Taming Policy Complexity: Model to Execution

Sven van der Meer, John Keeney, and Liam Fallon
Network Management Lab, Ericsson, Athlone, Co. Westmeath, Ireland
{sven.van.der.meer, john.keeney, liam.fallon}@ericsson.com

***Abstract*—Since the 1970's it has been acknowledged that a complex system can be broken into (a) its invariant functional parts (mechanism), and (b) the externalized choices for how the system should behave (policy). Policy-based management's main objective is to separate and externalize the decisions required by a system from the mechanisms provided by the system, and provide a way to define and evaluate these decisions. A few decades later, we have today a plethora of different policy models and even more policy languages – plus tooling – offering policy-based solutions for virtually any use case and scenario. However, policy-based management as a standalone domain has never been evaluated in terms of which parts are variant / invariant, i.e. which parts of policy-based management can be domain-, model-, language-, usecase-independent. In this paper, we introduce and define a formal universal policy model that does exactly that. The result is a model that can be used to design, implement, and deploy immutable policy infrastructure (engine and executor) being able to execute (virtually) any policy model.**

I. Introduction

In the past few years, the concept of policy-based management has taken center stage – again – in the communication industry. Policy is core to activities such as the TM Forum SID [1] and Zoom [2], IETF's SUPA [3] and to an extent ANIMA [4], AT&T's D2 [5] and ECOMP [6] open sourced in ONAP [7], ETSI's MANO [8] open sourced in OSM [9], and Ericsson's control architecture COMPA [10].

For any given scenario the first choice is the most critical: select a policy model[1] and soon after a policy language[2] from the ever growing list of candidates. These choices are often made based on limited information or experience, even though they are critical for the success of any activity, be it a standard, an open source project, a company's product portfolio, or a single product. When a wrong choice becomes obvious, usually much later, this can (usually will) lead to massively increased efforts and cost for an organization. Changing a policy model and/or language can be very difficult (or impossible), especially when policy is a system's core component.

A. This Work

In a manner similar to how any domain is examined, when we examine different policy approaches and systems it becomes clear that there are aspects of them that are similar and invariant, and parts that are different, and varying (variant). The principle of "separating policy from mechanism" [11] applies the term *mechanism* to the invariant parts and *policy* to the variant parts. When we maximize the invariant parts we can then build immutable infrastructure that is independent of the policy approach, and define a Universal Policy Model (UPM) to operate on this infrastructure and to describe and interchange between different policy- and application-domains.

Using Domain Driven Development (DDD) concepts [12][13], we have defined a UPM, detailed in Fig.1. The Reference Model for Policies (RM-Pol) captures all *policy mechanisms*. This model is presented as a base formal Domain Model (DM) for the policy domain in §II. The Application Domain Model (ADM) and Policy Domain Model (PDM) describe the invariant parts. The variant parts are then concrete Policy Models (PMs) and associated languages expressing them. In other words, the PDM defines all aspects to build immutable policy infrastructure while the PMs define how concrete, domain-specific policies should be specified. §IV details one example PM: an action policy (e.g. [14]).

The UPM can then be translated to execute on the common immutable infrastructure, i.e. policy executors and engines. This translation uses defined templates detailed in §III. The resulting Universal Policy Execution Environment (UPEE) is described in §V, with discussion of a candidate concrete implementation of the UPM and UPEE called APEX [15].

As a result, a decision for a particular policy model and language is no longer mission critical. With immutable infrastructure (policy engine and executor) and basic tooling (policy authoring), policy models and languages can be changed any time, or coexist solving different business problems. Only policy definitions change, not components.

B. Related Work

Understanding the role of policies and the policy-based approach requires some historical context. Policy was first used in the 1970's for security [16] where a relation W provides access control *rules* to govern system security. These rules help to govern state transitions on receiving requests. In [17], the authors state that a system specification describes *what* a system does while a policy describes *how*. Here, the dynamic features are *policy*, *role*, and *control*.

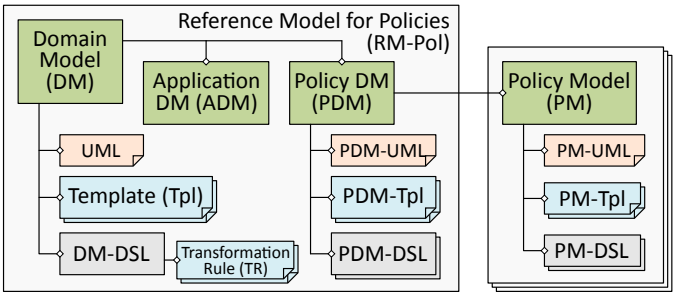


Fig. 1: Universal Policy Model (UPM)

[1] e.g. obligation, adaptive, promise, goal, utility, authorization, intent, refrain
[2] e.g. Ponder, N3, Turtle, KAoS, Rei, Drools, XACML, SAML, WS-Policy

978-1-5386-3416-5/18/$31.00

In the mid 1970's operating systems began using the term policy as an artifact of control [11], where users could influence kernel-space decisions without requiring an expensive kernel to user space switch. In [18] the authors separate static and dynamic policies for memory allocation, for *static* memory allocated at scheduling time with *dynamic* memory changing with the process. The policy/mechanism principle for operating system resource allocation is introduced in [11], while [19] explains how this principle can be applied to networks and their management.

Policies are first used in communication systems in [20] to control sharing resources. In [21] rules are used to detect and later prevent network congestion. Event Condition Action (ECA) rules appear first in active databases [22]. A defined event triggers the evaluation of a defined set of queries (condition) and a defined action is executed if the condition is satisfied. The processing of rules is strongly associated to database transactions. Event types for ECA are defined as database operation, temporal, and external notification.

Policy as a paradigm for network management is originally defined in [23], based on earlier work [24][25][26]. Originally focusing on access control, the work introduced domains, subject, and target, plus policy categories (and models) and a policy system (with language and tooling) called Ponder. This is then followed by policy standards, such as the IETF policy framework, DMTF CIM, TMF SID, to name a just view. A detailed historic perspective on policy can be found in [27].

A deep understanding of policy requires study of policy frameworks [28], approaches [29], and concepts for specifications [30]. Policies from different domains were also analyzed, such as cognitive radio [31], security [32], network traffic and QoS [33], and XML and open environments [34].

Approaches for models that allow multiple PMs in a single PDM have been developed in [35] and more recently in [3]. Here, each PM is bound to its specific definitions, for instance an ECA policy is bound to its inherent rule structure, so while being extensible, it is not easy to add new policy models.

