A Framework for Dynamic Service Composition

Paramai Supadulchai and Finn Arve Aagesen
Department of Telematics, Norwegian University of Science and Technology (NTNU)
{paramai, finnarve}@item.ntnu.no

Abstract

To be able to utilize the generative potential of future networks for service composition, the attributes of services and networks must be appropriately formalized, stored and made available. Important attributes are the capability and the status. A capability is an inherent property of a node or a user, which defines the ability to do something. A capability in a network node is a feature available to implement services. A capability of a user is a feature that makes the user capable of using services. Status is a measure for the situation in a system. This paper proposes a representation framework for capability and status, denoted as Unified Capability and Status Representation Framework (UniCS). This framework is used to decide upon dynamic use of capabilities, and is used to support the dynamic composition of a service system. UniCS consists of facts and configuration rules. The facts describe the availability and requirement of capabilities and status of a service system. The configuration rules verify, manipulate, transform and discover new facts with defined axioms and constraints. An instance of UniCS is the input specification for a reasoning engine to dynamically generate a composition plan for a service system.

1. Introduction

A network-based service system consisting of services, service components and nodes is considered. A *service* is realized by the structural and behavioral arrangement of *service components*, which by their inter working provide a service in the role of a *service provider* to a *service user*. Service components are executed as *software components* in *nodes*, which are physical processing units such as servers, routers, switches and user terminals. User terminals can be phones, laptops, PCs and PDAs etc.

Traditionally, the nodes as well as the service components have a predefined functionality. However, changes are taking place. Nodes are getting more generic and can have any kind of capabilities such as MP3, camera and storage. The software components have been also changed from being *static* components to become more *dynamic* and be able to download and execute different functionality depending on the need. Such generic programs are from now on denoted as *actors*. The name actor is chosen because of the analogy with the actor in the theatre, which is able to play different roles in different plays.

We are entering a generative era, which gives a high degree of flexibility. To utilize the generative potential, the attributes of services, service components, software components and nodes must be appropriately formalized, stored and made available. As a first further step towards this formalization, the concepts capability and status are introduced.

A capability is an inherent property of a node or a user, which defines the ability to do something. A capability in a node is a feature available to implement services. An actor executes a program, which may need capabilities in the node. A capability of a user is the feature that makes the user capable of using services. Capabilities can be classified into:

- Resources: physical hardware components with finite capacity,
- Functions: pure software or combined software/ hardware component performing particular tasks,
- Data: just data, the interpretation, validity and life span of which depend on the context of the usage.

Status is a measure for the situation in a system with respect to the number of active entities, the traffic situation and the Quality of Services (QoS) etc. Status reflects an instantaneous state of the system.

Rather than using the traditional approach that nodes and service components have a pre-defined functionality, the functionality can be composed from several cooperating actors hosted in nodes. We propose a representation framework for capability and status, denoted as Unified Capability and Status Representation Framework (UniCS). This framework is used to decide upon dynamic use of capabilities, and is used to support the dynamic composition of a service system. UniCS consists of facts and configuration rules. The facts describe the availability and requirement of capabilities and status of a service system. The configuration rules verify, manipulate, transform and discover new facts with defined axioms and constraints. An instance of UniCS is the specification given as an input to a reasoning engine to generate a composition plan for a service system.

The work presented in this paper has been related to the Telematics Architecture for Play-based Adaptable System (TAPAS) [1]. Section 2 discusses related work. Section 3 gives some TAPAS concepts, which are extensions to the generic concepts already defined. Section 4 gives an overview of UniCS. Section 5 describes the methodology of the dynamic composition of a service system. Section 6 gives a summary and presents our conclusions.

2. Related work

Several research activities are related to capability representations [5,8,9,11]. A similar work to UniCS presented in this paper in term of objectives, functionalities and architecture, is a Resource Definition Framework (RDF)-based knowledge model for network management [5], which provides an analogous framework to describe capability facts in RDF. However, it requires additional framework(s) to describe configuration rules. Directory Enabled Network NGOSS (DEN-ng) [9] describes the configuration rule partially in term of constraints in a specific language, i.e. Object Constraint Language (OCL). However, OCL limits the power of DEN-ng to only verifying facts, while not permitting it to manipulate, transform or discover them.

