From Processes to Policies — Concepts for Large Scale Policy Generation

V. A. Danciu, B. Kempter
Munich Network Management Team
University of Munich
Oettingenstr. 67
D-80538 Munich
Germany
{danciu|kempter}@nm.ifi.lmu.de

Abstract

Process-oriented IT management has gained more and more significance due to the need to specify and control IT infrastructures and services. An indicator for this circumstance is the success of the IT Infrastructure Library and the enhanced Telecom Operations Map. A process-oriented view allows administrators to specify management tasks and to decompose complex tasks into process chains and hierarchies. Policy based management is a promising candidate for implementation of processes, as it delivers a powerful concept for the implementation of management tasks. Policies are well suited for large scale systems, as the distributed execution in heterogeneous environments is one of the important features of policy architectures.

In this paper we introduce patterns for the automated generation of policies from process specifications. To achieve the translation between process and policy specification, we leverage common characteristics of processes and policies and define mappings that serve as generic translation patterns. The use of these patterns allows automated translation of process definitions to policies. The approach enables high-level, intuitive management specification, while taking a step towards augmented automation and integration at the operational level. We derive requirements imposed by the translation and present ProPolis, a policy language incorporating those requirements.

Keywords

IT management process, policy-based management, translation patterns

1 Introduction

Process orientation is gaining more and more attention in the industry and research area. The need to implement processes throughout all parts of a company grows, as it promises synergy and automation effects through integration with existing business processes.

This paper focuses on IT management processes. In our view, a management process at the operational level consists of a set of management operations, executed on managed systems. The IT infrastructure library (ITIL) [18] and the enhanced Telecom Operations Map (eTOM) [11] are prominent representatives of standardization documents, aiming to achieve common specifications of management processes.

At present, the assessment of business practices and the derivation of process specification from the analysis results continually gain importance in the IT industry. Reference process specifications provide a common base for this effort. The resulting, organization-specific management processes are implemented using various technologies. Some emerging technologies allow the direct execution of processes/workflows, however they are constrained to specific services or specific types of managed objects.

On the other hand, policy based management is a concept accepted by the research community as a powerful means of specifying management down to implementation level.

The greatest benefit from deploying policy-based management is expected for organizations that maintain large networks and cater for large user bases. One of the major difficulties encountered, when such an organization moves to policy-based management, is the large number of policies that have to be specified 'from scratch'. In consequence, the initial cost of introducing policy-based management is quite high. Changes to the business processes have to be reflected as changes to the policy base, thus adding maintenance cost.

In this paper we systematically analyze common management process patterns and derive a generic mapping to policies. By applying such a mapping, policies can be generated automatically from process specifications. This approach allows a more intuitive management design through specification of management processes, and enables the use of graphical tools. The policies generated from the specification implement the operational management.

Using a generic approach, we avoid constraining the mapping to specific policy languages or process representations. By satisfying the specific requirements imposed by the process to policy translation, the implementation complexity of a translator software can be diminished. For this reason, a modular, process-aware policy definition language called ProPoliS has been designed. A 'machine-friendly', XML-based, prototypical implementation of this language serves as a platform for further study.

Section 2 presents the necessary concepts for a process to policy mapping. Section 3 is dedicated to the mapping of management processes to policies. From the common characteristics of processes (and policies) we develop *translation patterns*. By analyzing a process definition and identifying such patterns, a translation tool can generate policies from process partitions. An example process from the area of accounting management demonstrates the applicability of our concept. Section 4 introduces the policy definition language ProPoliS.

Publications of importance to our approach are discussed in section 5. Section 6 summarizes the paper and presents further work.

2 Fundamental Concepts

To specify a generic mapping between processes and policies, we identify their common characteristics by analyzing their elements. To achieve general, valid mapping rules called *translation patterns,* a general representation of management processes and policies has to be identified. For this purpose, prominent standards and research work have been analyzed as a prerequisite of Sections 2.1 and 2.2.

2.1 Common Process Characteristics

To summarize important work [11, 18, 22, 24], a process can be informally described as a set of entities, events and actions:

Entities	can be either human personnel or managed objects (MOs). Entities may generate events or execute actions. Examples for entities are: 'change manager', 'user' and 'application server'.
Events	communicate a change of state in an entity, that is relevant to management. Examples for events are: 'document changed', 'user requests service' and 'application server restarted'.
Actions	are performed by entities in response to events. They can perform a change of state in entities. A special form of action is the generation of an event. Examples for actions are: 'update document', 'authenticate user' and 'restart application server'.

Process chaining is implemented by having the source process generate an event that is significant to the target process. Thus, processes can be modeled independently of each other, having their interfaces to other processes defined by the expected and potentially generated events.