A better approach is to specify a formal taxonomy that informs a formal PDM, independent of any specific PM, for instance in [36]. We apply the underlying concepts of this approach but strive for less variance in the PDM.

Tool support for syntactic and semantic translation between models can be achieved [37]. However, it is important to note that semantic translation cannot be fully automated. In [38] we study inter-domain relationships and policy translation, both important aspects for UPM. We have summarized today's challenges for policy-based management in [39] and [40].

II. Reference Model for Policies

The DM $\mathfrak{D}$ defines required concepts (their semantics and relationships) and sub-models. Sub-models define specific responsibilities and decomposition. The main two sub-models are the ADM $\mathfrak{D}^a$ (applications A) and PDM $\mathfrak{D}^p$ (policies P). Other sub-models can be added if required, e.g. for a naming architecture $\mathfrak{D}^{ns}$, time $\mathfrak{D}^{\check{t}}$, or a location model $\mathfrak{D}^{\check{l}}$. Table I shows that the domain models ($\mathfrak{D}^a$, $\mathfrak{D}^p$) can be specialized in models ($\mathfrak{M}^a$, $\mathfrak{M}^p$), which can then be further specialized in concrete models ($\mathfrak{C}^a$, $\mathfrak{C}^p$). For instance, an application model $\mathfrak{M}^a_{sdn}$ can define sets of applications for orchestration as A^{odl} and A^{onos} and concrete models for their implementation and deployment. Similar, we can use the PDM for policy models P^{eca} and P^{goal}.

Domain Model	$\mathfrak{D} = \{concept_{1..k}, \mathfrak{D}^a, \mathfrak{D}^p, \mathfrak{D}_{1..n}\}$
Application DM	$\mathfrak{D}^a = \{A, concept^a_{1..k}\} \models \{\mathfrak{M}^a_{1..m}\} \models \{\mathfrak{C}^a_{1..n}\}$
Policy DM	$\mathfrak{D}^p = \{P, concept^p_{1..k}\} \models \{\mathfrak{M}^p_{1..m}\} \models \{\mathfrak{C}^p_{1..n}\}$

TABLE I: RM-Pol

A. Domain Model Concepts

The DM defines fundamental concepts shared among all sub-models, shown in Table II. A set of names N contains names $n_{1..n}$ as unique identifiers of concepts and their instances. Name structure and semantics in a sub-model $\mathfrak{D}^{ns}$, for example the architecture in [41]. Most concepts introduced below have a name n and a natural language description d.

A timestamp $\check{t}$ defines a point in time, further defined in a sub-model $\mathfrak{D}^{\check{t}}$. Some concepts need to be typed as a type $\pi \in \Pi$, usually defined in a type model $\mathfrak{D}^{\pi}$.

A relationship ϕ formalizes how two concepts z_1 and z_2 relate to each other. For instance, an application a is executed in a processing system ψ^p. Further semantics might be defined in a relationship model $\mathfrak{D}^r$. A multi-relation ϕ^{ϕ} allows for n-dimensional relationships, e.g. multi-layer network topologies.

Executable and declarative expressions are modeled with the concepts *Language*, *Statement*, and *Strategy*. A language $\hat{l}$ maps to an (external) execution environment for statements. A statement λ is a mapping from a language $\hat{l}$ to a set of expressions. Finally, a strategy ξ defines a mapping of a type π to a statement λ. They allow different implementations of the same process or algorithm (cf. pp 315 in [42]).

Some concepts require typed values, e.g. a policy parameter `delay` as type `int` with integer values. These types need to be declared (introduced), defined (run-time association), and can finally be used (with values). A type declaration y binds a `key` (usually a string) to a `type`, which is in a given type

Name	$N = \{n_{1..n}\}, n_j = name : name \in \mathfrak{D}^{ns}$
Description	$d =$ "*text*"
Time stamp	$\check{t} \in \mathfrak{D}^{\check{t}}$
Location	$\check{L} = \{\check{l}_{1..n}\} : \check{l}_l \in \mathfrak{D}^{\check{l}}$
Type	$\Pi = \{\pi_{1..n}\} : \pi_j \in \mathfrak{D}^{\pi}$
Relationship	$\Phi = \{\phi_{1..n}\} : \phi_j \in \mathfrak{D}^r = \langle n, d, z_1, z_2 \rangle$
- multi	$\Phi^{\phi} = \{\phi^{\phi}_{1..n}\} : \phi^{\phi}_j = \langle n, d, \phi_1, \phi_2 \rangle$
Language	$\hat{L} = \{\hat{l}_{1..n}\}$
Statement	$\Lambda = \{\lambda_{1..n}\}, \lambda_j = \langle \hat{l}, \{expression\} \rangle$
Strategy	$\Xi = \{\xi_{1..n}\}, \xi_j = \langle n, d, \pi, \lambda \rangle$
Typed value	$V = \{v_{1..n}\}, v_j = \langle id, value \rangle : id \in F$
- definition	$F = \{f_{1..n}\}, f_j = \langle id, key, d \rangle : key \in \Upsilon$
- declaration	$\Upsilon = \{y_{1..n}\}, y_j = \langle key, type, d \rangle : type \in \Pi^y$
Event	$E = \{e_{1..n}\}, e_j = \langle n, d, \check{t}, V^e \rangle : n \in \Gamma$
- definition	$\Gamma = \{\gamma_{1..n}\}, \gamma_j = \langle n, d, \Upsilon \rangle$
Context	$C = \{c_{1..n}\}, c_j = \langle n, d, V^c \rangle : n \in \Omega$
- definition	$\Omega = \{\omega_{1..n}\}, \omega_j = \langle n, d, \Upsilon \rangle$