3. Necessary TAPAS concepts

The Telematics Architecture for Play-based Adaptable System (TAPAS) intends to be an architecture for *autonomic network-based systems* that gives *rearrangement flexibility*, *failure robustness* and *resource load awareness and control* [1]. In analogy with the TINA architecture [3], the TAPAS architecture is separated into a *service architecture* and a *computing architecture* as follows.

- The *service* architecture is an architecture showing the structure of *services* and *services components*.
- The *computing* architecture is a generic architecture for the modeling of any service *software components*.

These architectures are not independent and can be seen as architectures at different abstraction layers. The service architecture, however, has focus on the functionality independent of implementation, and the computing architecture has focus on the modeling of functionality with respect to implementation, but independent of the nature of the functionality.

The relationship of services and service components in the service architecture are realized by the computing architecture, which will be the focus of this paper.

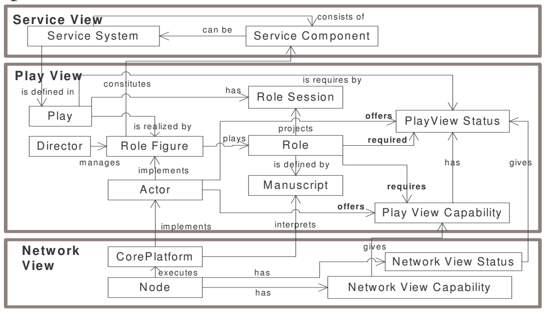


Figure 1 the TAPAS computing architecture

3.1. TAPAS computing architecture

TAPAS computing architecture has three layers: the *service view*, the *play view* and the *network view* as shown in Figure 1. **The service view** concepts are rather generic and should be consistent with any service architecture. **The network view** concepts are consistent with any corresponding network architecture, with exception of the *core platform*, which is a specific platform supporting the play view concepts. The network view consists of *nodes*, which are typically network processing units such as mobile phone, desktop computer, laptop, printer and router that possess particular *Network View Capabilities*, from now on abbreviated as *NV-Capabilities*. Nodes are installed in the *core platform.* At a specific time point, a *Network View Status* denoted as *NV-Status* is the state of a system with respect to the number of active entities, traffic situation and QoS etc.

The play view is a basis for designing functionality that can meet the *rearrangement*, the *robustness*, the *survivability*, the *QoS* awareness and *resource control* properties. The play view concepts are seemingly rearrangement flexibility oriented. The *capability and status* concepts, however, also give a basis for the further design of the robustness, the survivability, the QoS awareness and resource control properties.

3.2. TAPAS theater metaphor

The play view is founded on a theater metaphor. The TAPAS actor is a generic software component consistent with the actor definition given in Section 1. However, the TAPAS actor is specialized as follows. Actors perform *roles* according to predefined manuscripts, and a director manages their performance. Actors are software components in the nodes that can download manuscripts. They have *Play View Capabilities and Status* abbreviated as *PV-Capabilities and -Status*, which are transformed from NV-Capability and -Status of the nodes. The transformation is also based on UniCS and is referred to [10]. An actor will constitute a role figure by behaving according to a manuscript that defines the functional behavior of that particular role in a play. A role session is a projection of the behavior of a role figure with respect to one of its interacting role figures. Actors can be moved between nodes and their role sessions can be re-instantiated automatically [6].

A director is an actor with supervisory status regarding other actors. When the director needs to choose a fitting actor for a certain role figure, he requests help from *a service manager*, which is a dedicated role figure to generate a composition plan for the dynamic service composition. The director reads the generated composition plan and assigns role-based manuscripts to the recommended actors. The actor interprets the manuscript and behaves accordingly. The utilization of manuscripts is beyond the scope of this paper and is referred to [7].

A service system is defined by *a play*. A play consists of several actors playing different roles, each possibly having different PV-Capabilities and –Status requirements. *An actor will constitute a role figure, which will constitute a service component based on a role defined by a manuscript*. The ability of an actor to play a role depends on the matching of the required PV-Capabilities and -Status of the role and the offered PV-Capabilities and -Status of the actor [1].

