ON CAPABILITY-RELATED ADAPTATION IN NETWORKED SERVICE SYSTEMS

Finn Arve Aagesen and Patcharee Thongtra

Department of Telematics, NTNU, Trondheim, Norway
{finnarve, patt}@item.ntnu.no

ABSTRACT

Adaptability is a property related to engineering as well as to the execution of networked service systems. This publication considers issues of adaptability both within a general and a scoped view. The general view considers issues of adaptation at two levels: 1) System of entities, functions and adaptability types, and 2) Architectures supporting adaptability. Adaptability types defined are capability-related, functionality-related and context-related adaptation. The scoped view of the publication is focusing on capability-related adaptation. A dynamic goal-based policy ontology is presented. The adaptation functionality is realized by the combination of Extended Finite State Machines, Reasoning Machines and Learning Mechanisms. An example case demonstrating the use of a dynamic goal-based policy is presented.

KEYWORDS

Adaptable service systems, Adaptability types, Adaptability architecture, Capability-based adaptation, Goal-based policy ontology, Policy-based adaptation

1. INTRODUCTION

Networked service systems are considered. *Services* are realized by *service components* which by their inter-working constitutes a *service system*. Service components are executed as software components in nodes, which are physical processing units such as servers, routers, switches, PCs and mobile phones. A *service framework* is here defined as a system for the *specification*, *management* and *execution* of *service systems*.

Adaptability can generally be defined as *the ability of a system to fit to changed circumstances*. Adaptability is generally realised by some closed feed-back loop. Fitting behavior can be of various types and can take place several levels. It is tendency, however, to denote many aspects of changes or "fitting behavior" as adaptation, without setting requirements to which changes that "qualifies" with *changed circumstances*. An *autonomic service system* is a specialization of an adaptable service system. Autonomic systems have ability to manage themselves and to adapt dynamically to changes in accordance with given objectives [1,2]. An autonomic system is constituted by distributed components denoted as *autonomic elements*.

The contribution of this publication is a general concept framework for adaptable systems as well as a scoped framework for capability-related adaptation. Adaptable service systems are accordingly considered within a *general view* and within a *scoped view*. Within the *general view* various adaptability issues of networked service systems are considered at two levels. *Entities*, *functions* and *adaptability types* are considered at Level 1, *Architectures* supporting adaptability at Level 2. The adaptability *types* defined are *capability-related*, *functionality-related* and *context-related* adaptation. Within the *scoped view*, *capability-related* adaptation is focused. A goal-based policy ontology for capability-related adaptation is presented. The experience background for this publication is work with the TAPAS *architecture* and *platform* [3-5].

Another important reference is the FOCALE architecture for autonomic networking [6]. Even if parts of this publication are inspired by TAPAS, the intention is to be rather generic.

The rest of the publication is organized as follows. Section 2 considers related works. In Section 3 important performance concepts related to capabilities and services are defined. Section 4-5 presents the Level 1 and 2 issues as defined above. Section 4 presents entities and adaptability functions, while Section 5 presents adaptability types. Section 6 presents adaptability supporting architectures. In Section 7-8 capability-related adaptation is handled. Section 7 defines a goal-based policy ontology, and Section 8 presents an example case demonstrating the use of this ontology. Summary and conclusions are presented in Section 9.

2. Related Works

In Section 1 and the Sections 3-7 some references to related works [1-21] are presented. In this Section additional references [22-28] are provided. Existing service system frameworks that support run-time self-management and adaptation can be classified according to how the management and adaptation functionalities are specified. Some works propose to use templates [22] or adaptation classes [23] for specification. However, such approach lacks flexibility. All possible adaptation cases must be known, and new adaptation cases require re-compilation. The architecture presented in this publication specifies the adaptability functionality based on Extended Finite State Machines, Reasoning Machines and Learning Mechanisms, to be dynamically modified during run-time. For Extended Finite State Machine specifications, an update of changes is done by deployment of the whole specification. For Reasoning Machine and Learning Mechanism, only incremental changes of policies and goals are deployed. The complete policy and goal based functionality, however, need to be validated before deployment of the incremental changes.

There are several works that use policies to specify the adaptation, such as [3, 4], [24], [25-28]. In [24] a framework that defines autonomic applications as dynamic composition of autonomic elements is described. Our approach as well as the approaches described in [3, 4, 25-28] go beyond by adding mechanisms to adapt policies or the way of using policies. Such policy adaptation can be grouped into three categories: 1) changing the policy parameters, considered in [3, 4, 25, 26]; 2) enabling/disabling a policy, found in [3, 4, 25]; 3) using techniques to select the most suitable policy and action; for instance, rewarding policies and their actions, presented in [27, 28]. Our approach is of category 1 and 3. Tesauro et al. [27] presented a hybrid reinforcement technique used for resource allocation in multi-application data centers. This technique is to select optimal policies that can maximize rewards. Mesnier et al. [28] used decision trees to select accurate policies in storage systems.

3. Capability and Service Performance Concepts

A capability is here defined as an *inherent property* of a node used as a basis to implement services. A service component may need capabilities to be allocated before deployment and instantiation. Capability types are classified as *resources*, *functions* and *data*. Resource examples are CPU, memory, transmission links, sensors and batteries. Within network management the concept managed objects is used. Managed objects such as MIB (Management Information Base) objects in SNMP [7] and CIM (Common Information Model) objects in WBEM (Web-based Enterprise Management) [8] are considered as capabilities.

Capability parameter describes the characteristics of a capability type and can be classified as functionality, performance and inference parameters [9]. *Functionality* parameters define functionality features, *performance* parameters define performance measures and *inference* parameters define logical relations to other capability types. Capability *performance parameters* can further be classified as: *capacity* parameters, *state* parameters and *QoS* parameters. Capability capacity parameters examples are transmission channel capacity, the number of streaming connections and CPU processing speed. Capability state parameters examples are number of connections, and the number that is waiting. Capability QoS parameter examples are transfer time, throughput, utilization, availability and recovery time after errors. The services provided to the service user can in the same way as capabilities be described by *functional* and *performance* parameters. *Service performance parameters* are further classified as state and QoS parameters. The capability and service performance provided at an observed time instance or during an observation time interval are denoted as *inherent* capability and service performance.

Service level agreements (SLA) are agreements between the service users and the service provider. The agreement can contain elements such as: required service functionalities and performance, payment for the service when the agreed performance is offered and penalty in case of reduced performance. The service can be differentiated in various *QoS classes*. This QoS class will be reflected in the SLA. A service component can generally handle user services related to several QoS classes.

4. ENTITIES AND FUNCTIONS

4.1. Entities

Entities related to the life-cycle of an adaptable service system are illustrated in Figure 1. The *service framework* is constituted by the *Primary Service System* itself, the *Service Creation System*, the *Service Repository*, and the *Network and Service Management System*. The *environment* is constituted by the administrative *Service Provider* and the *Service User*. The Primary Service System provides services to the Service Users, while the Network and Service Management System provides management services to the Primary Service System. Both the Primary Service System and the Network and Service Management System are service systems according to the definitions in Section 1. The boundary between these systems highly depends on the nature of the primary service, the nature of the service management functionality, and how the various functions are realized in software components. Some management functionalities can be integrated in the software components executing the primary service functionality. In the figure this is denoted as *delegated management*.