TABLE II: RM-Pol Domain Model Concepts

Application	$A = \{a_{1..n}\}, a_j = \langle n, d, \pi, U, U^a \rangle$
- distributed	$A^d = \{a^d_{1..n}\}, a^d_j = \langle n, d, \pi, U, U^a, U^d \rangle$
- unit	$U = \{u_{1..n}\}, u_j = \langle n, d, U, I, F^u \cup \Xi^u \cup P \rangle$
- process	$A^p = \{a^p_{1..n}\}, a^p_j = \langle n, d, A^t \rangle$
- task	$A^t = \{a^t_{1..n}\}, a^t_j = \langle n, d, \pi, A^t, I, F^t \cup \Xi^t \cup P \rangle$
Interface	$I = \{i_{1..n}\}, i_j = \langle n, d, \lambda \rangle$
Resource	$R = \{r_{1..n}\}, r_j = \langle n, d, I, P \rangle$
Processing System	$\Psi^p = \{\psi^p_{1..n}\}, \psi^p_j = \langle n, d, \check{l}, R^{hw}, R^{sw}, A^p \rangle$
Computing System	$\Psi^c = \{\psi^c_{1..n}\}, \psi^c_j = \langle n, d, \{\langle \psi^p, d \rangle\} \rangle$
Layer	$L = \langle l_{1..n} \rangle, l_j = \langle n, d, \{\langle N^{A^p}, \psi^p \rangle\} \rangle$
Domain	$\Delta = \langle \delta_{1..n} \rangle, \delta_j = \langle n, d, N^{A^p} \rangle$
Executor	$x \in U = \langle n, d, U^{\hat{l}}, I^x, F^x \cup \Xi^x \rangle$
Engine	$x^e \in A = \langle n, d, \pi, U^i, U^a \cup X \cup U^{\hat{l}} \rangle$
- cluster	$x^c \in A^d = \langle n, d, \pi, U^i, U^d \cup X^e \cup X^c \rangle$

TABLE III: RM-Pol Application Domain Model

set Π^y. The type set should provide the semantics of the type. For execution the type set should be supported by a language. Once declared, a type can defined. A type definition f binds an identifier `id` to a `key` declared in Υ. Once defined by a concept, the type can be used. Using a typed value v binds an `id` (previously defined in F) to a `value`.

The concepts *Event* and *Context* model information other concepts can accept, process, manipulate, or produce. Here, events transport information between concepts as domain events [43]. Context describes what information a component (e.g. an event, an application, a policy) requires that is external to itself. This well-defined information facilitates semantic interoperability without complicated APIs.

Both, event and context, are essentially containers of typed values. So they need to be defined before they can be used. Their definitions (γ for events, ω for context) bind a name with a type declaration y. Once defined, they can be used for information exchange. An event e associates the name with a time stamp $\check{t}$ (creation) and typed values V^e. A context item c associates the name with typed values V^c.

The similarity allows us to define an invariant equivalence function. Two events (or two context items) are equivalent if $\forall v_1 \in V_1 \wedge v_2 \in V_2$ the `key` and `type` associated with their `id` are identical. The differences between are that events are immutable, and context can have veracity (read/write).

B. Application Domain Model

The ADM main concepts, modeled following the principles introduced in [44], are: *Application*, *Unit*, *Process*, and *Task*. The complete set of concepts is shown in Table III. An in-depth discussion of a 5G application domain is in §3 of [45].

We are using fundamental concepts of distributed operating systems. An application a of type π has units U realizing the purpose of the application and application units U^a for its management (tasks, memory, and I/O). A distributed application a^d extends a with units U^d for the distribution management (the distributed versions of U^a).

When a (distributed) application is executed it becomes an *Application Process* and its units become *Tasks* inside this process. Units and tasks can have interfaces I for interactions and a combination of policies for control and management:

Policy	$P = \{p_{1..n}\}, p_j = \langle n, d, \pi, M^\pi, \Xi^p \rangle$
DFA	$M^\pi = (Q, \Sigma, \delta, q_0, F) : q_0 = \bullet, F = \{\circledcirc\}$
State set	$Q = \{\bullet, \circledcirc\} \cup S^\pi$
State tuple	$S^\pi = \langle s_{1..k} \rangle : \langle \overrightarrow{s^{tr}} \rangle \vee \langle \overrightarrow{s^{to}}, \overrightarrow{s^{ir}_{1..n}} \rangle \vee \langle \overrightarrow{s^{to}}, \overrightarrow{s^{io}_{1..m}}, \overrightarrow{s^{ir}_{1..n}} \rangle$
Alphabet	$\Sigma = E^t_{s_1} \bigcup_{i=1}^{k} E^o_{s_k}$
State	$\overrightarrow{s^{to}} : E^i \to E^o = \langle n, d, \Gamma^i, \sigma, T, \tau_0, Z, O, \Xi^s, \Omega^s \rangle$
- types	$\overrightarrow{s^{ir}} : E^i \to E^r, \overrightarrow{s^{to}} : E^t \to E^o, \overrightarrow{s^{tr}} : E^t \to E^r$
- task set	$T = \{\langle \tau, V^\tau, n^z \rangle\} : \tau_0 \in T$
- finalizer	$Z = \{z_{1..n}\}, z_j = \langle n, d, \lambda^z, F^i, \Xi^z \rangle$
- output	$O = o_0 \cup \{o_{1..n}\}, o_j = \langle \Gamma^o, s_n \rangle$
Task selector	$\sigma = \langle n, d, \lambda^\sigma, \Xi^\sigma \rangle$
Task	$\tau = \langle n, d, \lambda^\tau, F^i, F^o, F^\tau, \Xi^\tau, \Omega^\tau \rangle$

TABLE IV: RM-Pol Policy Domain Model

static parameters F, execution strategies Ξ, and decision policies P. Resources R are applications related to hardware (or abstracted software) resources supporting an application.

There are three orthogonal concepts to group distributed applications. Each application is executed on a processing *System* ψ^p with HW and SW resources. A computing system ψ^c manages multiple processing systems. A *Layer* contains applications of the same scope (for processing and I/O). A *Domain* comprises applications that use similar function(s) mapping from input values (domain) to output values (range).

Finally, we can use these concepts to define an application for logic execution. We start with an application unit called *Executor* x. It can execute a concept that contains some logic statements λ using one or more execution strategies. Once executed as a task, the task type can be used to indicate which languages the *executor* supports. Supported languages and their specific execution environment are included or can be provided by the surrounding application. This application is called an *Engine* x^e. Engines can aggregate one or more executors and provide the required execution environments $U^{\hat{l}}$. A $Cluster$ then is a distributed application combining engines and other clusters as deeply nested as required. This realizes a composite pattern (cf. pp 163 in [42]) allowing fine-grained deployment configurations (cf. Fig.8).

C. Policy Domain Model

The main concept of the PDM, as shown in Table IV, is a policy p of type π defined by a state machine M^π with optional execution strategies Ξ^p. The type identifies the underlying PM, e.g. *eca* for an ECA policy. The strategies are requirements for a policy executor stating how the policy should be executed.

