- G1, G2 in lean DNF
- DG1 := Set of all disjuncts in G1
- DG2 := Set of all disjuncts in G2
- TMP := $\emptyset$
For each di in DG1
    For each dj in DG2
        If $di \rightarrow dj$ then TMP := TMP $\cup$ {di}
        If $dj \rightarrow di$ then TMP := TMP $\cup$ {dj}
DG1 := DG1 $\setminus$ TMP
DG2 := DG2 $\setminus$ TMP
For each di in DG1
    For each dj in DG2
        If $(di \wedge dj) \not\equiv$ FALSE then TMP := TMP $\cup$ {$di \wedge dj$}
unify(G1, G2) := Disjunction of all elements of TMP

3.3 Comparison

Besides unification as the main function to determine the common basis for a cooperation between two parties *directly*, comparison functions which are based on logical implication are necessary to decide if a certain policy is satisfied by another policy or by some "offer" formalized in terms of a policy. Especially, such comparison

functions can serve as a decision basis to realize automatic *negotiation* which can be considered a *stepwise* process of determining common cooperation parameters.

Using the policy structure presented in 2.2, the implication relation for two policies P1, P2 can be decided as follows:

(P1 $\rightarrow$ P2) iff
($\forall$p1|p1 is a disjunct in P1): ($\exists$p2|(p2 is a disjunct in P2) $\wedge$ (p1 $\rightarrow$ p2))

(p1 $\rightarrow$ p2) iff
($\forall$b|b is a conjunct in p2): ($\exists$a|(a is a conjunct in p1) $\wedge$ (a $\rightarrow$ b))

To decide if the implication relation holds for two atomic expressions, a decision matrix based on the mathematical properties of canonical relations such as '$<$' can be used.

Based on the decision method for logical implication, similar comparison functions as "equivalent" and "stronger" can be easily implemented:

(F1 equivalent F2) iff (F1 $\rightarrow$ F2) and (F2 $\rightarrow$ F1)

(F1 stronger F2) iff (F1 $\rightarrow$ F2) and $\neg$(F2 $\rightarrow$ F1)

Examples:

$\neg$((cost $>$ 50) $\vee$ (speed $<$ 5)) equivalent (cost $\leq$ 50) $\wedge$ (speed $\geq$ 5)

$\neg$((cost $>$ 50) $\vee$ (speed $<$ 5)) stronger (cost $\leq$ 100)

3.4 Arbitration

The `unify`-operation returns the value *false* if there is no policy goal that can logically satisfy both inputs. This case can simply happen when, for example, one party specifies that the value of the `payment_protocol` property is "Ecash", while the other requires that it be "NetBill". In such cases, some additional means are required to resolve the conflict if the cooperation should still come about. A suggestion that has often been made by standardization activities is using *arbitration* policies in such cases in order to influence the matching process. For example, in (ISO 1995) *arbitration for trader policies* is described as follows:

> An arbitration action template is a template for actions which combine a criteria argument (provided at an interface) with trader criteria and property values (available from the trader's own state). The action produces a resultant criteria which corresponds to the policy (in enterprise terms) for performing a given operation.
> The arbitration action represents some computational algorithm within the trader object. It corresponds to the enterprise specification's arbitration policy. (p. 38)

However, there is no description of the semantics of the arbitration action and especially how conflicting policies can be resolved. In the following, we describe a simple generic method of resolving policy conflicts and a formalization of arbitration policies.

Whenever a conflict between policy goals exists, one or both of the goals can be weakened by *overwriting* certain property values of one policy goal by the values of the same properties of the other goal. Which properties can be overwritten during

this process is determined by arbitration policies. An arbitration policy defines the set of properties on which a role has *priority*. For example,

A1: (role = importer) → (priority = {max_return_card})
≡ ¬(role = importer) ∨ (priority = {max_return_card})
A2: (role = trader) → (priority = {max_search_time})
≡ ¬(role = trader) ∨ (priority = {max_search_time})

means that the *importer* has priority on the property max_return_card and the *trader* has priority on the property max_search_time. Thus, arbitration policies, which are used as meta-policies to influence the matching process, are formalized in the same way as the policies being matched.