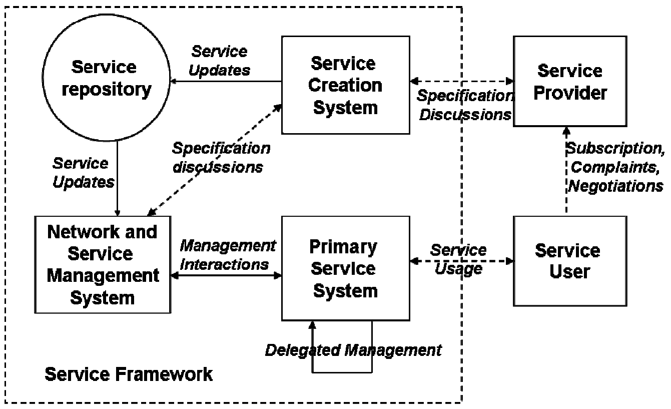

Figure 1. Service life-cycle Entities

4.2. Functions

The functionalities of the service framework entities are here grouped in the *functionality groups*: Service Creation, Node Configuration, Service Configuration, Service Provisioning, and Monitoring and Diagnosis as illustrated in Figure 2. This system of functionality groups and functions are parallel and continuous. The functions in the functionality groups are not necessarily single functions and can in some cases be considered as functionality sub-groups. With reference to Figure 1, Service Creation is done by the Service Creation Node Configuration, Service Configuration, Service Provisioning, and Monitoring and Diagnosis are done *in cooperation* between the Network and Service Management System and the Primary Service System.

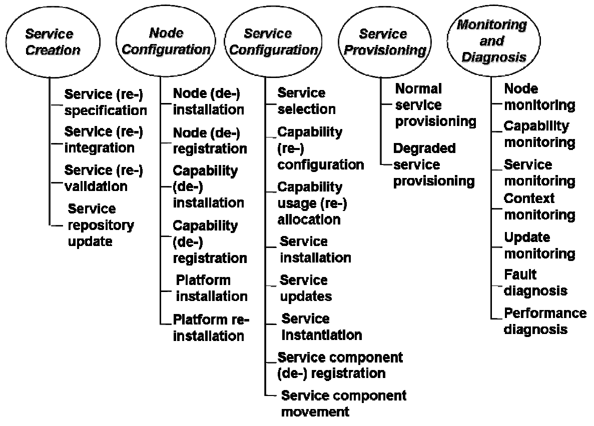

Figure 2. Service Life-cycle Functionalities

Concerning the sub-functionalities of *Node Configuration*, installation is the provision of the physical existence of the component, while registration is the logical registration of nodes and capability instances. The physical installation is done by humans, while the logical registration can be done by a service system. But this functionality will normally require cooperation with a *monitoring function* that can detect the physical presence of the component. Platform re-installation can generally be the installation done after a failure of a node, but can also be the installation of a new version of the platform software.

Concerning the sub-functionalities of *Service Configuration*, *Service selection* is primarily a function used in cases where change of context needs a new service. *Capability configuration* comprises *Capability parameter configuration* as well as *Node selection*. *Capability parameter configuration* is the validation and settings of node capability parameter values according to a capability parameter configuration specification. *Node selection* is the selection of node with respect to the required capability functionality and performance defined for the service. *Capability usage allocation* determines the usage of allocated capabilities. *Service installation* is the deployment of the service components constituting a service system. *Service update* is here indicated as a different function. This is because an update does not need to comprise a new complete installation of all service components constituting a service system. Instantiation starts the execution of the installed or updated service components. *Service component (de-) registration* is the (de-)registration of the service component instance in a system that provides an overview of the instantiated service component instances. *Capability re-configuration* can in general initiate the movement of service components. *Service component movement* is different from installation in general as it includes the movement of an instance of a service component with present states and local variable values

Service Provisioning comprises *Normal service provisioning* as well as *Degraded service provisioning* as realised by the Primary Service System as illustrated in Figure 1. Degraded here refers to degraded service and capability performance that require adaptation actions. With respect to Figure 2, adaptation actions can be *Capability usage re-allocation* only, but in more serious cases it can comprise *Capability re-configuration*, *Capability usage re-allocation*, *Service component movement*, *Service component instantiation*, as well as *Service component de-registration* and Service *component registration*.

Concerning the sub-functionalities of *Monitoring and Diagnosis*, *Node monitoring* monitors the existence and liveness of nodes, *Capability monitoring* monitors the existence of capability types and the parameter values, and *Service monitoring* monitors the existence, liveness, functionality and performance of service components. *Update monitoring* is the monitoring for the existence of service system software updates in the service repository, *Fault diagnosis* detects failures related to nodes, capabilities and service components, and *Performance diagnosis* detects mismatch between required and inherent service and capability performance.

Context is here related to the *adaptability terrain*, i.e. the environment in which the adaptable system operates. The terrain can be classified as *physical* or *logical*. *Physical terrain context* is defined by physical positions as well as the state of the terrain. Physical positions can be absolute position of the node executing the adaptable service system as well as positions relatively to other nodes and objects. State measures can be terrain type, temperature, concentration of gases, friction, fluidity, etc. A *logical terrain* is defined by *logical positions* and application layer *associations* between service components. Physical terrain examples can be streets, buildings on fire, collapsed buildings, water and nature. One adaptable system scenario is a robot snake [10] operating in a house on fire changing the movement type according to the terrain type as well as to blocked ways. Service examples are measurements, searching humans, video recording and the spraying of water. As nodes operating in a physical terrain also can have wire-less communication, they will both operate in a physical and logical terrain. One example service system is a robot soccer play where the various players in the team has roles and interact logically in addition to the behaviour defined from the monitoring made by physical sensors [11]. Context can also be defined to include capabilities, such as memory, CPU, battery, bandwidth as well as user preferences and profile [12]. User preferences and profile are here considered as data and are not visible in the functionality models presented.

5. ADAPTABILITY TYPES

Adaptability types are here classified as Capability-related, Functionality-related and Context-related adaptation. *Capability-related* adaptation is here defined as the ability to adapt because of shortage of capabilities with appropriate logical functionality or overload or failure. *Functionality-related* adaptation is the ability to adapt to new functionality requirements, and *Context-related* adaptation is the ability to adapt to context changes as defined in Section 3.

As capabilities are the basic fundament for the implementation of service systems, a needed property for any adaptability type is *to be aware of node and capabilities* and the ability to *configure service systems* according to the present availability of nodes and capabilities. This is denoted as *basic capability awareness.* The needed functionality is illustrated in Figure 3. In this case there is assumingly no failures and overload. Installation of new versions of the platform software is not part of the functionality considered in the architecture models presented in this publication. The functionalities needed for the various adaptability types are defined as follows:

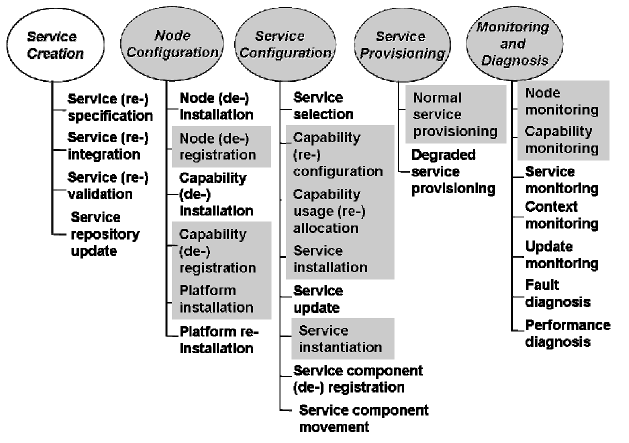

Figure 3. Basic Capability Awareness

1) *Capability-related adaptation* needs functionality as illustrated in Figure 4. The functionality needed for basic capability awareness is illustrated in grey. The added needed functionality needed is illustrated in blue. In addition to the Basic Capability Awareness Platform (re-) installation, Service component (de-) registration, Service component movement, Degraded service provisioning, Service monitoring, Fault diagnosis and Performance diagnosis is needed.