Introducing the concept of *rich events* allows an additional information flow between processes. Rich events (as e.g. in [20]) serve as a transport facility for any application-specific information.

2.2 Common Policy Characteristics

From the existing work on policies ([7, 21, 15, 9, 17]) a canonical form for policies can be derived.

Subject	designates the entity to which the policy applies.
Target	identifies an entity that is affected by the policy (i. e. the target of the action of the policy).
Event	describes the event which causes the policy to be evaluated.
Action	is the operation performed on a *target* to enforce the policy.
Condition	is the constraint under which the policy is enforced.

While only the event, action and condition fields are ubiquitous, the introduction of subjects and targets is necessary for referencing entities.

3 Mapping Management Processes to Policies

The characteristics of management processes are used as a common base for the targeted mapping. Figure 1 shows an overview of the mapping of characteristics. Entities are mapped to subjects and targets. Thus, persons or MOs can have policies specified for them (subject). They can also be affected by policies (target).

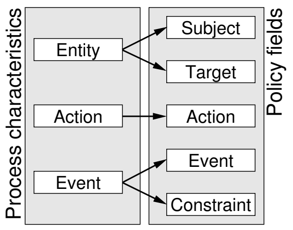


Figure 1 Mapping process to policy characteristics

Actions in process semantics correspond to actions of policies. An action can be anything between one single function call and entire scripts executed on a target MO. Events in processes match the event-constraint-pair in policies. While it is common for process specifications to contain conditions, these can be represented using a sufficiently detailed typing of events. Such a typing will yield a very large number of event types and is, in consequence, unsuited for implementation. Even if, in our view, conditions are not part of the indispensable process characteristics, we employ UML decision notation (e.g. in Figure 3) for clarity. We deem an event in process to have happened, when an event notification (e.g. a signal) has been received, and the conditions (if any) yield *true*.

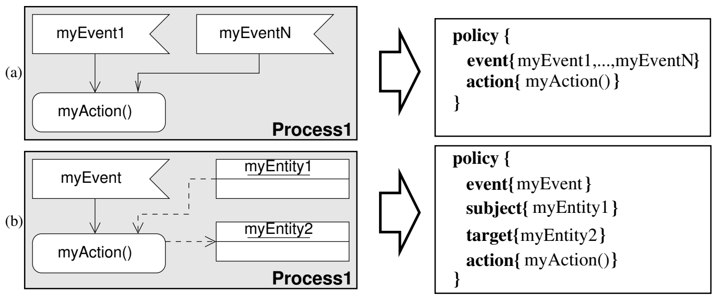
3.1 Translation primitives

To extract translation primitives, we describe the fundamental patterns occurring in processes and give the general notation of the policies generated from them. We use UML Activity Diagrams [24] for graphical description of processes, though, in general, any equivalent notation can be used.

Basic patterns

The most fundamental structure in an event-driven process is the unconditional execution of an action in response to an event. As shown in Figure 2a, several events can be specified to trigger the execution of the action (the comma in the event list implies *or*).

In most cases, the entity responsible for the action and the target of that action need to be specified as well (Figure 2b).



```
policy {
  event{ myEvent1,...,myEventN}
  action{ myAction() }
}
```

```
policy {
  event{ myEvent }
  subject{ myEntity1 }
  target{myEntity2 }
  action{ myAction() }
}
```

Figure 2 Basic patterns

The first case results in a single policy with only the *event* and *action* fields present. In the second case, *subject* and *target* of the policy are determined by examining the direction of the arrows that connect the entities to the action.

An entity having an arrow pointing *towards* it (UML output object) results in a policy target, while entities having arrows pointing *away* (input object) from them (and towards the action) result in policy subjects. The policies resulting from the patterns are shown in order in Figure 2.

Decision pattern

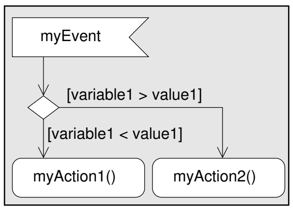
The process in Figure 3 chooses between two different actions depending on the value of a predicate. Two policies are generated from this process, differing at least in the condition field.

The general case has several decision branches, each carrying its guard condition. In this case, a number of policies equal to the number of branches needs to be generated in such a way that every policy includes the condition of the decision branch it corresponds to. Note that any condition can be formulated for a decision branch; for clarity, Figure 3 is constrained to an <`expression`> <`operator`> <`expression`> pattern.

Parallel actions and synchronization

Parallelization and synchronization in UML Activity Diagrams are modeled using *synch bars*. The generation of policies for parallel actions in a process is done by generating one policy for each action, analogously to the generation of policies for multiple decision branches.