The overwriting of property values should be done *stepwise*, i.e. as few properties as possible should be overwritten to produce a non-empty resultant policy. A great advantage of this simple arbitration mechanism is its orthogonality, i.e. it works for arbitrary property value types. However, this kind of arbitration may be too coarse-granulated when dealing with numerical value types, for which a stepwise *tuning* of a property value might be much more appropriate than overwriting it. Therefore, additional fine-granulated arbitration mechanisms should also be implemented to deal with special value types.

3.5 Activation

In the last subsections, we have considered the processing of policies in a relatively passive sense, i.e. restricted to questions of evaluating and unifying policy goals. The concept of a policy also gains an active aspect if it is associated with some *action* that can be performed to achieve the goal. In order to provide *system support* for electronic business transactions, we are interested in *generic action types* that can be performed automatically.

Moreover, it is usually supposed that an action is carried out if and only if a triggering *event* has occurred and certain boundary *conditions* are satisfied. These concepts can be modeled in a simple way by the formal representation presented in 2 as follows:

Only a standardized property is needed to model events. When an event has occurred, this property is set to a corresponding value (e.g. by an *event monitor*). Similarly, a corresponding property is introduced for each boundary condition. Then, the relationship between events, boundary conditions and policy actions can be expressed within the policy goal, e.g. by the following expression:

G: (event = import_request) ∧ (trader_state = ready) →
(def_follow_link = always)

G is equivalent to

G': ¬(event = import_request) ∨ ¬(trader_state = ready) ∨
(def_follow_link = always)

which is a well-formed policy goal according to section 2. The action associated with this policy will only be activated if both (event = import_request) and

(trader_state = ready) are *true*, i.e. the triggering event has occurred and boundary conditions are satisfied. In order to cope with a potentially great number of policies arising in complex applications, policies should be organized in such a way that when an event has occurred, only those relevant to the corresponding event type are triggered. This can be done by grouping policies into different *policy sets*, each of which corresponds to some event type (as shown in the GPM architecture below).

If parameters of procedures called by an application can be directly modeled by corresponding properties, we can use a generic action type - called the *SET* operation - that directly sets properties to values that satisfy a policy goal. Using this operation, a relatively simple (re)configuration of applications at run-time can be implemented.

3.6 Matching Related Properties

In many practical application environments it often happens that properties are not isolated, but closely related to one another. Thus, specifying a certain value for a property may require specifying certain values for other properties. There are some general problems resulting from this relationship, which have been identified in the context of matching support services for business transactions in (Merz, Tu & Lamersdorf 1996).

This paper shows that a support service can be specified by the properties support service *class* (e.g. "payment"), support service *protocol* (e.g. "Ecash") and support service *provider* (e.g. "German Fed"). All three properties must be known to be able to bind a specific support service instance into the business transaction at run-time. However, in order to enable a flexible, natural specification of support service requirements, it should be allowed that customers and suppliers specify only the properties they consider relevant for their respective transaction or even none of them. For example, a customer who decides to *pay* by *Ecash* and is not interested in *which bank* is involved in the transaction, only needs to specify the first two properties. If this kind of flexibility is allowed, additional problems can arise for the *matching process*:

1. How to treat a unilateral requirement (one party requires a payment service while the other doesn't even specify such a requirement)?
2. How to resolve different levels of specification? For example, the client only requires *any* payment service, while the server specifies it at the provider level (e.g., "Bill's Bank")?

The solution of the first problem depends on how to interpret the fact that nothing at all has been specified. Logically, this can be considered as equivalent with specifying either the constant *true* or *false*. In the first case, the `unify`-operation defined in 3.2 would return as resultant goal the goal of the party who has specified some requirement, so this would mean that the non-specifying party implicitly accepts any requirement specified by the other. As for the other case, the `unify`-operation would

return *false* which means the non-specifying party implicitly denies any requirement made by the other party. The first alternative is obviously more *cooperative*, but which semantics is more meaningful could depend a lot on the type of application.