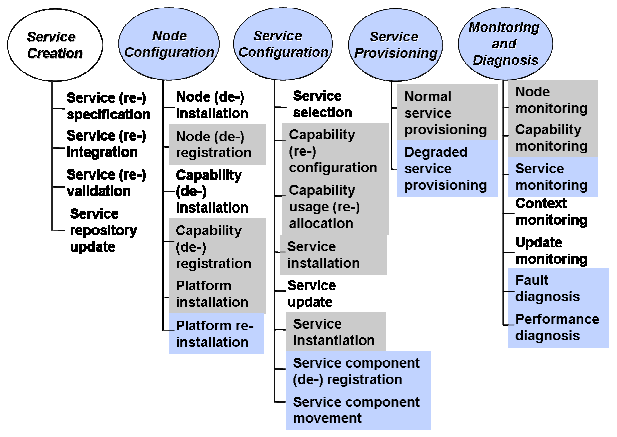

Figure 4. Capability-Related Adaptation

2) *Functionality-related adaptation*: In addition to the functionalities of Capability-related adaptation, the following functionality is needed: Service (re-) specification, Service (re-) integration, Service (re-) validaton, Service repository update, Update monotoring and Service updating. Fault and failure diagnosis are included for covering the cases where faults or performance leads to redesign of the service system software. Degraded service provisioning covers cases where new software needs capability re-configuration because of reduced performance. Service creation functionalities must have involvement by humans. The functionalities that can be automated are Service repository update and Service update [13].

3) *Context-related adaptation*: In addition to Basic capability awareness the following functionality is needed: Service Monitoring, Context monitoring and Service Selection. Context-related adaptation is related to terrain as defined in Section 4. Context defined by physical position is used in a wide variety of commercial user services related to ticket sales, visiting touristic places, restaurant services etc. [12]. Context-related adaptation is to some

extension pre-programmed. Physical terrain adaptation is programmed mathematically. Logical adaptation is realized by client applications, i.e. client service components that flexibly are able to interwork with new server service components. Mostly user interaction is also needed to select among the available applications.

6. ADAPTABILITY SUPPORTING ARCHITECTURES

6.1. General

The service framework is constituted by a computing architecture and a service functionality architecture. This is the same architecture structure as applied in TINA [14]. The computing architecture has concepts for the specification of service system behaviour. The architecture that describes the structure of services functionalities is denoted as service functionality architecture. An execution platform must support the concepts of the computing architecture and also provide management support for the service systems as illustrated in Figure 5. The management functionality which is a part of the service functionality architecture comprises both network and service management functionality.

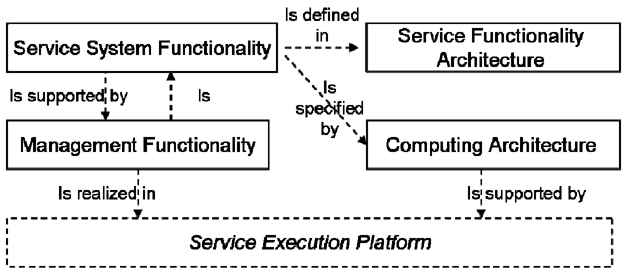

Figure 5. Functionality, Architecture and Platform

6.2. Computing Architecture

6.2.1. Service Layer and Physical Layer

A computing architecture based on two abstraction layers is illustrated in Figure 6. These two layers are denoted as service layer and physical layer. This is a generalization of the three-layer TAPAS model [3]. Several architecture models with a variable number of layers have been proposed. In [15], a five-layer model is presented.

The service layer defines the service constituted by service components. Leaf service components are realized by Actors. Actors are executed as operating system software components. The service system is specified by EFSMs (Extended Finite State Machines), goals and policies. Actors are EFSM interpreting mechanisms. The service components are implemented by a combination of Actors, Reasoning Machine (RM) and Learning Mechanisms (LM). EFSMs can activate RM and LM, and the EFSM can use the RM to select appropriate actions in situation when strategic decisions are needed. RM and LM models will be described in Section 7.

6.2.2. Service Component Features

The service components have features in addition to the service behavior execution. The following features must be supported: i) Renewal of Service Component EFSM behavior specification ii) Renewal of Policy and Goal specifications, iii) Movement of Service component while preserving state, variables and messages, and iv) Management of Service component EFSM states.

These features need support of management functionality. During runtime new versions EFSM functionality, policy and goal specifications can be downloaded and instantiated. Service components can be moved by instantiating a new Actor in a new node. Some of the EFSM states are classified as stable states. A stable state is a state where the functionality of a service component can move safely and be re-instantiated. Re-instantiation includes the restoration of EFSM state, EFSM local variables, and queued messages.

Actors can manage its EFSM states and local variables based on received EFSM input messages as well as responses from the Reasoning Machine. The generic Actor states are: {Initial, Normal, Degraded, Moving, Idle, Terminated}. A service component is instantiated in Initial state in a node where the required capability functionalities and performances are met. In Normal state services are provided with satisfactory performance. In Degraded state the performance is considered too low. Adaptation can be initiated to return to Normal state. In Moving state an Actor is moved and re-instantiated in a new node. In Idle state the execution of current EFSM specification is ended and allocated capabilities are released. From this state an Actor can be initiated as new service component in Initial State.

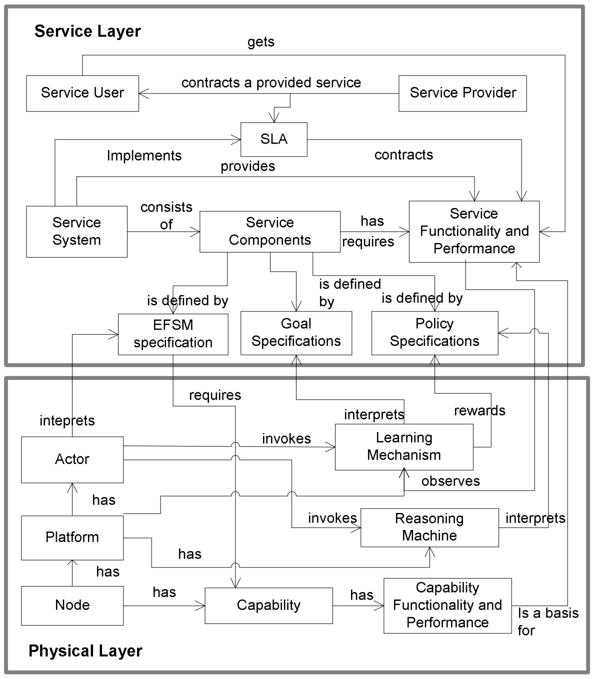

Figure 6. Computing Architecture

6.2.3. Computing Architecture and Adaptability Features

As previously stated, the adaptability functionality is realized by a combination of concepts and mechanisms of the computing architecture and supporting management functionality. The computing architecture has a service layer and a physical layer and has functionality and performance concepts both related to capability and service. This provides a basis for basic capability-related awareness, which is the basis for all adaptation types defined. The architecture further opens for service systems defined by flexible combination of EFSM-, goal-, and policy specifications combined with Learning Mechanisms. The two-layer architecture combined with the flexible Actor execution behaviour also makes adaptation during runtime possible. The model as defined, however, is open with respect to the goal and policy ontology applied and also to the realization of Reasoning Machine and Learning Mechanism. This basic model should make it possible to define and implement all adaptation types as defined in Section 5.