4. Unified Capability and Status Representation Framework (UniCS)

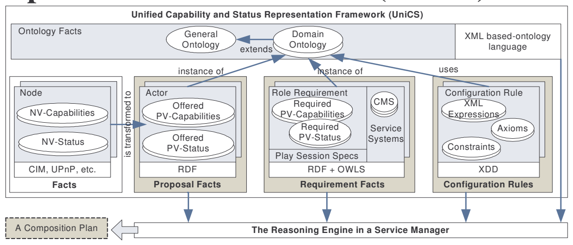
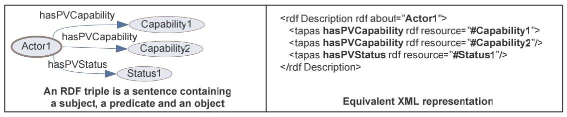


Figure 2 the UniCS framework

As already defined in Section 3.2, the behaviors of service components are based on *roles*. To allocate an actor to a specific role, the information of available actors and their PV-Capabilities and -Status is needed. This capability and status information must be described in a *formal* and *machine-understandable* way. *Unified Capability and Status Representation Framework (UniCS)* as shown in Figure 2 is a *unified* representation and an *executable* framework for capability and status information.

UniCS is used to represent the requirement on how to dynamically compose a service system. This requirement is given to the reasoning engine in a service manager to generate a composition plan, which suggests appropriate actors to play roles and become the service components of the service system.

4.1 Syntactic representation


hasPVCapability
hasPVCapability
hasPVStatus
Actor1
Capability1
Capability2
Status1
An RDF triple is a sentence containing a subject, a predicate and an object
Equivalent XML representation

```
<rdf Description rdf about="Actor1">
  <tapas hasPVCapability rdf resource="#Capability1">
  <tapas hasPVCapability rdf resource="#Capability2"/>
  <tapas hasPVStatus rdf resource="#Status1"/>
</rdf Description>
```

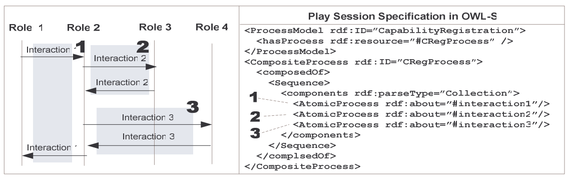
Figure 3 the PV-Capability and -Status representation

UniCS consists of *facts* and *configuration rules* providing a way to separate syntactic and semantic representation respectively. Facts indicate relationships in a system. An example of a fact is *"printer A" "has capability" "duplex printing"*. Concerning only capability and status, facts can be represented in various XML syntaxes. In the network view, any syntax chosen by the manufacturers or the network management program can be used. The examples are Common Information Model encoded in XML (CIM-XML) [11] and Universal Plug-and-Play (UPnP) [8].

In the play view, PV-Capability and -Status are the projection of NV-Capability and -Status. Facts concerning actors and PV-Capabilities and -Status are from now on classified as *proposal facts*. The PV-Capability syntax is carried on RDF. The main reason is because representing facts transformed from various NV-Capabilities and –Status syntax is rather easy with the RDF construct triple [10]. Facts in RDF can also be seamlessly facilitated with *additional domain ontology* from any XML-based ontology language.

The facts concerning roles in a service system are defined as *play session specification* and *role requirement*. A play session specification is a set of all *role sessions* in a play constituting a service component. A play session is specified by *an atomic process* in the *Web Ontology Language for Semantic Web (OWL-S)* [4] and has two associative roles: *invoker* and *server*. The invoker role generally has no PV-Capability and -Status requirement. The server role requires specific PV-Capabilities and -Status. Figure 4 shows a play session specification example.

At the time of writing, OWL-S is incapable of describing the server roles' required PV-Capabilities and -Status, which are essential criteria in the composition of a service system. We propose the using role requirement, modeled in RDF, as an extension to each play session to describe the roles' required PV-Capabilities and –Status. Role requirement representation is similar to Figure 3 except that an actor is replaced by a role as the subject of the fact. Facts concerning the play session specification and the role requirement of a service system are classified as *requirement facts*. The requirement facts and the proposal facts will be matched by *configuration rules*.



```
<ProcessModel rdf:ID="CapabilityRegistration">
  <hasProcess rdf:resource="#CRegProcess" />
</ProcessModel>
<CompositeProcess rdf:ID="CRegProcess">
  <composedOf>
    <Sequence>
      <components rdf:parseType="Collection">
1       <AtomicProcess rdf:about="#interaction1"/>
2       <AtomicProcess rdf:about="#interaction2"/>
3       <AtomicProcess rdf:about="#interaction3"/>
      </components>
    </Sequence>
  </complsedOf>
</CompositeProcess>
```