From the logical viewpoint, the answer to the second question is very simple: In this case, the `unify`-operation just returns the conjunction of the two requirements (SS-CLASS = "payment" ∧ SS-PROVIDER = "Bill's Bank") because this is exactly the weakest goal that can satisfy both inputs. But there is a more severe problem due to the fact that the requirements are *under-specified* so that no definite support service *instance* can be identified. A simple solution for this problem would be just choosing any arbitrary instance (that is offered on the service market) to support the transaction. However, since the number of instances matching an under-specified requirement is potentially great, it is likely that the chosen instance is inappropriate, i.e. does not provide support for the specific parties involved (for example, either customer or supplier doesn't have an account at the respective bank). Therefore, a better solution is using an automatic mechanism to *expand* an under-specified goal to a fully specified one before performing the matching process, which requires that the related property values are stored in some database and can be retrieved by the policy management component.

4 PROTOTYPE IMPLEMENTATION

The policy management mechanisms presented above are currently implemented as part of the "Policy Management" project at University of Hamburg. The main goal of this project is to develop a *generic policy management component* (GPM) that can be used to support (simultaneously) several different distributed applications to *specify* and *enforce* a specific behavior at run-time or to *negotiate* a specific behavior for a cooperation between several parties.

The GPM prototype is implemented as a CORBA service and can be considered a *common facility*. The CORBA framework (OMG 1996*c*) enables high-level, standardized object interoperability over multiple platforms. Especially, the CORBA DII (*Dynamic Invocation Interface*) enables the invocation of arbitrary methods specified in interfaces that are potentially unknown at the compilation time of an application what is an essential requirement to invoke dynamically introduced actions that can be performed to achieve policy goals.

The GPM consists of two main components: the *Property Manager* and the *Policy Manager* (see fig. 1). The Property Manager is responsible for *storing*, *updating* and *retrieving* properties of arbitrary applications and conforms to the *Property Service* specified in the CORBA standard (OMG 1996*a*). The Policy Manager is responsible for *storing*, *retrieving*, *processing* and *activating* policies. All properties and policies can be stored persistently in corresponding *repositories*, within which different *sets* can be created to group properties and policies according to different applications or users. Appropriate access control mechanisms are imposed on property and policy sets so that only authorized clients can insert, retrieve, modify or activate a property or policy.

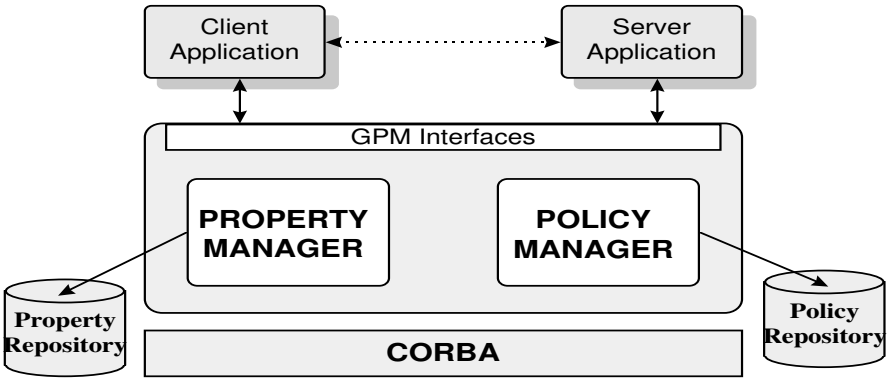

Figure 1 GPM architecture

The general procedure for a client to make use of the GPM is as follows: Properties representing the behavior of the application are to be registered with the Property Manager. Policies defined in terms of those properties are registered with the Policy Manager. At run-time, a policy of a client can be directly activated or can be unified with the policy of another client to create a matching policy that is activated for both clients. When a policy is *activated*, the Policy Manager first checks if the policy goal is satisfied by evaluating the constraints imposed on the properties whose values are retrieved from the Property Manager. If the goal is not already satisfied, then any actions specified by the policy will be performed. Then, the policy goal is evaluated again, and if it is still not satisfied, a *policy exception* will be raised.