Concerning functionality-related adaptation, it is emphasized that is only the service configuration features of functionality-related adaptation that is feasible. Concerning context-related adaptation the model is open to any use of sensor capabilities. This model is however a generic model. In specific application cases, refinement and elaboration is needed.

6.3. Service Functionality Architecture

The service functionality architecture consists of primary service functionalities and the management functionality components as illustrated in Figure 7. The following five repositories are defined: Service specification repository (SpcRep), Capability type repository (CapRep), Inherent capability and service repository (InhRep), Context repository (ConRep) and Platform repository (PltRep). SpcRep stores the services behavior specifications, SLAs and required capability functionality and performance specifications. CapRep stores the capability type concepts. Several languages are used for capability type definition. Examples are SNMP MIB-objects defined by ASN.1 [7], WBEM CIM objects defined by XML [8] and NETCONF [16] objects based on YANG [17]. In TAPAS platform [5] capability ontology is represented by OWL [18] and OWL/XDD [19]. InhRep stores data about available nodes, capability instances and instantiated service components. This comprises the address of the Actor realizing the service component, service type reference, state of the Actor, and Capability and Service performance parameter values. ConRep stores predefined context states while PltRep stores programs needed to execute service functionality. With respect to the physical view of the computing architecture, PltRep comprises Actor, Reasoning Machine and Learning Mechanism software.

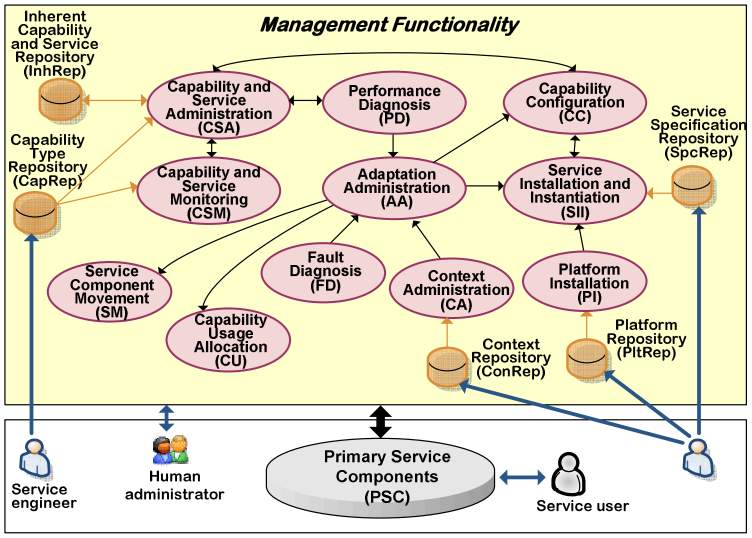
Management Functionality
Inherent Capability and Service Repository (InhRep)
Capability and Service Administration (CSA)
Performance Diagnosis (PD)
Capability Configuration (CC)
Service Specification Repository (SpcRep)
Capability Type Repository (CapRep)
Capability and Service Monitoring (CSM)
Adaptation Administration (AA)
Service Installation and Instantiation (SII)
Service Component Movement (SM)
Fault Diagnosis (FD)
Context Administration (CA)
Platform Installation (PI)
Capability Usage Allocation (CU)
Context Repository (ConRep)
Platform Repository (PltRep)
Service engineer
Human administrator
Primary Service Components (PSC)
Service user

Figure 7. Service Functionality Architecture

The functionality components as illustrated in Figure 7 implement functions for capability-related, functionality-related and context-related adaptation as defined in Section 5. The functions Capability Configuration (CC), Capability Usage Allocation (CU), Service Component Movement (SM), Platform Installation (PI), Fault Diagnosis (FD) and Performance Diagnosis (PD) correspond directly to functions defined in Section 4.

Capability and Service Administration (CSA) performs Node (de-) registration, Capability (de-) registration and Service component (de-)registration. A view of InhRep is also provided.

Capability and Service Monitoring (CSM) performs Node, Capability and Service monitoring. The result of the monitoring is reported to CSA. *Capability Configuration* (CC) generates configuration plans for service components. A configuration plan defines the node for deployment and instantiation. *Capability Usage Allocation* (CU) allocates capabilities in accordance with the present performance, SLAs and the optimization criteria chosen by the service provider. *Service Installation and instantiation* (SII) comprises deployment of EFSM, goal and policy specifications, as well as the execution of the configuration plan. *Service Component Movement* (SM) manages the ongoing sessions on behalf of a moving service component during movement. In TAPAS it forwards received messages after a service component is re-instantiated in a new location. SM will by broadcasting inform other service components of the “Moving” and “Normal” states of the moving component.

Context Administration (CA) monitors context and initiates configuration of context-dependent application software. Platform Installation (PI) is bootstrap functionality installing needed platform software when nodes are (re-) started. *Adaptation Administration* (AA) plans and administers re-configuration initiated by non-wanted events or states during the normal service system execution. This can be initiated by FD, PD, CA, or by a human administrator. The re-configuration can result in CU only, or the combination of CC and CU, including SM and SII.

Primary service components basically implement the service provisioning as illustrated in Figure 2. There is, however, not always a clear boundary between primary service functionality and management functionality. Most primary service systems need capabilities and functionality components such as *PD*, *CU* and *CC.* Such functionalities can often be designed as part of the primary service system.

7. A Goal-based Policy Ontology For Capability-related Adaptation

An ontology is a formal and explicit specification of a *shared* conceptualization [20], containing both object types and functions operating on instances of object types. We can define independent concepts and relational concepts. Logic concepts can be defined by mathematical logics, e.g., if-then-else or by rules [9].

7.1. A Static Model based on Reasoning Machine

Figure 8 presents a goal-based policy ontology. At the top level we have *goal*, *policy* and *inherent state*. This model is denoted as *static* because there is no feed-back from a Learning Mechanism that rewards actions that have the ability to bring the system to states that complies with the defined goals.

As a basis for the optimal adaptation, required performance as well as prices and penalty agreements defined in the SLAs must be taken into consideration. Service income includes the estimated income paid by the users for using services in normal QoS conditions and the penalty cost paid back to the users when the service qualities and functionalities are lower than defined by SLA. In general, goal, policy and inherent state concepts have the SLA class as a parameter. The inherent states of the service components can comprise measures related to functionality, performance and income. The goal is defined by a goal expression and a weight. The goal expression defines a required system performance or service income measure. A goal example is: “Service response time of premium service SLA class < 2 secs”. The goal weight identifies a goal's importance. A goal can be associated with a set of policies. A policy is defined by *conditions*, *constraints* and *actions*. The *condition* defines the activation of the policy execution. The *constraint* restricts the usage of the policy, and is described by an expression of required and inherent functionality and performance of services and capabilities, required and inherent service incomes, available nodes and their capabilities, as well as system time. A policy example related to the goal example given above is: "*If CPU utilization > 95% and the time is between 18:00-24:00, ignore new service requests of users of ordinary SLA classes that request service time > 2 mins*". It is expressed with Conditions: CPU utilization > 95%, Constraints: system time between 18:00-24:00 and service time request > 2 minutes, and Actions: ignore new service requests of users of ordinary SLA classes.