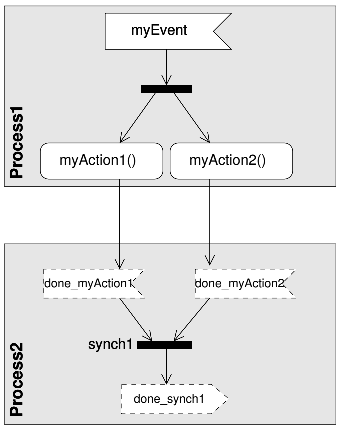
Synchronizing actions requires the policy system to track execution of actions and to



```
policy {
      event{ myEvent}
      action{ myAction1()}
      condition{  variable1 < value1}
}
policy {
      event{ myEvent}
      action{ myAction2()}
      condition{  variable1 > value1}
}
```

Figure 3 Decision pattern

generate events upon their termination. We call this type of internal events *intermediary events*. They need not be noted in the process specification but can be deducted from it.



```
policy {
   event{ myEvent}
   action{ myAction1()}
}
```

```
policy {
   event{ myEvent}
   action{ myAction2()}
}
```

```
policy { priority="99"
   event{ done_myAction1}
   target{SynchService}
   action{ setCompleted(synch1,
                       myAction1)
}
```

```
policy { priority="99"
   event{ done_myAction2}
   target{SynchService}
   action{ setCompleted(synch1,
           myAction2) }
}
```

```
policy { priority="0"
   event{ done_myAction1,
          done_myAction2 }
   target{System }
   action{generateEvent(done_synch1) }
   condition{
     SynchService.isCompleted(synch1)}
}
```

Figure 4 Synchronization pattern

Keeping state To implement synchronization, an instrument is required that can keep track of all the synchronized actions pending. That instrument will need to record the completed actions in the context of a synchronization point.

We design this instrument as a `SynchService`, that provides three functions to be used in policies. To distinguish concurrent instances of the process, a label called `synchID` is used. For each synchronization procedure, an instance of the service needs to be initialized by the policy system with an identifier of the synchronization point, the number of concurrent actions, and the names of these actions:

```
void init(String synchID, int numberOfActions, String[] actionNames)
```

Completion of an action must be recorded with the service using the following function:

```
void setCompleted(String synchID, String actionName)
```

The service can be used to determine whether there are pending actions, or if the synchronization procedure is complete:

```
boolean isCompleted(String synchID)
```

Execution order In response to an event pertaining to the synchronization point (`synch1`), two policies will fire: one that sets the state of the completed action, and one that checks if all actions have completed.

It is crucial that these policies are evaluated in order, i.e. the state of the completed action must be set *first*. With a random order of evaluation, the last action to complete may be recorded at the SynchService after the last check for completion has been performed; in consequence, the synchronization barrier will never be passed.

The ordering of policy evaluation is enforced using *priorities* as suggested in [9, 17], with a priority of 0 being the lowest. Execution ordering by means of priorities introduces the problem of priority assignment. For all synchronization instances, only two different priority values are needed – for policies setting the state and policies checking the state; these can be assigned automatically. Policies belonging to different instances will not interfere with each other, since they will call the SynchService using different `synchID` values. Policies governing the synchronization of the pattern are shown in Figure 4.

Fault handling

The execution of actions enforced by policies sometimes fails. Though most policy-based approaches do not seem to address this problem, handling failure of actions is an important part of a robust process design. While action failure detection is impossible to assert in general, implementations relying on a form of remote procedure call with synchronous procedure call semantics (e.g. based on a middleware like [6, 23]) are expected to provide it.

```
policy{
    event{myEvent1 }
    target{myTarget}
    action{default{myAction1()}
           onError{myAction2()}
 } }
```

Figure 5 Secondary action

Thus, it is useful to allow *secondary* actions to be specified within policies. They will be executed when the *primary* action (i.e. the action hitherto referred to) of the policy fails (Figure 5). This construct allows process patterns that need to test the success of an action to be translated similarly to the condition pattern sketched in Figure 3, but generating only one policy. The secondary action can be used to generate an event, transferring control to an error handling process [12].

Complexity estimates

The size of processes become very large at a detailed level. The number of policies generated from such a process specification is a co-determinant of the maintainability of the management system. Fortunately, the number of policies created from the patterns presented in this section can be estimated with some accuracy. In the following we will provide estimates for the number of policies generated from every pattern.