The core of a *policy* is the state machine or Deterministic Finite Automaton (DFA) M^π, the 5-tuple $(Q, \Sigma, \delta, q_0, F)$. The initial state q_0 defined as $\bullet$ is the entry point for the policy executor x^p. The accepted states F are the exit points back to x^p as $\{\circledcirc\}$. The state set Q contains q_0 and F and the policy states S^ψ. The alphabet Σ is the union of the input events of the first state and all output events of all states in S^π.

In other words, we model a policy as a DFA with a single entry ($\bullet$), a single exit ($\circledcirc$), and a variable set of states in between (S^π). Each state accepts an event e^i defined in Γ^i and produces an event e^o defined in Γ^o. Input events of the

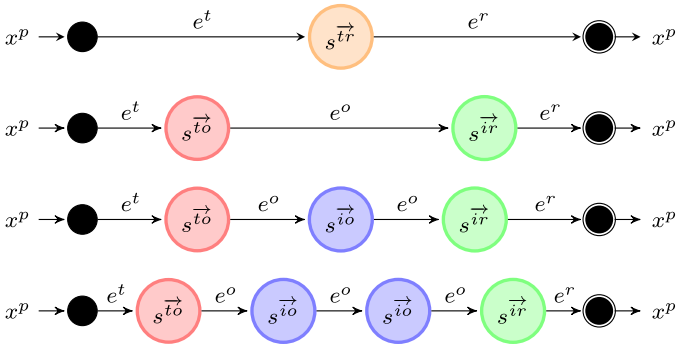


Fig. 2: PDM State Machines

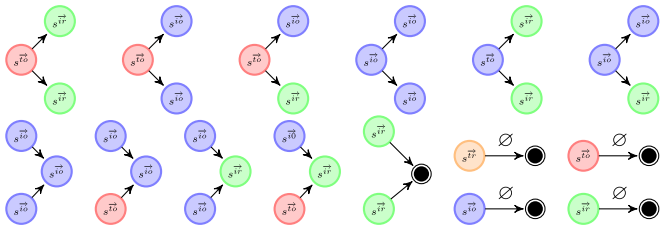


Fig. 3: PDM State Machine Options

first state in S^π are called stimulus events e^t (they trigger or stimulate a policy). Output events of the last state in S^π are called response events e^r (they are the response or actions of the policy). The transition function δ can then be defined as $(s_n, e^o_n) = s_{n+1}$. The initial transition is always $(\bullet, e^t) = s_1$. Final transitions are always $(s_n, e^o_n) = \circledcirc$ with variable s_n.

All states in S^π should be defined according to the underlying PM. For an ECA policy we would for instance define states *event*, *condition*, and *action*. S^π must realize a Directed Acyclic Graph (DAG). A topological sort (e.g. Kahn's algorithm [46]) can be used to test this condition.

This leads to four possible state machines for a sequential (non-branching) execution (in sequence from S^π_1 to S^π_n). The resulting state machines are shown in Fig.2.

1) state $s^{\vec{tr}}$ taking e^t producing e^r. $S^\pi = \langle s^{\vec{tr}} \rangle$.
2) state $s^{\vec{to}}$ accepting e^t producing e^o and state $s^{\vec{ir}}$ accepting e^i producing e^r. $S^\pi = \langle s^{\vec{to}}, s^{\vec{ir}} \rangle$.
3) state $s^{\vec{to}}$, state $s^{\vec{io}}$ accepting e^i producing e^o, and state $s^{\vec{ir}}$. $S^\pi = \langle s^{\vec{to}}, s^{\vec{io}}, s^{\vec{ir}} \rangle$.
4) case 3 variation with n states $s^{\vec{io}}$. $S^\pi = \langle s^{\vec{to}}, s^{\vec{io}}_{1..m}, s^{\vec{ir}} \rangle$.

There are two more sets of situations for the DFA not covered by the simple sequential model. This first set covers more complex policy models, e.g. implementing SON functions as described in [47]. The second set looks at situations were a state does not produce an expected output event.

All these situations are shown as options in Fig.3. For the first set of situations, the figure shows all possible (10) options for branching the DFA. The more of those options are used in a policy, the more the policy becomes a general purpose application. Thus these options need to be used with care to achieve the right balance between a policy and an application. For the second set of situations the figure shows how empty events can transit to the accepted (final) state of the DFA.

With these considerations of cases and options for the state machine we can define the policy *State* concept as shown in Table IV. The standard state $s^{\vec{io}}$ realizes a mapping from E^i to E^o. It is defined by the tuple $\langle n, d, \Gamma^i, \sigma, T, \tau_0, Z, O, \Xi^s, \Omega^s \rangle$. The standard elements of the tuple are the name n, description d, input event definitions Γ^i, and execution strategies Ξ^s. Furthermore, the state might require context defined as Ω^s.

The remaining state elements realize an adaptive decision-making per state with generation of output events realizing a deterministic transition function δ. Using the task selector σ, the state can select a task (implementation of the decision-making logic) from a set of tasks T. In simple (or not-decidable) cases, the default task τ_0 is selected. The possible set of state outputs O provides mappings from possible output events Γ^o to a single next state s_n. Each task $\tau \in T$ has an associated state finalizer z with logic to create an output $o \in O$. For simple states (1 task), a default output o_0 is defined.

Task selector σ, task τ and state finalizer z have their respective logic λ^σ, σ^τ, and λ^s; as well as supported strategies. A task accepts typed values F^i and produces typed values F^o. Input and output values of a task must be member of the state's input and output events. This ensures that tasks can be reused in different states, as long as this condition is met. Tasks can also have parameters (static configurations) F^τ and task-specific context Ω^τ. A finalizer accepts typed values F^i corresponding to the output typed values of the associated task. Finalizer and state output can use the state context.

The other three introduced states differ only in the types of input and output events. The standard state $s^{\vec{tr}}$ realizes a mapping from E^t to E^r, $s^{\vec{to}}$ realizes a mapping from E^t to E^o, and $s^{\vec{ir}}$ realizes a mapping from E^i to E^r.

The PDM concepts, their relationships, and the design patterns they realize are shown in Fig.4 as a UML class diagram. *Events* realize the domain event pattern [43]. Policy *states* S are an expression of the state pattern [42]. They are expressed in the form of a state machine M for execution. *Task Selector* and *Task* represent the strategy pattern [42]. All logic is designed using the foreign code pattern [13].

The concepts can also be mapped to different areas of interest. *Events* realize *input* and *output*. *State*, *Task Selector*, an *Task* represent *execution*. All logic captures *application area* and *domain expertise*.