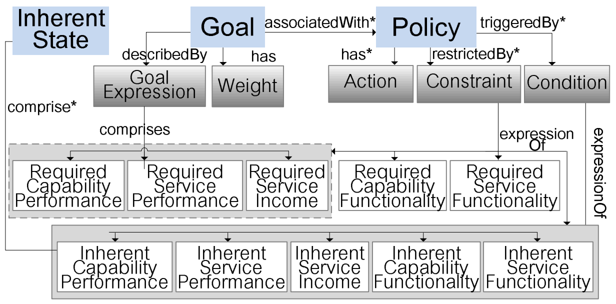
Inherent State
Goal
associatedWith*
Policy
triggeredBy*
describedBy
has
has*
restrictedBy*
Goal Expression
Weight
Action
Constraint
Condition
comprise*
comprises
expression Of
expressionOf
Required Capability Performance
Required Service Performance
Required Service Income
Required Capability Functionality
Required Service Functionality
Inherent Capability Performance
Inherent Service Performance
Inherent Service Income
Inherent Capability Functionality
Inherent Service Functionality

Figure 8. A Static Goal-based Policy Ontology

Table 1 lists notations used for capability, service and income concepts. The *required capability* functionality and performance are service component defined requirements. *Required service* functionality and performance are SLA defined requirements.

Table 1. Capability, service and income concepts notation

Notation	**Description**
$\hat{C}_R$	Required capability performance set
$\hat{C}_I$	Inherent capability performance set
$\overline{C}_R$	Required capability functionality set
$\overline{C}_I$	Inherent capability functionality set
$\hat{C}_{A,n}$	Set of available capabilities in node n; n=[1, N]
$\hat{S}_R$	Required service performance set
$\hat{S}_I$	Inherent service performance set
$\overline{S}_R$	Required service functionality set
$\overline{S}_I$	Inherent service functionality set
I_R	Required service income
I_I	Inherent service income

An RM model R extended from [3, 4] is now defined as:

$$R \equiv \{ \Theta, \Phi, \Pi, \xi \} \tag{1}$$

Here Θ is a set of query expressions with variables, Φ is a generic *reasoning procedure,* Π is a set of policies and ξ is the data including the inherent states values. The expression (1) can be further elaborated as follows:

$$\xi \equiv (\overline{S}_I , \hat{S}_I, \overline{C}_I , \hat{C}_I, I_I, \hat{C}_{A,n}; n=[1, N]) \tag{2}$$

$$\Pi \equiv \{ p_i \} \tag{3}$$

$$p_i \equiv (\Sigma_i, X_i, A_i) \tag{4}$$

$$\Sigma_i \equiv \text{Expression}(\overline{S}_I , \hat{S}_I, \overline{C}_I , \hat{C}_I, I_I) \tag{5}$$

$$X_i \equiv \text{Expression}(\overline{S}_R , \hat{S}_R, \overline{C}_R , \hat{C}_R, I_R, \overline{S}_I , \hat{S}_I, \overline{C}_I , \hat{C}_I, I_I, \hat{C}_{A,n}; n=[1, N], \Gamma) \tag{6}$$

A policy p_i has conditions Σ_i, constraints X_i and actions A_i. The condition is an expression of the inherent states of relevant service components. The constraint is an expression of required functionality and performance of services and capabilities, required service incomes, relevant service components, available nodes and their capabilities, as well as the system clock time Γ.

A reasoning procedure is applied to select appropriate actions with maximum accumulated rewards. It is based on *Equivalent transformation (ET)* [21], which solves a given problem by finding values for the variables of the queries. The conditions, constraints and actions can have variables. The result of the reasoning procedure can, in addition to actions, give instantiated variables.

7.2. A Dynamic Model based on a Learning Mechanism

The static model presented in the previous subsection can be made dynamic by introducing a Learning Mechanism (LM) and by adding the parameters *Accumulated Reward* and *Operation Cost* in the model as illustrated in Figure 9.

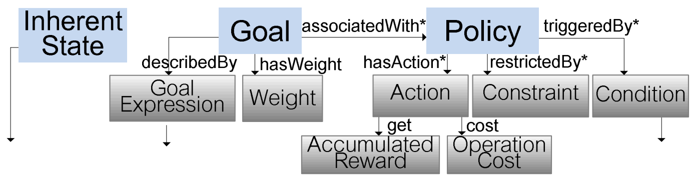

Figure 9. Making a Dynamic Goal-based Policy Ontology.

The proposed LM model gives rewards to actions to be selected by RM. The reward is measure for the ability *to move towards a state with goal performance and income measures.* The rewards will be accumulated over a period of time. The *LM* model L is defined as:

$$L \equiv \{ \Omega, \Lambda, \Psi, \zeta \} \tag{7}$$

where Ω is a set of *goals*, Λ is a generic *rewarding procedure*, Ψ is a *reward database* storing the accumulated rewards of actions, and ζ is the LM data including the inherent states from this service component as well as other service components. We further have:

$$\zeta \equiv (\overline{S_I} , \hat{S}_I, \overline{C_I} , \hat{C}_I, I_I) \tag{8}$$

$$\Omega \equiv \{ g_k \} \tag{9}$$

$$g_k \equiv (d_k, \omega_k) \tag{10}$$

A goal g_k has goal expression d_k and weight ω_k. The sum of the goal weights is equal to 1. At time t, the rewarding procedure will calculate the reward of an action a_i, which was applied at time t-1 as:

$$\text{reward}(a_i,i_{k,t-1},d_k) = (\Delta(i_{k,t},i_{k,t-1})/\Delta(d_k,i_{k,t-1})) * \omega_k - \text{cost}(a_i) \tag{11}$$

where $i_{k,t-1}$ and $i_{k,t}$ are an inherent state measure before and after applying the action for an monitoring interval [t-1, t], $i_k \in \zeta$ and d_k is an associated goal required measures. $\Delta(i_{k,t},i_{k,t-1})$ is the difference between $i_{k,t}$ and $i_{k,t-1}$. $\Delta(d_k,i_{k,t-1})$ is the difference between d_k and $i_{k,t-1}$. ω_k is the goal weight and cost(a_i) is the operation cost of a_i. The measure accumulated_reward($a_i,i_{k,t-1},d_k$), is then the sum of the rewards of an action a_i for an inherent state measure $i_{k,t-1}$ and a goal measure d_k. Equation (2) is accordingly modified as follows to include the reward database Ψ:

$$\xi \equiv (\overline{S_I} , \hat{S}_I, \overline{C_I} , \hat{C}_I, I_I, \Psi, \hat{C}_{A,n}; n=[1, N]) \tag{2'}$$

8. Case Study

An example music video streaming system is presented with the intention to demonstrate the *Reasoning Machine* and *Learning Mechanism* behaviour models as presented in Section 7. The streaming system is illustrated in Figure 10. The streaming case is the same as applied in a previous work [4]. The goal-based policy and the Learning Mechanism, however, were not applied in [4].