Conventions We define the following symbols used in the formulas:

N The number of policies generated; in general this is the result of the formulas.
B The number of branches in the condition pattern
B_a The number of branches ending in an action
B_t The number of branches *not* ending in an action
A The number of actions in the parallelization and synchronization patterns

Basic, multiple events, fault handling patterns In their isolated form, these pattern create only one policy.

$$N = 1 \tag{1}$$

Decision patterns The number of policies generated from a decision pattern is dependent on the number of available decision branches. In general, decision branches, that end in an action, yield one policy each. For decision branches not immediately leading to an action, the target of the branches must be analyzed.

$$B_a \leq N \leq B = B_a + B_t \tag{2}$$

Parallel actions Assuming a policy definition language that uses one single action per policy, the number of parallelized actions is determinant of the number of policies.

$$N = A \tag{3}$$

Synchronized actions Using the translation suggested, two policies will be generated for every synchronized action: one for registering its completion and one that checks if the completed action was the last one in the synchronization instance.

$$N = 2A \tag{4}$$

The sum of all the complexities of all patterns occurring in a process yields the complexity of the process as a whole. Taking our estimates into account, the complexity of the number of policies is $k \cdot n$, n being the number of patterns, and k a function yielding the typical number of policies created per pattern. Since k is a constant term, the order is $\mathcal{O}(n)$.

3.2 Mapping an Example Process

In order to illustrate the modeling of processes, we have chosen an example based on the accounting process presented in [21]. The example describes parts of the processes *Change*, *Deploy meters* and *Configure accounting system* that are triggered if a new user is being added to a service.

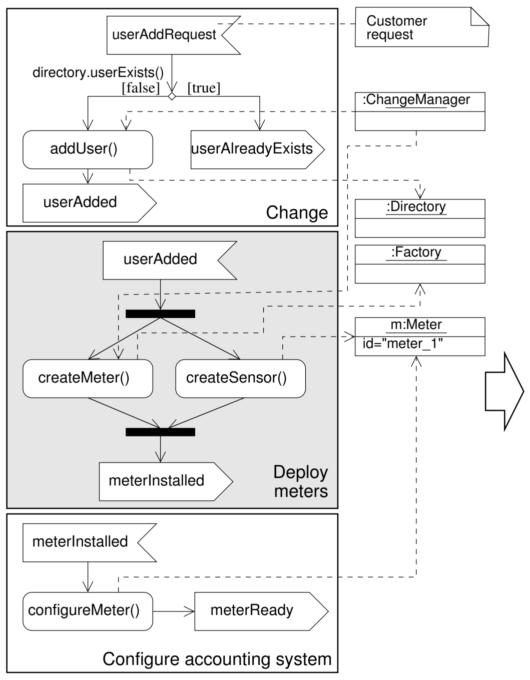
The change process is initiated by the event `userAddRequest`. In response to the event, the action `addUser()` is executed or the event `userAlreadyExists` is generated, depending whether the condition `directory.userExists()` is `false` or `true` (arguments to the `userExists()` function are transported by the event). If the user is added successfully, the event `userAdded` is generated.

In response to the event `userAdded`, features of the process *Deploy meters* are activated. The policies in Figure 6 originate from the *Deploy meters* process, that contains the parallelization/synchronization pattern. For every new user, a usage meter for the service that produces meter records (`createMeter()`) and a sensor (`createSensor()`) counting the accountable units is created.

As these actions can be executed independently of each other, two parallel control flows are initialized. Both actions are synchronized before the event `meterInstalled` is dispatched. This event, in turn, triggers features of the *Configure accounting system* process. Following the same pattern, the newly created meter is configured within this process.

3.3 Requirements for a Policy Definition Language

Based on the common characteristics of management processes and policies identified in Section 2.1 and 2.2, generic translation patterns were developed. The derived patterns and the complexity estimates for the translation indicate that process and policies fit each other quite well at the operational level. Our approach supports any policy language featuring the elements described in Section 2.2. Nevertheless, meeting a number of requirements imposed by the translation will help reduce complexity in the implementation of



```
policy {
    priority="99"
    event{ done_createMeter }
    target{SynchService}
    action{ setCompleted(synch1, createMeter())}
}
policy {
    priority="99"
    event{ done_createSensor}
    target{SynchService }
    action{setCompleted(synch1, createSensor())}
}
policy {
    priority="0"
    event{ done_createMeter }
    target{System}
    action{ generateEvent(done_synch1) }
    condition{ SynchService.isCompleted(synch1)}
}
policy {
    priority="0"
    event{ done_createSensor}
    target{System}
    action{ generateEvent(done_synch1) }
    condition{ SynchService.isCompleted(synch1)}
}

policy {
    event{ done_synch1 }
    target{System}
    action{ generateEvent(meterInstalled)}
}
```

Figure 6 Example process and mapping of deploy meters

a process-to-policy translator, enable maintenance and debugging features, and natively support robust process design.

Rich Events Events are the central mechanism for notifications and process chaining. Rich Events are an elegant and easy means to transport information necessary to process activity.

Execution Order Processes demand the possibility to preset an execution order of concurrently active policies.

Fault Handling For robust process design, it is necessary to be able to define actions that handle actions' failure.