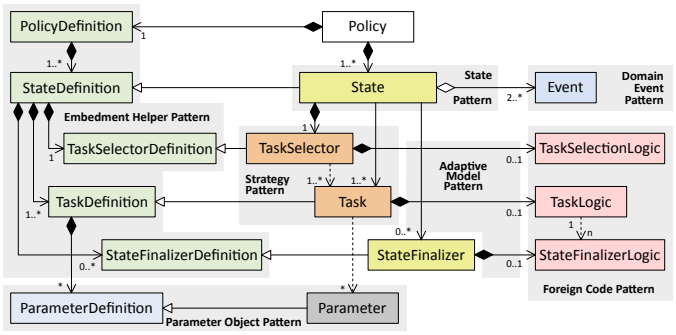


Fig. 4: PDM UML

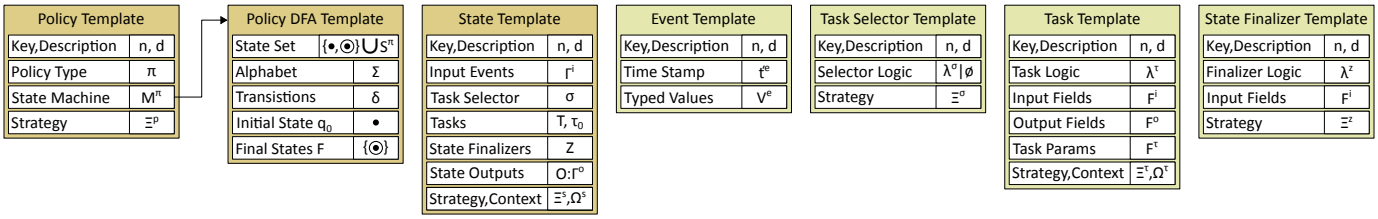


Fig. 5: Model Templates

III. PDM and PM Templates

The RM-Pol can be expressed in form of templates as the basis for a Universal Executable Policy Specification (UEPS). Fig.5 shows the templates. They can be further refined for specific PMs. In combinations, the templates are a blueprint for a concrete expression of a policy as UEPS in structured languages such as JSON [48], YAML [49], or XML [50].

A policy template defines the main concepts of name (key), description, type, state machine, and strategies. It references the DFA with its defined 5-tuple. The DFA's state set is expressed using one or more state templates, each referencing all other templates (event, task selector, task, and finalizer). These general templates can be further refined if required, typically altering policy states: the input/output events and the cardinality of tasks, outputs, and finalizers.

IV. Policy Model Example - ECA

We have modeled several PMs based on the general templates, namely: ECA [14], CA [51], variations of Goal and Utility Function policies [51], SON OM policy using fuzzy logic, OODA [52], and our own adaptive MEDA [53].

For the ECA PM we assume three states: an event s_e uses definition of events to decide if the policy is triggered, a condition s_c evaluates a set of conditions with binary result, and an action s_a fires (enforces) a set of actions.

$$M^{eca} = \left(\begin{array}{l} \{\bullet, \circledcirc\} \cup S^{eca}, E_e^t \cup e_e^o \cup e_c^o \cup E_a^r \cup \{\varnothing\} \\ (\delta(\bullet, e_{1..k}^t) = s_e, \delta(s_e, e_e^o) = s_c, \delta(s_c, e_c^o) = s_a, \\ \delta(s_a, e_{1..k}^r) = \circledcirc, \delta(s_e, \varnothing) = \circledcirc, \delta(s_a, \varnothing) = \circledcirc), \bullet, \{\circledcirc\} \end{array} \right)$$

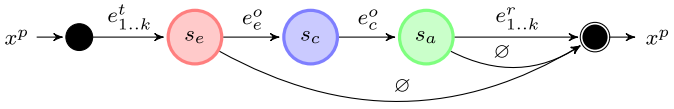


Fig. 6: DFA for an ECA PM

A few conditions apply for an ECA policy: States s_e and s_a can fail, i.e. produce $\varnothing$. A failing s_e means the policy is not triggered. The result of s_c as Boolean, if false, s_a fails, i.e. produces $\varnothing$. These conditions can be formalized as: $\pi = eca$, $S^{eca} = \langle s_e, s_c, s_a \rangle$, and $\gamma_c^o = \gamma_a^i = \langle condition, true|false \rangle$. The resulting state machine can be defined as shown in Fig.6.

Fig.7 shows the all templates for an ECA policy. Grey marked concepts indicate refinements from general templates.

V. Universal Policy Execution Environment

For the design of UPEE we map the PDM execution units to architecture components. The executor x^p is realized by

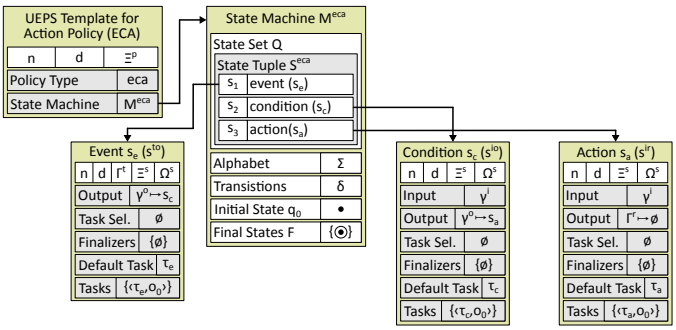


Fig. 7: Templates for an ECA PM

Universal Policy Executor (UPx). The engine x^e is realized by Universal Policy Engine (UPe). A cluster of engines x^c is realized by Universal Policy Engine Cluster (UPec).

There are various options for clustering and deploying UPEE components. Available deployment options are: as library for an application, as component, as a service, in a closed control loop (such as COMPA [54]), and in cloud environments (supported are OpenStack, Docker, and Kubernetes).

Fig.8 shows two clustering examples. Example 1 has one engine with three executors. The engine accepts different stimulus events and produces different response events. This configuration requires some instrumentation to route stimulus events to appropriate executors and created response events to the engine egress interface. Routing can be realized for instance using messaging systems with topics.

Example 2 shows a cluster of clusters. Routing becomes more complex, probably requiring the introduction of name spaces for components. A load-balanced routing solution for high-volume events is detailed in [55]. The distribution of (context) information is detailed in [56].

UPx executes a policy, i.e. the policy's state machine. Since the PDM is invariant, UPx and its algorithms are invariant.

