AUTONOMIC SERVICE CONFIGURATION BY A COMBINED STATE MACHINE AND REASONING ENGINE BASED ACTOR

Paramai Supadulchai and Finn Arve Aagesen
NTNU, Department of Telematics, N-7491 Trondheim, Norway

Abstract: Service systems constituted by service components are considered. Service components are executed as software components in nodes, which are physical processing units such as servers, routers, switches and user terminals. A capability is an inherent property of a node or a user, which defines the ability to do something. Status is a measure for the situation in a system. A service system has defined requirements to capabilities and status. Because of continuous changes in capabilities and status, dynamic service configuration with respect to capabilities and status is needed. Software components are generic components, denoted as actors. An actor is able to download, execute and move functionality, denoted as a role. Configuration is based on the matching between required capability and status of a role and the present executing capabilities and status of nodes. We propose an approach for role specification and execution based on a combination an Extended Finite State Machine and a rule based reasoning engine. Actor execution support consisting of a state machine interpreter and a reasoning engine has been implemented, and has also been applied for a service configuration example.

1. INTRODUCTION

Service systems constituted by service components are considered. Service components are executed as software components in nodes, which are physical processing units such as servers, routers, switches and user terminals such as phones, laptops, PCs, and PDAs. 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* play defined in different *plays*.

To utilize the flexibility 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 *status* and *capability* are introduced.

*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 Service. 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. A capability of a user is a property that makes the user capable of using services. An actor executes a program, which may need capabilities in the node. 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.

The functionality to be played by an actor participating in the constitution of a service is denoted as its role. We use the role-figure as a generic concept for the actor which is playing a role. So services and service components are realized by role-figures. *Service configuration* is here the configuration of services with respect to the required capability and status of the roles.

The Role of an actor is defined in a manuscript, which consists of an EFSM (*Extended Finite State Machine*) extended with rule-based policies. Using a local *rule-based reasoning* engine adds the ability to cope with various situations more flexible than is possible by the pure EFSM. Actors can locally take place in the configuration and reconfiguration of the services, in which they are a part of. The reasoning engine is based an XET (XML Equivalent Transformation) rule-based language.

The work presented in this paper has been related to the Telematics architecture for Play-based Adaptable System (TAPAS) [2]. Section 2 discusses related work. Section 3 presents the model used for the combined EFSM and reasoning engine based actor. Section 4 gives a short presentation of the TAPAS architecture with focus on the elements relevant for the autonomic service configuration. Section 5 presents the data model. Section 6 presents a simple scenario where an actor actively participates in service reconfiguration. Section 7 gives a summary and presents our conclusions.
2. RELATED WORK

The mobility of service components have been dealt with in a number of approaches. An example is the Intelligent Agent, which is the most related to our work. DOSE [4] is an agent-based autonomic platform that uses Semantic Web to come up with response to failures. However, the behavior of each service component must be fixed along with the moving codes that cannot be downloaded or changed. A technique to overcome this shortcoming has been proposed using Java Reflection [5]. The behavior of server components can be downloaded or changed based up on a reasoning mechanism. However, the reasoning mechanism itself cannot be downloaded or altered. In our approach, the behavior of service components and the rules used in the reasoning mechanism are downloadable and can be changed upon needs.

3. THE ACTOR MODEL

The actor role is defined as an Extended Finite State Machine (EFSM) extended with policies. The mechanism interpreting the manuscript is an EFSM interpreter extended with a reasoning mechanism. The data structure applied for the representation of an EFSM is shown in Figure 1. An EFSM contains the EFSM name, initial state, data and variables and a set of states. The state structure defines the name of the state and a set of transition rules for this state. Each transition rule specifies that for each input, the actor will perform a number of actions, and/or send a number of outputs, and then go to the next state. Actions are functions and tasks performed during a specific state: computation on local data, role session initialization, message passing, etc. The structure of the <ACTIONS> list specifies the name, the parameters and the classification of an action.

Rule-based reasoning is considered as a special type of EFSM action that executes policies. Policies are expresses in the XML Equivalent Transformation language (XET) [3]. The reasoning engine can directly operate and reason about XET descriptions.
The XET language 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 denoted as ground XML expressions. Every component of an XML expression can contain variables as shown in Table 1. Every variable is prefixed with ‘Tvar_’ where T denotes its type.