Synchronization Parallelization and synchronization mechanisms for actions must be provided. This mechanism, in turn, requires the detection of action termination.

Partial Reverse Mapping Processes may change over time, entailing an update of the associated policies. Also, since faulty policies will always have been derived from an erroneous process specification, a reverse mapping of policies (back) to processes is required for debugging.

4 ProPoliS: A Process Aware Policy Definition Language

The automated generation of policies from process specifications poses new requirements on a policy language (Section 3.3), while the inherently automated management of the policies themselves allows the introduction of new language features.

4.1 Language Features

Tagging Mapping policies back to their origin process can be supported by *tagging* every policy with the identifier (e.g. name) of the process it is derived from. Policies triggering transitions between processes are tagged with the names of the source and target of the transition.

Modular policies In PCIM [5], an object oriented model for policy representation is proposed. In the context of management processes, the modularization of policies leverages already defined information by allowing reuse of e.g. *configuration items* (CI) defined in an ITIL Configuration Management process. Consistency between entities and roles can be ensured by providing a comprehensive set of entity definitions defined within an organization during process modeling. This also ensures consistency between the entities and roles used in policies.

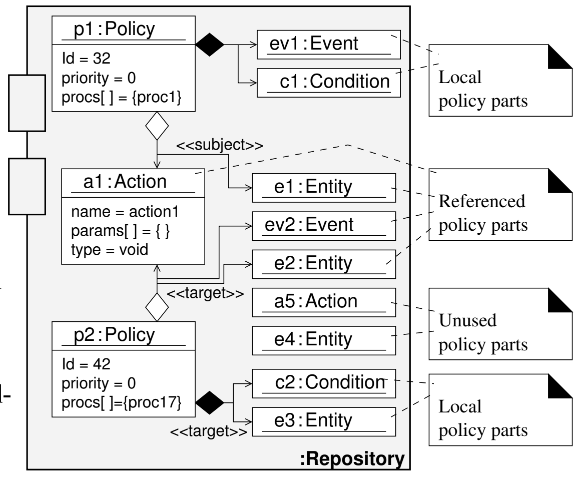


Figure 7 Repository for policies and policy parts

Therefore, we apply PCIM's modularization concepts in decomposing policies into parts, thus allowing the fields of the policy – the *policy parts* – to be locally defined or references to a global part definition. Note that local parts are only usable by the policy within which they are defined. Policies as well as policy parts are stored in a policy repository, as shown in Figure 7.

Roles and Domains Roles and domains are well known and pervasively implemented concepts [7]. ProPoliS allows the substitution of domains and/or roles for entities, i.e. instead of an entity, a role or domain can be specified.

Management of policies using meta-policies Automated maintenance of the policy base can be performed using meta-policies. We define meta-policies as operational policies operating on policies or policy parts.

4.2 ProPoliS Syntax Overview

ProPoliS was designed with an XML mapping of the language in mind. To meet this requirement, language elements are *well formed* in the sense of [26]; we use braces as delimiters. Some language elements are designed as attributes, allowing lists as the most complex data structure for their values. We use value types defined in [27] to further ease the mapping. An overview of the syntax specification in EBNF is given in the following. Due to space constraints, some of the grammar productions are left out.

References Globally defined policy parts carry an `id` to allow references to them from within policy definitions. Additionally, parts can be marked as ’library’ items; implementations should not delete unreferenced parts with this flag set. The `relatedProcess` construct allows tagging policies and parts with process identifiers.

```
⟨processAssignable⟩ ::=  ⟨referable⟩ [ ⟨relatedProcess⟩ ]
⟨referable⟩ ::=  ”id=” ⟨xsd:ID⟩ [ ⟨libraryItem⟩ ] [ ⟨comment⟩ ]
⟨relatedProcess⟩ ::=  ”relatedProcesses{” ⟨process⟩ { ”,” ⟨process⟩ } ”}”
⟨process⟩ ::=   ⟨identifier⟩ | ”OTHER” | ”ALL”
```

Policy A policy consists of the main fields (event, action etc.), attributes and administrative data. The policy descriptor and attributes determine its state (active or not), its priority, expiration date and authors. The language version can be noted for a policy, as ProPoliS may evolve to meet new requirements.

```
⟨policy⟩ ::=      ”policy{” ⟨processAssignable⟩ [ ”domain{” ⟨domain⟩ ”}” ]
                  [ ⟨subjectSet⟩ ] [ ⟨targetSet⟩ ] ⟨eventSet⟩ ⟨actionSet⟩
                  [ ⟨constraintSet⟩ ] [ ⟨descriptor⟩ ] ⟨attribs⟩ ”}”
⟨descriptor⟩ ::=  ”descriptor{” [ ⟨createdBy⟩ ] [ ⟨creationDate⟩ ]
                  [ ⟨lastModified⟩ ] [ ⟨modifiedBy⟩ ] [ ⟨expires⟩ ] ”}”
⟨attribs⟩ ::=     ”attributes{” ⟨enabled⟩ [ ⟨priority⟩ ] [ ⟨version⟩ ] ”}”
```