We designed two strategies for executing a policy. The first

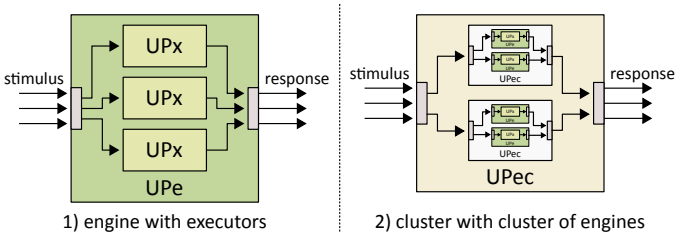


Fig. 8: UPEE Example Clustering Options

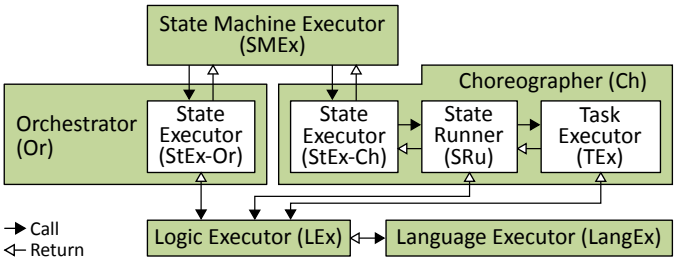
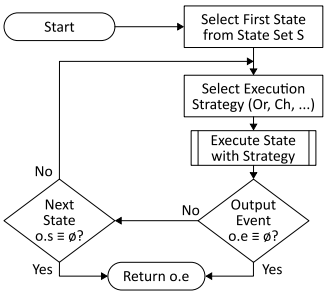
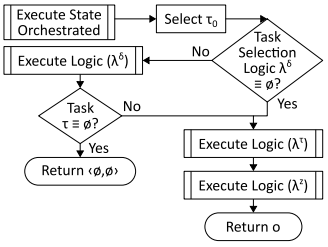


Fig. 9: Universal Policy Executor (UPx)




```
executeSMEx(e^t, p) ≙
    Result: e^r ∨ ∅
    o ← ⟨e^t, p.M.S.s_1⟩
    while o.e ≠ ∅ ∧ o.s ≠ ∅ do
        if Ξ.or then
            o ← stexor(o.s, o.e)
        else
            o ← stexch(o.s, o.e)
        if o.e ∉ o.s.E^i then
            → log(error for o.e)
            o.e ← ∅
    return o.e
```

Fig. 10: UPx State Machine Executor (SMEx)




```
stexor(s, e^i) ≙
    Result: o
    τ ← τ0
    if s_σ ≠ ∅ then
        τ ← lex(λ^σ, e^i, T, τ0, C^s)
    if τ ≡ ∅ then
        return ⟨∅, ∅⟩
    V_ret ← lex(λ^τ, V^i, V^τ, C^τ)
    o ← lex(λ^z, V_ret, s.O)
    return o
```

Fig. 11: UPx State Executor Orchestrated (StExOr)

strategy is called *orchestrated* using a centralized component in full control of the execution. The second strategy is called *choreographed* using a choreographer and other components realizing a de-centralized execution. Both are supported by components realizing actual logic execution for specific languages, for instance a JVM and Jars for Java or a Javascript engine for Javascript logic. Fig.9 shows this architecture.

The UPx decomposition is realized in three steps. In *Step 1* the State Machine Executor (SMEx) executes the first state and then requesting the current state to provide for the next state to be executed, until no further next state is returned. Fig.10 shows the invariant workflow and algorithm for SMEx.

Step 2 Strategy. Our two strategies are realized by the *Choreographer (Ch)* and the *Orchestrator (Or)*. *Or* uses a State Executor Orchestrated (StExOr) for centralized policy execution (cf Fig.11): select default task τ_0, if a task selector exists execute it, if the resulting task is empty (no task selected) return an empty output, otherwise execute the task followed by executing the finalizer, return the generated output.

Ch uses a State Executor Choreographed (StExCh) for policy execution delegating parts of it to a State Runner (SRu) and a Task Executor (TEx). It realizes the same workflow as the *Or* just distributed over those components. This strategy can be important to balance load on large deployments.

```
lex(λ^σ, e^i, T, τ0, C^s) ≙
    Result: τ ∨ ∅
    le ← L.l_λσ
    τ ← le.exec(e^i, T, τ0, C^s)
    if τ ≡ ∅ then
        → log(no τ)
    return τ
```

```
lex(λ^τ, V^i, V^τ, C^τ) ≙
    Result: V_ret ∨ ∅
    le ← L.l_λτ
    V ← le.exec(V^i, V^τ, C^τ)
    if V_ret ≡ ∅ then
        → log(no V_ret)
    return V_ret
```

```
lex(λ^z, V^i, O) ≙
    Result: o
    le ← L.l_λz
    o ← le.exec(V^i, O)
    if o ≡ ∅ then
        → log(no o)
        o ← ⟨∅, ∅⟩
    return o
```

Fig. 12: Logic Executor (LEx) for Task Selector, Task, and State Finalizer

Step 3 Logic Execution. The execution of logic (task selection, tasks, finalizers) is the same for both strategies. The Logic Executor (LEx) is responsible for executing any logic, including required run-time validation. Fig.12 shows the algorithms for all logic execution. The difference lies in the input and output parameters of the functions, logic execution is the same. Language Executor (LangEx) is an implementation (and potentially deployment) specific realization of logic execution. Each language can have specific mechanisms and requirements. Externally provided components, for instance a Java VM, might provide Just In Time (JIT) or Ahead of Time (AOT) compilation. Other language run-time environments might facilitate scripted or byte-code execution.

Our APEX policy engine [15] implements all shown components of UPEE. We support plugins for logic written in Java, Javascript, Python, Ruby, Drools, and XACML. Templates from RM-Pol are expressed in well-defined (schema supported) JSON or XML, deployed to an APEX engine, and directly executed using JAXB [57]. No further translation or transformation is required. Any concrete policy model (for instance the detailed ECA policy) is then *only* a specialization, thus directly executable using the *same* infrastructure.

VI. Conclusion

In this paper, we have defined a Universal Policy Model. Following the principle of "separation of mechanism and policy", UPM identifies and details the invariant parts of *policy* described in the PDM. A number of DDD concepts and design patterns have been used in the model. The PDM model can be translated into templates, effectively creating a UEPS. Execution concepts of the DM have been translated into components for a UPEE architecture. Finally, we have discussed the invariant algorithms of the UPEE components.