<table>
<thead>
<tr>
<th>Type</th>
<th>Instantiation and examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>XML element or attribute names Ex: &lt;Nvar_X&gt;...&lt;/Nvar_X&gt; can be instantiated to &lt;div&gt;...&lt;/div&gt; or &lt;span&gt;...&lt;/span&gt;</td>
</tr>
<tr>
<td>S</td>
<td>XML string Ex: &lt;a name='Svar_Y'/&gt; can be instantiated into &lt;a name='http://...'/&gt; or &lt;a name='ftp://...'/&gt;</td>
</tr>
<tr>
<td>P</td>
<td>Sequence of zero or more attribute-value pairs Ex: &lt;p Pvar_2='NULL'/&gt; can be instantiated into &lt;p/&gt; or &lt;p style='...'/&gt;</td>
</tr>
<tr>
<td>E</td>
<td>Sequence of zero or more XML expressions Ex: &lt;p&gt;Evar_P&lt;/p&gt; can be instantiated into &lt;p/&gt; or &lt;p&gt;&lt;div&gt;...&lt;/div&gt;&lt;br/&gt;&lt;/p&gt;</td>
</tr>
<tr>
<td>I</td>
<td>Part of XML expressions Ex: &lt;Ivar_X&gt;&lt;hr/&gt;&lt;/IvarX&gt; can be instantiated into &lt;body&gt;&lt;hr/&gt;&lt;/body&gt; or &lt;hr/&gt;</td>
</tr>
</tbody>
</table>

A rule is an XML clause of the form:

\[ H, \{ C_1, ..., C_m \} \rightarrow B_1, ..., B_n \]

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 constraints modeling for a rule. Axioms are defined from one or more rule(s). The XML expression \( H \) is called the head of the clause. The \( B_i \) is a body atom of the clause. When the list of body atom is empty, such a clause is referred to an XML unit clause, and the symbol ‘→’ will be omitted. Hence ordinary XML elements or documents can be mapped directly onto a ground XML unit clause.

The reasoning process begins with an XML expression-based query. An XML clause will be formulated from the query in form:

\[ Q \rightarrow Q \]

XML expression \( Q \) represents the constructor of the expected answer which can be derived if all the body atoms of the clause hold. However, if one or more XML expression body atoms 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 \( R_i \) in the form:

\[ R_i: H, \{ C_i \} \rightarrow B_1, B_2 \]

If the XML structure of the body \( Q \) of the clause and the head \( H \) of the rule \( R_i \) match without violating condition \( C_i \), 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:
\[ Q^* \rightarrow B_1^*, B_2^* \]

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.

4. TAPAS ARCHITECTURE

"Adaptable service systems" are service systems that adapts dynamic to changes in both time and position related to Users, Nodes, Capabilities, Status and Changed Service Requirements. Adaptability can be modeled as a property consisting of 3 property classes: 1) rearrangement flexibility, 2) failure robustness and survivability, and 3) QoS awareness and resource control. The Telematics Architecture for Play-based Adaptable System (TAPAS) intends to meet these properties [2]. In analogy with the TINA architecture [6], the TAPAS architecture is separated into a system management architecture and a computing architecture as follows:

- The system management 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 system management 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.

4.1 Computing architecture

TAPAS computing architecture has three layers: the service view, the play view and the network view as illustrated in Figure 2. For details see [1].

A service system consists of service components and the network system consists of nodes. The play view is the intended basis for designing functionality that can meet the adaptability properties as defined above. The play view is founded on the theater metaphor introduced in Sec.1. TAPAS actors are software components in nodes that can download manuscripts. An actor that does not have a role assigned is denoted as a free actor. An actor playing a role in a manuscript is denoted as a role figure. A service system is
constituted by a play, and leaf service component are constituted by role figures. A role session is the dialog between two executing role figures. A role figure can move between nodes and its role sessions can be re-instantiated automatically. This mechanism, however, is not the focus of this paper. It is referred to [7].

![Figure 2. The TAPAS computing architecture](image)

### 4.2 System management architecture

The main functionality components of the system management architecture are illustrated in the Figure 3. The primary service providing functionality comprises the ordinary services offered to human users.

![Figure 3. The TAPAS system management architecture](image)

In addition, the architecture has two repositories: the Play repository and the capability and status repository and fours management components: Configuration, Service, Capability and status, and Mobility management.

The play repository stores manuscripts and policies, which are the required status and capability of a role as well as local configuration rules. Local configuration rules describe configuration and constraints of a role which must always be maintained. In addition, these rules define policies for handling of reconfiguration related events such as the decision of an actor to move a role when a failure happens. The capability and status repository stores executing capability and status information.