Figure 4 play session specification

4.2. Semantic representation

Table 1 Types of XML variables

Type	Instantiation and examples
N	*XML element or attribute names* Ex: `<$N:var1>…</$N:var1>` can be instantiated to `<actor>...</actor>` or `<node>...</node>`
S	*XML string* Ex: `<prop name="$S:var1"/>` can be instantiated into `<prop name="prop1"/>` or `<prop name="prop2"/>`
P	*Sequence of zero or more attribute-value pairs* Ex: `<element $P:var1/>` can be instantiated into `<element/>` or `<element name="1"/>`
E	*Sequence of zero or more XML expressions* Ex: `<element>$E:var1</element>` can be instantiated into `<element/>` or `<element><value>1</value></element>`
I	*Part of XML expressions* Ex: `<$I:var1><attr/></$I:var1>` can be instantiated into `<element><prop><attr/></prop></element>`

Configuration rules are defined in a semantic web language, XML Declarative Description (XDD) [12]. XDD is an XML-based knowledge representation, which extends ordinary, well-formed XML elements by incorporation of variables for an enhancement of expressive power and representation of implicit information into so-called XML expressions. Ordinary XML elements – XML expression without variables – are called ground XML expressions. Every component of an XML expression can contain variables as in Table1. Every variable is prefixed with '$*T*:' where *T* denotes its type.

A configuration rule is an XML clause of the form:

$$H, \{C_1, \ldots C_m\} \rightarrow B_1, \ldots B_n$$

where m, $n \geq 0$, H and B_i are XML expressions. And each of the C_i is a predefined XML condition used to limit the rule for a certain circumstances. This allows the modeling of *constraints* for a rule. *Axioms* are defined from one or more rule(s) [11]. The XML expression H is called the *head* of the clause. The set of B_i is the body of the clause. When the body is empty, such a clause is referred to an XML unit clause, and the symbol '$\rightarrow$' will be omitted. Hence any facts in form of XML elements or documents can be mapped directly onto a ground XML unit clause.

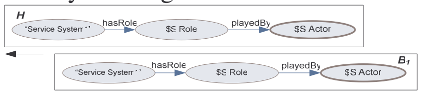


Figure 5 the graphical representation of the query clause

4.3 UniCS reasoning mechanism

Intuitively, the UniCS reasoning process begins with an XML expression based query. The reasoning engine formulates an XML clause from the query of the form:

$$Q \leftarrow Q$$

The XML expression Q represents the constructer of the expected answer which can be derived *if* all the bodies of the clause hold. However, if one or more XML expression bodies still contain XML variables. These variables must be matched and resolved from other rules.

A body from the query clause will be matched with the head of each rule. At the beginning, there is only one body Q. Consider a rule R_1 in the form:

$$R_1\text{: } H, C_1 \rightarrow B_1, B_2$$

If the XML structure of the body Q of the clause and the head H of the rule R_1 match without violating condition C_1, the body Q will be transformed into B_1 and B_2. All XML variables in the head Q and the new bodies B_1 and B_2 of the query clause will be instantiated. The query clause will be in the form:

$$Q^* \leftarrow B_1^*, B_2^*$$

Where X^* means the one or more variables in the XML expression X has been instantiated and removed.

The transformation process ends when either 1) the query clause has been transformed into a unit clause or 2) there is no rule that can transform the current bodies B_i of the query clause. If the constructor Q is transformed successfully into Q_f that contain no XML variable, the reasoning process ends and a desired answer is obtained. Due to the space limitation, the details how the reasoning engine performs the rule matching and the variable instantiation will not be presented in this paper. The reader should be referred to [12].

5. Service composition

Three generic configuration rules for composing any service system are formulated. For each rule, a graphical representation of RDF triples together with the various types of XML variables is used instead of the equivalent XML clause due to the space limitation. These rules and the query clause are as follows:

5.1. The query clause (Figure 5):

The query clause contains the name of the service system to be composed, which can be changed. The meaning of both head and body of the clause is that

"*Service System 1*" can have a role "*$S:Role*", which can be played by an actor "*$S:Actor*".