Main policy fields The main policy parts are organized in typed sets. The definitions of all policy parts follow the same pattern of allowing either a local definition, a single reference to a globally defined part, or a group of references. The meta-syntax is:

```
⟨part⟩ ::=        ”name{” ⟨local def.⟩ | ⟨ref⟩ | ⟨group⟩ ”}”
```

As examples, the `targetSet` and `constraintSet` productions are shown below. Constraint sets are in normal form and consist of unary and binary predicates applicable to values including literals, object attribute values and function return values.

```
⟨targetSet⟩ ::=  ”targetSet{” ( ⟨target⟩ | ⟨targetRef⟩ | ⟨targetGroup⟩ ) ”}”
⟨constraintSet⟩ ::=  ”constraintSet{” ( ⟨cnf⟩ | ⟨dnf⟩
                  | ⟨constraint⟩ | ⟨constraintRef⟩ ) ”}”
```

Actions are composite; they consist of a default action, and one that is to be executed if the first one fails. We use a notation common in object oriented languages, where an action consists of a symbol denoting a method name and a set of parameters. An object name is optional. We specify a special action for the generation of events, thus forcing implementation of event generation in the policy system.

```
⟨action⟩ ::=      ”action{” ⟨processAssignable⟩
                  ⟨defaultAction⟩ [ ⟨errorAction⟩ ] ”}”
⟨actionRef⟩ ::=  ⟨ref⟩
⟨actionGroup⟩ ::=  ⟨group⟩
```

```
⟨defaultAction⟩ ::=  ”default{” ⟨genericAction⟩ ”}”
⟨errorAction⟩ ::=  ”onError{” ⟨genericAction⟩ ”}”
⟨genericAction⟩ ::=  ⟨methodCall⟩ | ⟨generateEvent⟩
⟨methodCall⟩ ::=  ”invoke{” [ ⟨objectName⟩ ”.” ] ⟨method⟩ ”}”
⟨generateEvent⟩ ::=  ”generateEvent{” ⟨eventName⟩ ”(” ⟨parameterSet⟩ ”)}”
⟨method⟩ ::=    ⟨identifier⟩ ”(” ⟨parameterSet⟩ ”)”
⟨eventName⟩ ::=  ⟨identifier⟩
⟨objectName⟩ ::=  ⟨identifier⟩
```

The overview given does not constitute a complete specification, but is focused on constructs relevant to process translation. Language constructs not shown here include role and domain definitions, parameters and values, as well as constructs for logical expressions to be used in conditions.

5 Related Work

To our knowledge, the mapping of management processes to policies has not been addressed up till now. For this reason, we present prominent representative work in the area of management processes and policies, and discuss its significance in the focus of this paper.

5.1 Processes

Reference Processes The IT Infrastructure Library (ITIL) [18] is a continuously evolving collection of best practice documents, with regard to the service management of a IT service provider. It defines (among others) the process sets of *service support* and *service delivery*. Processes descriptions are derived from expert knowledge in a particular field and written more or less in prose, so that ITIL can be regarded to be a bottom-up approach. To use ITIL as a starting point for our approach, a few prerequisites must be met: as ITIL offers no formal process language, a direct mapping is not possible. The description of process chaining is often missing. Yet, an example for the formalization of the ITIL incident management process provides proof of concept that ITIL is a suitable and important source for formalizing management processes[4]. The event-driven chaining of processes is described using the event process chains (EPCs) described in ARIS [22]. EPC is a graphical modeling language and has equivalent expressiveness as UML activity diagrams.

A common process model for accounting management is presented in [21]. The dependencies of the subprocesses are described in a formal way using UML activity diagrams [19]. A mapping from the accounting management processes to policies is demonstrated, though a systematic approach for the mapping is not described.

The enhanced Telecom Operations Map (eTOM) [11] published by the TeleManagement Forum in 2002 is a business process framework for the telecommunication industry. A former, yet more formal representation of the processes, also describes the concrete association of processes[10].

eTOM is customer-centric and covers a broad range of important processes including processes for strategy, infrastructure, product, and operations. It has a hierarchical decomposition approach ranging from Level 0 (the highest conceptual level) down to Level 4. Level 0 covers seven process groups which are decomposed in Level 2 to over 80 (sub-) processes. Level 3 and Level 4 processes are the most detailed processes and include a process flow description (only at Level 2 all processes are described in eTOM). The processes are not described in a formal way, and neither input and output parameters, nor the linking of processes are described explicitely. eTOM serves as an important framework to define management processes. Using eTOM as a starting point of process to policy translation, first a formalization of the processes has to be accomplished.