The system is constituted by the following service components implementing the service functionalities as defined in Section 6: *Capability and Service Administration (CSA)*, *Capability and Service Monitoring (CSM)*, *Fault Diagnosis (FD)*, *Capability Configuration (CC)*, *Service Installation and Instantiation* (*II*), *Service Component Movement (SM*) and *Primary Service Components (PSC)*.

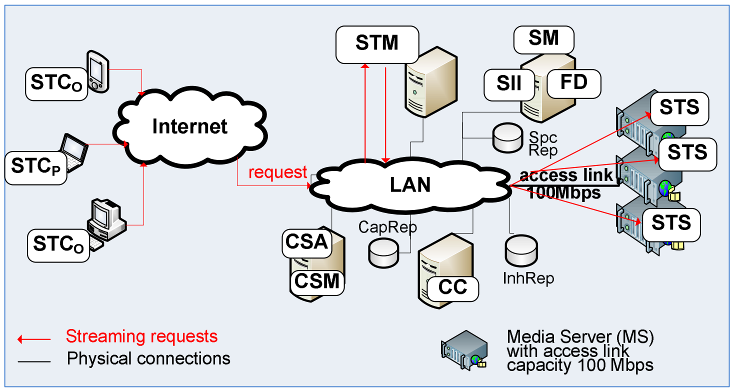

Figure 10. An Example Streaming System

In accordance with the previous discussion of boundary between primary service functionality and service management functionality, *Capability Usage Allocation* as well as *Performance Diagnosis* as defined in Section 4, is now realized by PSCs. The PSCs are S*treaming Client (STC)*, *Streaming Manager (STM)* and *Streaming Server (STS)*. An STS, executing on a *media streaming server (MS)*, streams the music video files to STCs. STM will accept the streaming requests on behalf of STSs. STM will decide which STS that can serve the requests, or STM may put them in waiting queues. STM can also instantiate a new STS in an available MS without executing STS.

An STC is associated with an SLA class, which defines *required streaming throughput*, *price for the service* and *service provider penalties* if the agreed QoS cannot be met. Two SLA classes are applied: *premium (P)* and *ordinary (O)*. An STC is denoted by its SLA class as STC_P or STC_O. Each SLA class has different required throughput (X); the STC_P required throughput (X_P) can be 1Mbps or 600Kbps for high-resolution and degraded fair-resolution videos, while the STC_O required throughput (X_O) is 500Kbps for low-resolution videos. The MS's required access link capacity ($C_{R,AL}$) is set to 100 Mbps. The number of STCs that can use the service at a time is limited by the MS access link capacity. When the required streaming throughput cannot be provided, a STC needs to wait until some streaming connections have finished. An STC_O can be disconnected, while an STC_P may have to degrade the video resolution. The service provider will pay penalties in case of waiting and disconnection of the STC.

The penalty and price functions are given in Table 2. A cost unit is the price paid by an ordinary client for one second streaming of the rate 500Kpbs. The price function for using the service is M(SLA_Class,X) (cost units/second). The penalty function for waiting is P_{WAIT}(SLA_Class) (cost units/second), and the penalty function for disconnection is P_{DISC}(SLA_Class) (cost units/connection).

Table 2. Prices and penalty functions

	STC_O (X_O=500Kbps)	STC_P (X_P=600Kbps)	STC_P (X_P=1Mbps)
M(SLA_Class,X)/s	1	1.875	2
P_{WAIT}(SLA_Class)/s	5	10	10
P_{DISC}(SLA_Class)/Connection	10	-	-

The complete set of actions $A = \{a_D, a_B, a_N, a_I, a_R, a_T, a_M\}$ and a subset $Á=A–\{a_M\}$.The action a_D disconnects the ordinary clients, a_B decreases the throughput of the premium clients. The action a_N instantiates a MS, a_I instantiates a new STS, a_R disconnects a MS, a_T terminates an STS and a_M moves connected client sessions from an STS to another STS. These actions are selected by the Reasoning Machine of STM. STM executes a_N, a_I, a_R and a_T, while STM suggests a_D, a_B and a_M to STSs.

The considered capability is the MS access link. The required and inherent capability performance sets are denoted as $\hat{C}_R \equiv \{C_{R,AL}\}$ and $\hat{C}_I \equiv \{C_{I,AL}\}$, where $C_{R,AL}$ is the required access link capacity, and $C_{I,AL}$ is the available access link capacity. The inherent service performance set $\hat{S}_I$ consists of the number of connected and waiting premium and ordinary clients ($N_{Con,P}$, $N_{Con,O}$, $N_{Wait,P}$, $N_{Wait,O}$), the number of disconnected ordinary clients ($N_{Disc,O}$), the number of MS (N_{Node}), the service time and waiting time of premium and ordinary clients ($T_{Serv,P}$, $T_{Serv,O}$, $T_{Wait,P}$, $T_{Wait,O}$). These values as well as the inherent service income (I_I) are observed per a monitoring interval Δ. The service income is defined as:

$$I_I = M(STC_O,X_O)*T_{Serv,O} + M(STC_P,X_P)*T_{Serv,P} - P_{WAIT}(STC_O)*T_{Wait,O} - P_{WAIT}(STC_P)*T_{Wait,P} - P_{DISC}(STC_O)*N_{Disc,O} - P_{Ser}*N_{Node}*\Delta \quad (13)$$

where P_{Ser} is the cost function for adding a new MS which is 150 units/second per node, while M(SLA_Class,X), P_{WAIT}(SLA_Class) and P_{DISC}(SLA_Class) are as already defined in Table 2.

8.1. RM and LM Specification

In this case study, STM plays an important role. Its RM specification is defined as follows:

$$R_{STM} \equiv \{ \Theta_{STM}, \Phi, \Pi_{STM}, \xi_{STM} \} \quad (14)$$

Π_{STM} consists of five policies (p_1-p_5) as presented in Appendix. The LM applied by STM is defined as follows:

$$L_{STM} \equiv \{ \Omega_{STM}, \Lambda, \Psi_{STM}, \zeta_{STM} \} \quad (15)$$
$$\Omega_{STM} \equiv \{ g_1, g_2\} \quad (16)$$
$$g_1 \equiv (d_1: I_R > 0, \omega_1: 0.8) \quad (17)$$
$$g_2 \equiv (d_2: T_{Wait} < \Delta, \omega_2: 0.2) \quad (18)$$

Here I_R is the required service income, and T_{Wait} is the sum of the waiting time of premium and ordinary clients. These goals are set in order to gain high income and to avoid high waiting time. The policies p_1-p_5 can be used when the required service income is not met, while the policies p_1-p_3 are used when the waiting time is higher than expected.

8.2. Experiments and Results

The measures considered are *accumulated service income* and the *accumulated waiting time*. The streaming request arrivals are modelled as a Poisson process with an arrival intensity parameter λ_{SLA_Class}. The duration of streaming connections d_{SLA_Class} is constant and is set to 10 minutes. The traffic per MS access link ρ is defined as:

$$\rho = ((\lambda_P * d_P * X_P) + (\lambda_O * d_O * X_O)) / (N_{Node} * C_{I,AL}) \tag{19}$$

The monitoring interval Δ is 1 minute. The STCs will stop waiting, and there is no penalty for waiting after 10 minutes. The number of available MS = 3. Initially, only one STS in one MS is instantiated. Three cases are considered:

I. The complete set of actions A is used,
II. The action subset Á is used.
III. Action set is A as in Case I, but no *Learning Mechanism* is used.