5.2. Rule 1 (Figure 6):

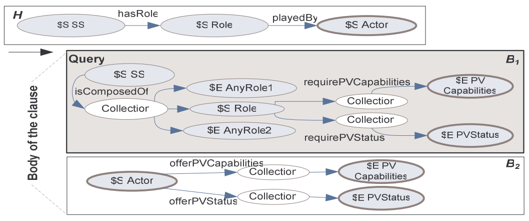


Figure 6 the graphical representation of rule 1

The head *H* of Rule 1 is similar to the body B_1 of the query clause except the variable *$S:SS* that makes this rule applicable to any service system. After the variable *$:SS* has been instantiated with the name of a service system from the query clause, the body B_1 will query the *requirement facts* and find the play session specification and the role requirement of that service system. Each role in *$S:SS* requires PV-Capabilities and -Status, represented by *$E:PV-Capabilities* and *$E:PV-Status*. The body B_2 looks for the *proposal facts* with the capabilities and status represented by the same E-variables. Rule 1 matches the proposal and requirement facts that refer to the same variables.

5.2. Rule 2 (Figure 7):

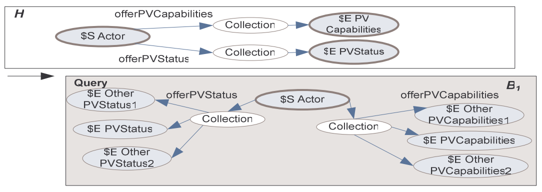


Figure 7 the graphical representation of rule 2

Rule 2 is for querying the actor that has a set of PV-Capability and -Status, represented by *$E:PV-Capability* and *$E:PV-Status*. The body B_1 will query all the proposal facts and find actors that have such a qualification. Additional E-variables in B_1 allows an actor to have PV-Capabilities and -Status in addition to those in *$E:PV-Capabilities* and *$E:PV-Status*.

5.4 The result (Figure 8)

After the execution, the composition plan of *"Service System 1"* is produced as illustrated in Figure 8. Note that Role 2 can be played by both *Actor B* and *C* because they both have the required PV-Capabilities and -Status.

We used XML Equivalent Transformation (XET) [2], a Java-based reasoning engine that transforms the query clause by the XDD-based rules to compose a plan for Capability Management System, a service system that manages capabilities in TAPAS.

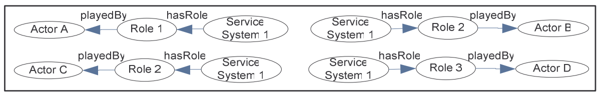


Figure 8 the graphical representation of the composition plan

6. Conclusion

This paper has presented a framework for capability and status representation, denoted as *Unified Capability and Status Representation Framework* (UniCS). This framework has been further used to support the modeling of the dynamic composition of service systems. UniCS consists of facts and configuration rules. Facts are categorized *as proposal facts*, which are actors, PV-Capabilities and -Status, and *requirement facts*, which are play session specification and role requirement. Configuration rules map the proposal facts and the requirement facts and discover a *composition plan*. As a result, a service system can be *dynamically* composed.

10. References

[1] Aagesen, F.A., et al., On Adaptable Networking. ICT'2003, Assumption University, Thailand, 4/2003.

[2] Anutariya, C., et al., An Equivalent-Transformation-Based XML Rule Language. Int'l Workshop Rule Markup Languages for Business Rules in the Semantic Web, Sardinia, Italy, 6/2002.

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

[4] OWL Service Coalition, Semantic Markup for Web Services, 11/2003.

[5] Shen, J. and Y. Yang, RDF-Based Knowledge Model for Network Management. IFIP/IEEE IM 2003. Colorado, Springs, 3/2003.

[6] Shiaa, M.M., Mobility Support Framework in Adaptable Service Architecture. IEEE/IFIP Net-Con'2003, Muscat, Oman, 10/2003.

[7] Shiaa, M.M., et al., An XML-based Framework for Dynamic Service Management. IFIP INTELLCOMM 2004, Bangkok, Thailand, 11/2004.

[8] Steinfeld, E.F., Devices that play together, work together, *EDN Magazine*, 9/2001.

[9] Strassner, J., DEN-ng: Achieving Business-Driven Network Management. IEEE/IFIP NOMS 2002, Florence, Italy, 4/2002.

[10] Supadulchai, P., Aagesen, F.A., An Approach to Capability and Status Modeling, NIK 2004, Stavanger, Norway, 11/2004.

[11] Westerinen, A. and W. Bumpus, The Continuing Evolution of Distributed Systems Management. *IEICE TRANS. INF & SYST.*, Vol. E86-D Nr. 11: 11/2003.

[12] Wuwongse, V., et al.: XML Declarative Description (XDD), A Language for the Semantic Web. *IEEE Intelligent Systems*, Vol. 16 Nr.3, 5-6/2001.