Workflows and Business Processes Several process definition and formalization frameworks have been authored by different institutions in the field. Prominent examples are the Workflow Coalition's (WfMC) XML-based process definition language [25], as well as the Business Process Modeling Language [2] defined by the Business Process Management Initialive (BPMI) and the ebXML Business Process Specification Schema [3].

These specifications may converge (functionally) into the Business Process Execution Language for Web Services (BPEL4WS) [1], a specification with strong support in the industry, aiming at actual execution of business processes using web service technology. For process to policy translation, BPEL provides useful formalisms for the description of actions, as well as some support for entity definition and asynchronous messaging. BPEL's focus on web services constrains its general applicability; nevertheless, the underlying process specification facilities appear useful.

5.2 Policies

Policy based management is a field of intensive research activities since several years. [16, 13] introduce a policy hierarchy featuring different abstraction levels to which a policy can belong to. [13] defines three levels, namely strategic, goal oriented and operational policies. As we focus on operational processes, operational policies are discussed.

No standard for policy languages has evolved yet, but as a common understanding, a policy can be defined [8, 16] as: *A policy is a persistent and declarative specification of a rule regarding the behavior of a system. A policy is derived from management goals.*

[7, 8] introduces Ponder which is one of the furthest developed policy definition language (PDL) in the research community. Significant for Ponder is the introduction of different types of policies: authorization and obligation policies. Authorization policies express certain rights of a subject to execute an action (which exists in the form of admission and prohibition). On the other hand, an obligation policy specifies the duty to execute an action if a certain event occurs. Ponder covers constructs for roles, groups and domains to handle a set of objects at once. Although Ponder is designed to express any management task, a focus towards access control is established. If Ponder is chosen as target PDL for process translation, only obligation policies are generated. As Ponder does not support tagging directly, this feature could be introduced by coding the name of the process as part of the policy name.

The PDL developed in this paper is a enhancement of the PDL outlined in [21]. Important add-ons are tagging and modularity.

[15] defines a policy as a function that maps a series of events to a set of actions. Their policy definition language called $\mathcal{PDL}$ has the structure of an event-condition-action (ECA) rule. Two basic constructs exist: 'policy rule' and 'policy defined event proposition'. As an important difference to Ponder, $\mathcal{PDL}$ does not support the concept of subjects, targets, or grouping structures. [14] introduces a workflow language for $\mathcal{PDL}$. It defines workflows as a *specification of complex actions* of a policy action. Thus, its understanding of workflows is an extention of the policy language.

6 Conclusions and Further Work

Process orientation is one of the important trends in IT management. While processes are an established concept in core business areas, in the area of IT management process orientation is presently being widely deployed. On the other hand, policy based management is a promising candidate for the execution of management operations on large-scaled managed systems. A mapping of processes to policies is of great interest as it allows an important automation step.

In this paper, we have systematically analyzed the characteristics process of management processes in a top-down approach. To achieve a broad applicability, no specific process or policy description was presumed. In order to map processes to policies, generic translation patterns were extracted, the fundamental ones being decision, parallelism and synchronization patterns. Moreover, requirements like reverse mapping and fault handling for a suitable policy definition language were derived from the analysis phase. A complexity estimate indicates only a linear order of generated policies, demonstrating the applicability of our approach for large scale systems. A policy definition language called ProPoliS was specified to reflect the requirements of process orientation.

Our approach can be instrumental to a rapid implementation of translation tools for a given concrete process specification and policy definition language. With such a tool, only the process definition is required; most operational parts of the management system are created automatically.

In the future, we will apply the concepts presented in this paper to existing formalisms for process/workflow definition, and implement translation tools with ProPoliS as the policy language. Another area of interest is the reverse mapping, generating process descriptions out of an existing policy base.

Acknowledgment

The authors wish to thank the members of the Munich Network Management (MNM) Team for helpful discussions and valuable comments on previous versions of this paper. The MNM Team directed by Prof. Dr. Heinz-Gerd Hegering is a group of researchers of the University of Munich, the Munich University of Technology, and the Leibniz Supercomputing Center of the Bavarian Academy of Sciences. Its web–server is located at http://wwwmnmteam.ifi.lmu.de.

References

[1] Tony Andrews et al. Business Process Execution Lanugage for Web Services Version 1.1. Specification, November 2003.

[2] Business Process Modeling Lanugage. Specification, Business Process Management Initiative (BPMI), November 2002.

[3] ebXML Business Process Specification Schema Version 1.01. Specification, UN/CEFACT and OASIS, May 2001.