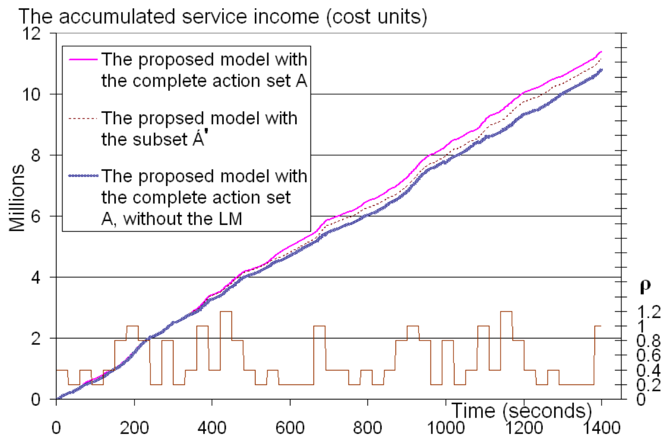

Figure 11. Accumulated service income.

Figure 11 and 12 show the accumulated service income and the accumulated waiting time of three cases I, II and III. The traffics offered are a function of time. The time with ρ at a fixed level, denoted as the *ρ period*, is set to 30 minutes. ρ varies from 0.2 to 1.2. λ_P is set to 50% of the total arrival intensity.

The brown line in these figures shows the variation of ρ. In Case I, the system learned that {a_M, a_T and a_R}, which move connected STC sessions, terminate an STS and disconnect a MS consecutively, is efficient to adapt the system when ρ drops and then the required service income is not met. As a result, Case I could produce the highest accumulated service income and the lowest accumulated waiting time. For the last case, the actions were selected randomly and they were not appropriate to the states of unwanted service income and the waiting time. So, the accumulated service income of Case III was the lowest, while the accumulated waiting time was the highest. So the Learning Mechanism applied has positive influence on both service income and waiting time.

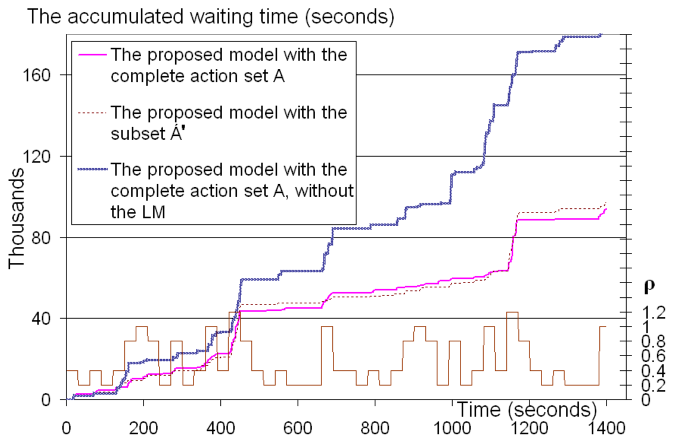

Figure 12. Accumulated waiting time.

9. Summary and Conclusions

Issues of adaptability of networked service systems both within a general and a scoped view have been presented. The general view considers issues of adaptation at two levels: 1) System of entities and functionalities related to the service system life-cycle, and adaptability types with required adaptability functionality, and 2) Architectures supporting adaptability. The adaptability types defined are capability-related, functionality-related and context-related adaptation. The architecture supporting adaptability is constituted by a computing architecture and service functionality architecture. The computing architecture has two layers represented by a service layer and a physical layer. The adaptability functionality is realized by Actors interpreting EFSM specifications supported by a Reasoning Machine and a Learning Mechanism. The presented computing architecture has functionality and performance concepts both related to capability and service. This provides a basis for basic capability-related awareness, which is the basis for all adaptation types as defined in Section 5. The architecture further opens for service systems defined by flexible combination of EFSM-, goal-, and policy specifications combined with Learning Mechanisms. The two-level architecture combined with the flexible Actor execution behavior also makes adaptation during runtime possible. This basic model should make it possible to define and implement all adaptation functionalities as defined in Section 5. In specific application cases, refinement and elaboration is needed.

The scoped view considers capability-related adaptation. A goal-based policy ontology was presented. It is realized by EFSMs, Reasoning Machine and Learning Mechanism. Finally a case study was presented to demonstrate the use and efficiency of the goal-based policy ontology as well as its realization by concrete policies and learning algorithm. Performance measures considered are service income and waiting time. For the considered cases, the Learning Mechanism has positive influence on the performance measures considered.

References

[1] J. O. Kephart and D. M. Chess. "The Vision of Autonomic Computing", IEEE Computer Society, January 2003, pp. 41-47.

[2] S. White, J. Hanson, I. Whalley, D. Chess, and J. Kephart. "An architectural approach to autonomic computing", In Proc. of 1st IEEE Int. Conf. on autonomic computing, New York, May 2004, pp. 2–9.

[3] P. Supadulchai and F. A. Aagesen. "Policy-based Adaptable Service Systems Architecture", In Proc. of 21st IEEE Int. Conf. on Advanced Information Networking and Applications (AINA'07), Canada, 2007.

[4] P. Supadulchai, F. A. Aagesen and P. Thongtra. "Towards Policy-Supported Adaptable Service Systems", EUNICE 13th EUNICE and IFIP TC6.6 Workshop on Dependable and Adaptable Networks and Services. Lecture Notes in Computer Science (LCNS) 4606, pp 128-140.

[5] P. Thongtra and F. A. Aagesen. "An Adaptable Capability Monitoring System", In Proc. of 6th Int. Conference on Networking and Services (ICNS 2010), Mexico, March, 2010.

[6] J. C. Strassner, N. Agoulmine, and E. Lehtihet. "FOCALE - A Novel Autonomic Networking Architecture", Latin American Autonomic Computing Symposium (LAACS), 2006, Campo Grande, MS, Brazil.

[7] K. McCloghrie, et al. RFC 2578, Structure of Management Information Version 2 (SMIv2), April 1999.

[8] A. Westerinen and J. C. Strassner. "Common Information Model (CIM) Core Model", Version 2.4, Distributed Management Task Force White Paper DSP0111, August 2000.

[9] P. Thongtra and F. A. Aagesen. "Capability Ontology in Adaptable Service System Framework", In Proc. of 5th Int. Multi-Conference on Computing in the Global Information Technology, Spain, Sep 2010.

[10] P. Liljeback. *Modelling, Development and Control of Snake Robots*, PhD Thesis at NTNU, 2011, ISBN 978-82-471-2667-7.

[11] H. Skubch, M. Wagner, R. Reichle, and K. Geihs. "A modelling language for cooperative plans in highly dynamic domains", Elsvier Mechatronics 21 (2011) 423–433.

[12] K. Geihs, C. Evers, R. Reichle, M. Wagner, and M. U. Khan. "Development Support for QoS-Aware Service-Adaptation in Ubiquitous Computing Applications", SAC'11, March 21-25, 2011, TaiChung, Taiwan, ACM 978-1-4503-0113-8/11/03.

[13] R. Di Cosmo, D. Di Ruscioa, P. Pelliccionea, A. Pierantonioa, and S. Zacchiroli. "Supporting Software Evolution", Component-Based FOSS Systems, Science of Computer Programming Volume 76, Issue 12, 1 December 2011, pp. 1144-1160, Special Issue on Software Evolution, Adaptability and Variability.