UPEE has been implemented in our APEX policy engine [15] *available as open source in the ONAP platform [7]*. A deployment of APEX in North America is described in [54]. In this use case we are using a MEDA PM linked to radio network and security ADMs. Here, we evaluate radio network anomalies and use security components (probes, observability servers) to control nodes and to deal with rogue handsets.

Acknowledgments

This work is partly funded by the European Commission via the ARCFIRE project (Grant 687871) under the H2020 program. Special thanks to John Strassner (policy modelling), Joel Halpern (model template), Joel Fleck II (adaptive policies), Dave Raymer (system architectures), Amy de Buitléir (formal model), William Leahy (guidance), Miguel Ponce de Leon (evaluation), and John Day (RINA and the ADM).

REFERENCES

[1] TM Forum. (2017) Information framework (sid). [Online]. Available: https://www.tmforum.org/information-framework-sid/

[2] ——. (2017) Zero-touch orchestration, operations and management (zoom). [Online]. Available: https://www.tmforum.org/collaboration/zoom-project/

[3] J. Strassner, J. Halpern, and S. van der Meer, "Generic policy information model for simplified use of policy abstractions (supa)," IETF draft, version 3, May 2017. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-supa-generic-policy-info-model/

[4] IETF. (2017) Autonomic networking integrated model and approach (anima). [Online]. Available: https://datatracker.ietf.org/wg/anima/documents/

[5] AT&T, "AT&T Vision Alignment Challenge Technology Survey - AT&T Domain 2.0 Vision White Paper," AT&T, Tech. Rep., Nov. 2013. [Online]. Available: https://www.att.com/Common/about_us/pdf/AT&T%20Domain%202.0%20Vision%20White%20Paper.pdf

[6] ——, "ECOMP (Enhanced Control, Orchestration, Management & Policy) Architecture White Paper," AT&T, Tech. Rep., 2016. [Online]. Available: http://about.att.com/content/dam/snrdocs/ecomp.pdf

[7] ONAP. (2017) Open network automation platform (onap). [Online]. Available: https://wiki.onap.org/

[8] ETSI, "Network functions virtualisation (nfv): Management and orchestration," ETSI GS NFV-MAN 001 V1.1.1, Dec. 2014. [Online]. Available: http://www.etsi.org/deliver/etsi_gs/NFV-MAN/001_099/001/01.01.01_60/gs_NFV-MAN001v010101p.pdf

[9] OSM. (2017) Open source mano (osm). [Online]. Available: https://osm.etsi.org/

[10] G. Rune, E. Westerberg, I. Cagenius, T.and Mas, B. Varga, H. Basilier, and L. Angelin, "Architecture evolution for automation and network programmability," Ericsson Review, Tech. Rep., Nov. 2014. [Online]. Available: http://www.ericsson.com/news/141128-er-architecture-evolution_244099435_c

[11] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf, "Policy/mechanism separation in hydra," *SIGOPS Oper. Syst. Rev.*, vol. 9, no. 5, pp. 132–140, Nov. 1975. [Online]. Available: http://www.cse.psu.edu/~trj1/cse543-f12/docs/p132-levin-hydra.pdf

[12] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison Wesley, 2003.

[13] M. Fowler, *Domain Specific Languages*. Addison-Wesley, Sep. 2010.

[14] N. Damianou, N. Dulay, E. Lupu, and M. Slomam, "The ponder policy specification language," in *International Workshop on Policies for Distributed Systems and Networks (Policy'01)*, May 2001.

[15] L. Fallon, S. van der Meer, and J. Keeney, "Apex: An engine for dynamic adaptive policy execution," in *The 15th IEEE/IFIP Network Operations and Management Symposium (NOMS 2016)*, Apr. 2016. [Online]. Available: https://www.researchgate.net/publication/303564082_Apex_An_Engine_for_Dynamic_Adaptive_Policy_Execution

[16] LaPadula, Leonard J. and Elliott Bell, D., "Secure Computer Systems: Mathematical Foundations," MITRE Technical Report 2547, Volume II, MITRE, Tech. Rep., May 1973. [Online]. Available: http://www-personal.umich.edu/~cja/LPS12b/refs/belllapadula1.pdf

[17] J. E. Dobson and J. A. McDermid, "A framework for expressing models of security policy," in *1989 IEEE Symposium on Security and Privacy*, 1989.

[18] G. Jomier, "A mathematical model for the comparison of static and dynamic memory allocation in a paged system," *IEEE Transactions on Software Engineering*, vol. SE-7, no. 4, pp. 375–385, Jul. 1981.

[19] J. Day, *Patterns in network architecture: A return to fundamentals*. Pearson Education, 2007.

[20] W. B. Rouse and S. H. Rouse, "A model-based approach to policy analysis in library networks," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 9, no. 9, Sep. 1979.

[21] F. Kamoun, "A drop and throttle flow control policy for computer networks," *IEEE Transactions on Communications*, vol. 29, no. 4, Apr. 1981.

[22] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D. McCarthy, A. Rosenthal, S. Sarin, M. J. Carey, M. Livny, and R. Jauhari, "The hipac project: Combining active databases and timing constraints," *SIGMOD Rec.*, vol. 17, no. 1, pp. 51–70, Mar. 1988. [Online]. Available: https://www.researchgate.net/profile/Arnon_Rosenthal/publication/220415822_The_HiPAC_Project_Combining_Active_Databases_and_Timing_Constraints/links/0deec52a0c2fb1be65000000.pdf

[23] M. J. Sloman, "Policy driven management for distributed systems," *Journal of Network and Systems Management*, vol. 2, no. 4, p. 333 – 360, Dec. 1994.

[24] K. Twidle and M. Sloman, "Domain based configuration and name management for distributed systems," in *Workshop on the Future Trends of Distributed Computing Systems in the 1990s (FTDCS)*, Sep. 1988, p. 147 – 153.

[25] D. Robinson and M. Sloman, "Domains: A new approach to distributed system management," in *Workshop on the Future Trends of Distributed Computing Systems in the 1990s (FTDCS)*, Sep. 1988, p. 154 – 163.

[26] J. Moffett, M. Sloman, and K. Twidle, "Specifying discretionary access control policy for distributed systems," *Computer Communications*, vol. 13, no. 9, pp. 571 – 580, Nov. 1990.

[27] R. Boutaba and I. Aib, "Policy-based management: A historical perspective," *Journal of Network and Systems Management*, vol. 15, no. 4, pp. 571 – 580, Nov. 2007.