[4] M. Brenner, I. Radisic, and M. Schollmeyer. A Criteria Catalog based Methodology for Analyzing Service Management Processes. In M. Feridun, P. Kropf, and G. Babin, editors, *Proceedings of the 13th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM 2002)*, Lecture Notes in Computer Science (LNCS) 2506, pages 145–156, Montreal, Canada, October 2002. IFIP/IEEE, Springer.

[5] Common Information Model (CIM) Specification Version 2.2. Specification, June 1999.

[6] Common Object Request Broker Architecture (CORBA/IIOP). Omg specification, Object Management Group, December 2002.

[7] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. Ponder: A language for Specifying Security and Management Policies for Distributed Systems. The Language Specification Version 2.3. Imperial college research report doc 2000/1, Imperial College of Science, Technology and Medicine, University of London, Department of Computing, October 2000.

[8] N. C. Damianou. *A Policy Framework for Management of Distributed Systems*. PhD thesis, Imperial College of Science, Technology and Medicine, Universi ty of London, Department of Computing, February 2002.

[9] DMTF Policy working group. CIM Core Policy Model White Paper. White Paper, Distributed Management Task Force, June 2003.

[10] Telecom Operations Map. Technical Report GB 910 Approved Version 2.1, TeleManagement Forum, March 2000.

[11] enhanced Telecom Operations Map (eTOM), The Business Process Framework For The Information and Communications Services Industry. Technical Report GB 921 Approved Version 3.6, TeleManagement Forum, November 2003.

[12] H.-G. Hegering, S. Abeck, and B. Neumair. *Integrated Management of Networked Systems – Concepts, Architectures and their Operational Application*. Morgan Kaufmann Publishers, ISBN 1-55860-571-1, 1999.

[13] Thomas Koch. *Automated Management of Distributed Systems*. PhD thesis, Fern-Universität Hagen, Germany, 1997.

[14] Madhur Kohli and Jorge Lobo. Policy based management of telecommunication networks. In *Policy Workshop 1999*, Bristol, U.K., 1999.

[15] Jorge Lobo, Randeep Bhatia, and Shamim A. Naqvi. A policy description language. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, pages 291–298, 1999.

[16] Damian A. Marriott. *Policy Service for Distributed Systems*. PhD thesis, Imperial College London, June 1997.

[17] B. Moore, E. Ellesson, J. Strassner, and A. Westerinen. RFC 3060: Policy core information model – version 1 specification. RFC, Internet Engineering Task Force (IETF), February 2001.

[18] Office of Government Commerce (OGC), editor. *Service Support*. IT Infrastructure Library (ITIL). The Stationary Office, Norwich, UK, 2000.

[19] UML 1.4 Interchange Metamodel in XML. TC Document ad/01-02-15, Object Management Group, February 2001.

[20] Notification Service Specification. Technical Report formal/02-08-04, Object Management Group, August 2002.

[21] I. Radisic. Using Policy–Based Concepts to Provide Service Oriented Accounting Management. In R. Stadler and M. Ulema, editors, *Proceedings of the 8th International IFIP/IEEE Network Operations and Management Symposium (NOMS 2002)*, pages 313–326, Florence, Italy, April 2002. IFIP/IEEE, IEEE Publishing.

[22] A.-W. Scheer. *ARIS – Business Process Modeling*. Springer, Berlin, 1999.

[23] Sun. Java remote method invocation specification. Technical report, Sun Microsystems, 1997. http://www.javasoft.com/products/jdk/1.1/docs/guide/rmi/spec/ rmiTOC.doc.html.

[24] OMG Unified Modeling Language Specification, Version 1.5. Technical Report formal/03-03-01, Object Management Group, March 2003. http://www.omg.org/cgi-bin/doc?formal/03-03-01.

[25] Workflow Process Definition Interface – XML Process Definition Language Version 1.0. Specification, Workflow Management Coalition, October 2002.

[26] Extensible Markup Language (XML) 1.0. W3c recommendation, W3C, 2000.

[27] XML Schema Part 2: Datatypes. W3C Recommendation REC-xmlschema-2-20010502, May 2001.

Biography

Bernhard Kempter studied computer science at the Munich University of Technology, Germany (TUM). He joined the MNM team after receiving his diploma degree (M. Sc.) in 1999 and is currently a Ph. D. candidate at LMU. There he also works as a research and teaching assistant. His research interests focus on integrated networks and management in general, with an emphasis on service management and the resolution of policy conflicts.

Vitalian A. Danciu studied computer science at the Ludwig-Maximilian-University in Munich, Germany (LMU), where he received his diploma degree (M. Sc.) . Since joining the MNM team in 2003, he is a Ph. D. candidate at LMU while working as a research and teaching assistant. His research interests include policy-based, process-oriented management, as well as management of mobile systems.