[14] Y. Inoue, et al. *The TINA Book: A Co-operative Solution for a Competitive World*, Prentice Hall, 1999.

[15] S. van der Meer1, et al. "Autonomic Networking: Prototype Implementation of the Policy Continuum", In Proc. of 1st IEEE International Workshop on Broadband Convergence Networks (BCN 2006), pages 1-10, IEEE, New York, 2006.

[16] R. Enns. RFC 4741, Network Configuration Protocol (NETCONF).

[17] M. Bjorklund. RFC 6020, YANG - A Data Modelling Language for the Network Configuration Protocol (NETCONF), October 2010

[18] W3C, OWL Web Ontology Language Overview, 2004. Available at: http://www.w3.org/TR/owl-features/

[19] V. Wuwonse and M. Yoshikawa, "Towards a language for metadata schemas for interoperability", In Proc. of 4th Int. Conf. on Dublin Core and Metadata Applications, China, 2004.

[20] R. Studer, V. R. Benjamins, and D. Fensel, "Knowledge Engineering: Principles and methods," Data & Knowledge Engineering, vol. 25, pp. 161-197, 1998.

[21] K. Akama, T. Shimitsu, and E. Miyamoto. Solving Problems by Equivalent Transformation of Declarative Programs. In Journal of the Japanese Society of Artificial Intelligence, vol. 13, pp. 944-952, 1998.

[22] F. Berman, et al. "Adaptive computing on the grid using AppLeS", IEEE Trans. Parallel Distributed System, vol. 14, no. 4, pp. 369–382, Apr. 2003.

[23] P. Boinot, R. Marlet, J. Noy´e, G. Muller, and C. Cosell. "A declarative approach for designing and developing adaptive components", In Proc. of the 15th IEEE Int. Conf. on Automated Software Engineering, 2000.

[24] H. Liu and M. Parashar. "Accord: a programming framework for autonomic applications", In IEEE Trans. on System, Man, and Cybernetics, vol. 36, pp. 341–352, 2006.

[25] L. Lymberopoulos, E. C. Lupu and M. S. Sloman. "An Adaptive Policy-Based Framework for Network Services Management", In Journal of Networks and Systems Management, vol. 11, pp. 277–303, 2003.

[26] K. Yoshihara, M. Isomura, and H. Horiuchi. "Distributed Policy-based Management Enabling Policy Adaptation on Monitoring using Active Network Technology", In Proc. of 12th IFIP/IEEE Int. Workshop on Distributed Systems: Operations and Management, France, Oct 2001.

[27] G. Tesauro, R. Das, N.K. Jong, and M.N. Bennani. "A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation", In Proc. of 3rd IEEE Int. Conf. on Autonomic Computing (ICAC'06), Ireland, Jun 2006, pp. 65–73.

[28] M. Mesnier, et al. "File classification in self-* storage systems", In Proc. of Int. Conf. on Autonomic Computing (ICAC-04), pp. 44–51.

APPENDIX: POLICY SPECIFICATIONS

The five policies (p_1-p_5) used in the case study are specified by OWL [18] and OWL/XDD [19]. Variables are integrated with ordinary OWL elements and are prefixed with $.

p_1	**Conditions:** $\$I_I <= 0$ or $\$T_{Wait} >= \Delta$, **Constraints:** $P_{WAIT}(STC_O) < P_{WAIT}(STC_P)$, **Actions:** $\{a_D\}$ **Operation Cost:** a_D costs $P_{DISC}(STC_O)$ units. This policy can be read as: a_D should be used to disconnect a list of STC_O when $P_{WAIT}(STC_O) < P_{WAIT}(STC_P)$, and the number of STC_O being disconnected is calculated from $X_{P,1Mbps}$ * $\$N_{Wait,P}$ / X_O.
p_2	**Conditions:** $\$I_I <= 0$ or $\$T_{Wait} >= \Delta$, **Constraints:** $P_{WAIT}(STC_O) > M(STC_P,X_{P,1Mbps})-M(STC_P,X_{P,600Kbps})$, **Actions:** $\{a_B\}$, **Operation Cost:** a_B costs $M(STC_P,X_{P,1Mbps})$ - $M(STC_P,X_{P,600Kbps})$ This policy can be read as: a_B should be used to decrease the throughput of a list of STC_P when $P_{WAIT}(STC_O) >$ $M(STC_P,X_{P,1Mbps})$ - $M(STC_P,X_{P,600Kbps})$, and the number of STC_P to decrease the throughput is calculated from X_O * $\$N_{Wait,O}$ / $(X_{P,1Mbps}$ - $X_{P,600Kbps})$.
p_3	**Conditions:** $\$I_I <= 0$ or $\$T_{Wait} >= \Delta$, **Constraints:** $(X_{P, 1Mbps} * \$N_{Wait,P} + X_O * \$N_{Wait,O}) / C_{R,AL} > 0.1$, **Actions:** $\{a_N, a_I\}$, **Operation Cost:** The actions $\{a_N, a_I\}$ cost $P_{Ser} * \Delta$ This policy can be read as: a_N and a_I should be used to instantiate a MS and to instantiate a new STS consecutively, when $(X_{P, 1Mbps} * \$N_{Wait,P} + X_O * \$N_{Wait,O}) / C_{R,AL} > 0.1$.
p_4	**Conditions:** $\$I_I <= 0$, **Constraints:** $(X_{P, 1Mbps} * \$N_{Wait,P} + X_O * \$N_{Wait,O}) / C_{R,AL} < 0.1$, **Actions:** $\{a_T, a_R\}$, **Operation Cost:** The actions $\{a_T, a_R\}$ cost $P_{DISC}(STC_O) + P_{WAIT}(STC_P) – P_{Ser} * \Delta$ This policy can be read as: a_T and a_R should be used to terminate an STS and to disconnect a MS consecutively, when $(X_{P, 1Mbps} * \$N_{Wait,P} + X_O * \$N_{Wait,O}) / C_{R,AL} < 0.1$.
p_5	**Conditions:** $\$I_I <= 0$, **Constraints:** $(X_{P, 1Mbps} * \$N_{Wait,P} + X_O * \$N_{Wait,O}) / C_{R,AL} < 0.1$, **Actions:** $\{a_M, a_T, a_R\}$, **Operation Cost:** These actions $\{a_M, a_T, a_R\}$ make profit $P_{Ser} * \Delta$ This policy can be read as: a_M, a_T and a_R should be used to move connected STC sessions, to terminate an STS and to disconnect a MS consecutively, when $(X_{P, 1Mbps} * \$N_{Wait,P} + X_O * \$N_{Wait,O}) / C_{R,AL} < 0.1$.

Authors Short Biography

Finn Arve Aagesen is Professor at the Department of telematics at NTNU (Norwegian University of Science and Technology), Norway. He has a PhD in Communication Technology from NTNU in 1977. He is the Norwegian delegate to IFIP TC6 Communication since 1996. His general research interests include network and service management, systems engineering and adaptable systems. His recent research interest also includes platforms for Smart Grid control systems.

Patcharee Thongtra received the B.E. degree in computer engineering from Chulalongkorn University, Thailand, in 1999, and the M.Sc. degree in computer science from Asian Institute of Technology, Thailand, in 2002. Currently, she is a PhD candidate at Department of Telematics, NTNU (Norwegian University of Science and Technology), Norway. Her research interests include adaptable systems, autonomic computing, network and service management, semantic web, and software engineering.