A prototype implementation of the functionality presented in section 3 has been finished and the automatic activation of policies using OrbixTalk (CORBA Event Service implementation of IONA Technologies Ltd.) as the event monitor is currently under implementation. (See also http://vsys-www.informatik.uni-hamburg.de/dynamics)

5 SUMMARY AND OUTLOOK

This paper presented a generic approach to both modeling and processing aspects of a policy management component that can support a variety of applications in the context of open service markets, especially those that require matching different requirements of the involved parties. A formalization of policies based on a special form of predicate logics was proposed and generic policy processing functions, including evaluation, unification, comparison, arbitration and activation, and some algorithms to implement them were presented. Finally, the design and current implementation state of a generic policy management component were described. Several questions that are currently further examined include:

- Which kinds of policy-driven application need *more expressive* goals than those presented in section 2, and (if there are any) how to process them.
- How to *optimize* policies automatically by defining *meta-policies*, which either influence the processing of policies (like arbitration policies described in section 3.4) or directly modify existing policies.
- How to implement *evolving* applications by defining and performing *action types* that add new (or remove) functionality to an existing application at run-time.

Acknowledgements: This work has been supported, in part, by grant no. La1061/1-1 from the German Research Council (Deutsche Forschungsgemeinschaft, DFG).

The authors would also like to acknowledge the cooperation and technical support provided by the GMD *Research Institute for Open Communication Systems* (FOKUS) during the work leading to the results reported here.

REFERENCES

ISO (1994*a*), 'ISO/IEC DIS 10181: Information Technology - Open Systems Interconnection - Security Frameworks for Open Systems', ISO/IEC.

ISO (1994*b*), 'ISO/IEC DIS 10746', ISO/IEC. Information Technology - Open Systems Interconnection - Data Management and Open Distributed Processing - Basic Reference Model of Open Distributed Processing.

ISO (1994*c*), 'ISO/IEC IS 7498-2: Information Processing Systems - Open Systems Interconnection - Basic Reference Model, Part 2: Security Architecture', ISO/IEC.

ISO (1995), 'ISO/IEC DIS 13235: Information Technology - Open Systems Interconnection - Data Management and Open Distributed Processing - Draft ODP Trading Function', ISO/IEC.

Merz, M. (1996), Electronic Service Markets, PhD thesis, University of Hamburg, Dept. of Computer Science.

Merz, M., Tu, M. & Lamersdorf, W. (1996), Dynamic Support Service Selection for Business Transactions in Electronic Service Markets, *in* O. Spaniol, C. Linnhoff-Popien & B. Meyer, eds, 'Proc. Aachen Workshop "Trends in Distributed Systems"', Verlag der Augustinus Buchhandlung, Aachener Beiträge zur Informatik, pp. 183–195.

Meyer, B. & Popien, C. (1994), Defining policies for performance management in open distributed systems, *in* 'Proceedings of the 5th IFIP/IEEE International Workshop on Distributed Systems: Operation and Management', Toulouse.

Moffett, J. (1994), Specification of Management Policies and Discretionary Access Control, *in* M. Sloman, ed., 'Network and Distributed Systems Management', Addison-Wesley, pp. 455–480.

OMG (1996*a*), 'CORBAservices: Common Object Services Specification. OMG Document, updated July 15'.

OMG (1996*b*), 'OMG RFP5 Submission - Trading Object Service - OMG Document 96-05-06 Version 1.0.0'.

OMG (1996*c*), 'The Common Object Request Broker: Architecture and Specification. OMG Document 96-03-04 Revision 2.0'.

Wies, R. (1995), *Policies in Integrated Network and Systems Management*, Verlag Shaker, Aachen.