Configuration management makes the initial configuration and reconfigures the service systems when needed. The Service management is responsible for deployment and invocation of services. Capability and status management registers, de-registers, updates and provide access to capability
and status repository and the Mobility management handles the various mobility types.

To fulfill the failure robustness and survivability requirements, the architecture must be dependable and distributed. The proposed actor model creates a distributed configuration management by adding reasoning functionality to actors.

5. DATA MODEL

This section presents XML based approaches to the representation of the elements of the Play repository as well as the Capability and status repository.

5.1 Manuscript

A manuscript consists of EFMSM-based behavior of individual roles. An XML-based EFSM given to an actor is executed by a state machine interpreter. A sample fragment of the XML-base manuscript is shown in Figure 4.

```xml
<state name="ConnectionTimeout">...
<state name="ConnectionLost">
<Transition name="RoleFigureMove">
<input msg="RoleFigureMoveReq source=""></input>
<action class="Reasoning" name="SearchFreeActor">
<param name="role_name" value="role1"></param>
</action>
<output name="Dest_Variable"></output>
<next_state name="PlugoutPending"></next_state>
</Transition>
</state>
```

After the state ConnectionTimeout is visited infinitely often, the actor playing this manuscript will move to ConnectionLost state. If there is an incoming message RoleFigureMoveReq, the actor will execute the RoleFigureMove transition and perform two subsequent actions. The first action uses the built-in reasoning machine to find out a free actor where the role should be moved to. The second action installs the role to a free actor suggested by the first action. At the end of the transition, the actor moves to PlugoutPending state and wait for a plugout message from the newly instantiated role figure.

Figure 4. Fragment of an example XML manuscript showing a transition of state ConnectionLost

SMI interprets the downloaded manuscript. SMI uses action libraries. Policy related actions are platform independent constraints expressed in XET (see Section 2). For non-policy actions, the actions are platform-specific (such as C++) or platform-independent (such as Java) executable codes from the local action library cache to execute the actions in the transition. If the required action libraries cannot be found, SMI will download the actions from an action library database.
5.2 Executing Capability and Status

Nodes possess particular Network View Capabilities and Status, from now on abbreviated as NV-capabilities and -status. They are represented in a network information model such as Common Information Model (CIM) or Universal Plug-and-Play. We have chosen the XML representation of CIM (CIM-XML) to implement our test systems.

Actors have Play View capabilities and status abbreviated as PV-capabilities and -status. The idea is to hide the complexity of the network view. PV-capabilities and -status of an actor are derived from one or more NV-capabilities and -status. PV-capabilities and -status are represented in Resource Definition Framework (RDF) [9], which can be used to either define pointers to NV-capabilities and -status or define derived PV-capabilities and -status from NV-capabilities and -status [8].

5.3 Policies

The policies comprises: role requirements, local configuration rules. These are modeled by the XET language (See Section 3).

5.3.1 Role requirements

Role requirements consist of PV-capabilities and -status required by a role. These PV-capabilities and -status are represented in RDF and XML variables.

5.3.2 Local configuration rules

The heads of the XET clauses identify components of the outcome of the configuration or reconfiguration, while the body describes the configuration, composition and dependency conditions. A sample local configuration rule is illustrated in Fig. 5.
6. DEMONSTRATION

A scenario of a secured database system is considered as an example. A database server contains sensitive information and will automatically blocks incoming requests from nodes that could possibly have malicious software such as viruses or trojans.

The goal of this demonstration is to show how a role figure in a blocked node can survive, move to other one other node, identified as harmless by the database server, and continue working with the database server. How the role figure proves itself as non-malicious software is not the focus and will not be further explained.

Fig. 6 illustrates a role figure RF\textsubscript{1} in Node 1, which is presently blocked by a database server role figure (DB RF). After RF\textsubscript{1} visits a state ConnectionTimeout infinitely often, it will move to ConnectionLost state. At ConnectionLost, RoleFigureMove transition will initiated by a
RoleFigureMoveReq message. The manuscript describing this transition has been presented in Figure 4.

6.1 R1 role requirement (required PV-capabilities and -status)

The PV-capabilities and -status required by the role R₁ are illustrated in Fig. 7. R₁ explicitly needs status Status_FreeActor. The connectivity between R₁ and dbServer must be a member of set {"Secured", "SecuredWireless"}. Actors trying to play R₁ may have other PV-capabilities and -status (as represented by Evar_otherActorProps and Evar_otherConnProps). These PV-capabilities and -status will be ignored by the reasoning engine.