[28] D. Triantafyllopoulou, A. Kliks, V. Rakovic, and L. Gavrilovska, "Existing policy frameworks: An overview," in *The Tenth International Symposium on Wireless Communication Systems (ISWCS)*, Jun. 2013.

[29] T. Phan, J. Han, J.-G. Schneider, T. Ebringer, and T. Rogers, "A survey of policy-based management approaches for service oriented systems," in *19th Australian Conference on Software Engineering (ASWEC)*, Mar. 2008.

[30] N. C. Damianou, A. K. Bandara, M. S. Sloman, and E. C. Lupu, "A survey of policy specification approaches," Imperial College, London, Tech. Rep., 2002. [Online]. Available: http://www.doc.ic.ac.uk/~mss/Papers/PolicySurvey.pdf

[31] J. Mitola III, "Cognitive radio policy languages," in *IEEE International Conference on Communications*, Jun. 2009.

[32] W. Han and C. Lei, "A survey on policy languages in network and security management," *Computer Networks*, vol. 56, no. 1, pp. 477 – 489, 2012.

[33] G. N. Stone, B. Lundy, and G. G. Xie, "Network policy languages: A survey and a new approach," *IEEE Network*, vol. 15, no. 1, pp. 477 – 489, 2001.

[34] M. I. Yagüe, "Survey on xml-based policy languages for open environments," *Journal of Information Assurance and Security*, vol. 1, pp. 11 – 20, 2006.

[35] J. C. Strassner, *Policy-based Network Management: Solutions for the Next Generation*. Morgan Kaufman Publishers, 2004.

[36] S. Davy, B. Jennings, and J. Strassner, "The policy continuum - policy authoring and conflict analysis," *Computer Communications*, vol. 31, no. 13, Aug. 2008.

[37] K. Barrett, S. Davy, J. Strassner, B. Jennings, S. van der Meer, and W. Donnelly, "A model based approach for policy tool generation and policy analysis," in *First International Global Information Infrastructure Symposium (GIIS)*, Aug. 2007.

[38] R. Brennan, K. Feeney, J. Keeney, D. O'Sullivan, I. Fleck, Joel J., S. Foley, and S. van der Meer, "Multidomain it architectures for next-generation communications service providers," *IEEE Communications Magazin*, vol. 48, no. 8, pp. 110–117, Aug. 2010. [Online]. Available: http://www.tara.tcd.ie/handle/2262/40593

[39] J. Keeney, S. van der Meer, and L. Fallon, "Towards real-time management of virtualized telecommunication networks," in *10th International Conference on Network and Service Management (CNSM)*, Nov. 2014. [Online]. Available: https://www.researchgate.net/publication/272164781_Towards_Real-time_Management_of_Virtualized_Telecommunication_Networks

[40] S. van der Meer, "5g & autonomic networking - challenges in closing the loop," in *IEEE First International 5G Summit*, May 2015. [Online]. Available: http://www.5gsummit.org/docs/slides/Sven-Meer-5GSummit-Princeton-05262015.pdf

[41] J. Day, "Things they never taught you about naming and addressing," in *Asia Future Internet Forum*, Feb. 2010. [Online]. Available: http://csr.bu.edu/rina/KoreaNamingFund100218.pdf

[42] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, Jan., 1 1995.

[43] M. Fowler. (2005, Dec.) Domain event. [Online]. Available: https://martinfowler.com/eaaDev/DomainEvent.html

[44] J. Day, I. Matta, and K. Mattar, "Networking is ipc: a guiding principle to a better internet," in *Proceedings of the 2008 ACM CoNEXT Conference*, Dec. 2008. [Online]. Available: http://www.cs.bu.edu/fac/matta/Papers/IPC-arch-rearch08.pdf

[45] ARCFIRE consortium. (2016, Dec.) H2020 arcfire deliverable d2.2: Converged service provider network design report. [Online]. Available: http://ict-arcfire.eu

[46] A. B. Kahn, "Topological sorting of large networks," *Commun. ACM*, vol. 5, no. 11, pp. 558–562, Nov. 1962. [Online]. Available: http://doi.acm.org/10.1145/368996.369025

[47] T. Bandh, R. Romeikat, H. Sanneck, and H. Tang, "Policy-based coordination and management of son functions," in *12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011)*, May 2011.

[48] ECMA, "The json data interchange format," ECMA International Standard, Oct. 2013. [Online]. Available: https://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

[49] O. Ben-Kiki, C. Evans, and I. döt Net, "Yaml ain't markup language (yaml) version 1.2," 3rd Edition, Patched at 2009-10-01, Oct. 2009. [Online]. Available: http://www.yaml.org/spec/1.2/spec.html

[50] W3C, "Extensible markup language (xml) 1.0," Fifth Edition, Nov. 2008. [Online]. Available: https://www.w3.org/TR/REC-xml/

[51] J. Strassner and J. Kephart, "Autonomic systems and networks: Theory and practice," in *IEEE/IFIP Network Operations and Management Symposium (NOMS 2006)*, Apr. 2006, tutorial.

[52] J. Boyd, "The essence of winning and losing," 1996, unpublished lecture notes. [Online]. Available: http://pogoarchives.org/m/dni/john_boyd_compendium/essence_of_winning_losing.pdf

[53] S. van der Meer, J. Keeney, and L. Fallon, "Dynamically adaptive policies for dynamically adaptive telecommunications networks," in *11th International Conference on Network and Service Management (CNSM)*, Nov. 2015. [Online]. Available: https://www.researchgate.net/publication/282576518_Dynamically_Adaptive_Policies_for_Dynamically_Adaptive_Telecommunications_Networks

[54] L. Fallon, J. Keeney, and S. van der Meer, "Using the compa autonomous architecture for mobile network security," in *2017 IFIP/IEEE International Symposium on Integrated Network Management (IM 2017)*, May 2017. [Online]. Available: https://www.researchgate.net/publication/317014658_Using_the_COMPA_Autonomous_Architecture_for_Mobile_Network_Security

[55] S. Achuthan and L. Fallon, "Load balanced telecommunication event consumption using pools," in *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, May 2015.

[56] L. Fallon, J. Keeney, and S. van der Meer, "Distributed management information models," in *2017 IFIP/IEEE International Symposium on Integrated Network Management (IM 2017)*, May 2017. [Online]. Available: https://www.researchgate.net/publication/316629867_Distributed_Management_Information_Models

[57] Oracle. (2017) Java architecture for xml binding (jaxb). [Online]. Available: http://www.oracle.com/technetwork/articles/javase/index-140168.html