6.2 Offered PV-capabilities and -status

Fig. 8 shows the offered PV-capabilities and -status of actors F₁, F₂ and F₄. The PV-capabilities and -status of F₃ are identical to F₂ while the capabilities and status of F₅ and F₆ are identical to F₄. For lack of space, they will not be presented.
6.3 Query clause

The query clause in Fig. 9 is constructed from a query expression. As already explained in Section 3, the body of the clause will be initially matched with a configuration rule, which will be defined in the Section 5.4.

![Diagram of query clause](image)

*Figure 9. Graphical notation of the query to search for available actors*

6.4 Local configuration rules

Local configuration rules as illustrated in Fig. 10 indicate that the connection status and the connection type of the link between $R_i$ and $DB RF$ must be maintained in a secured manner. The only QoS parameter defined here is $Svar\_connType$.

6.5 The configuration result

The configuration result is shown in Fig 11. Based on the offered PV-capability and -status provided in Fig. 8 and the role requirement defined in Fig. 7, actors $F_2$ and $F_3$ are the most appropriate actors to play $R_i$.

The reasoning process is conducted by Native XML Equivalent Transformation reasoning engine (NxET) implemented as a Java-based action for the state machine interpreter (SMI). NxET is used by SMI to execute SearchFreeActor action defined in Fig. 5. The parameter actorList of the PluginActor action will be substituted with the available actors in Fig. 11. PluginActor will try to move $R_i$ to $F_2$ first. If the moving is not successful, PluginActor will try again with $F_3$. Subsequently, $RF_i$ will move to PlugoutPending state after $R_i$ has been successfully moved to either $R_2$ or $R_3$. At this state, $R_i$ will be plugged out from $RF_i$, which will become a new free actor.
Rule SearchFreeActor will be matched with the body of the query clause defined in Section 5.3. After the matching, the body of the query clause will be re-written with xfn:SetOf, which is the only body atom of the rule. xfn:SetOf will try to construct the list of available actors and add them into Evar_actors variable. Each members of Evar_actors will have the structure similar to the expression in xfn:Constructor. To actually instantiate the possible values for the constructor, the condition expression in xfn:Condition will be matched with other rules (clearly R1Requirement).

R1Requirement queries the R1 role requirement (required PV-capabilities and -status), which have been defined in Section 5.1. The rule again queries actors offering the same PV-capabilities and -status, which R1 requires. The matching between required and offers PV-capabilities and -status are accomplished though the instantiation of variable Evar_properties. The actors queried from the capability and status repository needs Status_Active and Status_FreeActor. The actors can also have other PV-capabilities and -status because they are allowed by the role requirement.

The structure of PV-capabilities and -status of each actor will be matched with xfn:MatchD function so that PV-capability connType with a value Svar_connType can be inspected. Function StringIsMember verifies the instantiated value of Svar_connType to make sure that it is a member of the list "Secure Secured Wireless". Actors that do not offer secured or secured wireless connection will be filtered out. Only qualified one will be selected. R1Requirement can return many answers.

The answers returned by R1Requirement will be aggregated and added to Evar_actors list in the rule SearchFreeActor. The value of Evar_actors will be instantiated to the head of the query clause, which will be the answer of the reasoning process.

7. CONCLUSION

This paper presents an approach to model the behavior of service systems by actors playing roles defined in manuscripts. The actor is a combination of an Extended Finite State Machine (EFSM) and a rule based reasoning engine.

A service system has defined requirements to capabilities and status. Because of continuous changes in capabilities and status, dynamic service configuration with respect to capabilities and status is needed. Configuration is based on the matching between required capability and status of a role and the present executing capabilities and status. Roles are allowed to be moved to increase failure robustness and survivability of a service system. This role
mobility can be achieved through EFMS behavior. However, using a rule-based reasoning mechanism allows actors to use local configuration rules to take decisions based on the current executing capabilities and status. The actor model improves actor functionality, increases survivability and makes the configuration management distributed.

Generic actor execution support consisting of a state machine interpreter and a reasoning engine has been implemented and applied for the presented example. All capability and status related data as well as actor behavior is based on XML representations, with exceptions of the EFMS actions. Normal EFMS actions are platform-specific (such as C++) or platform-independent (such as Java) executable codes while reasoning-based EFMS actions are XML-based. The reasoning engine is based on Native XML Equivalent Transformation.

REFERENCES