FORMAL MODELLING OF AN ADAPTABLE SERVICE SYSTEM

Mazen Malek Shiaa, Finn Arve Aagesen, and Cyril Carrez
NTNU, Department of Telematics, N-7491 Trondheim, Norway
{malek, finnarve, carrez}@item.ntnu.no

Abstract: Adaptable service systems are service systems that adapt dynamically to changes in both time and position related to users, nodes, capabilities, status, and changed service requirements. We present a formal model for the basic entity used for the implementation of the service functionality in the Telematics Architecture for Play-based Adaptable Service systems (TAPAS). This basic entity is the *role-figure*, which executes in the nodes of the network. The formal model is denoted as the *role-figure model.* It comprises behaviour specification, interfaces, capabilities, queue of messages, and executing methods for role-figures. Its semantics is based on an ODP (Open Distributed Processing) semantic model and rewriting logic, and is used to prove properties such as: *plug ability*, *consumption ability*, and *play ability.*

1. INTRODUCTION

Service systems consisting of services realized by service components are considered. Service components are executed as software components in network nodes and terminals. A terminal is a node operated by a human user. Those generic components are denoted as *actors*. This name comes from the analogy with the actor in the theatre, where an *actor* plays a *role* in a *play* defined by a *manuscript.* We use *role-figure* as a generic concept for the *actor* which is playing a *role.* So services and service components are constituted by *role-figures*. The attributes of services, service components and nodes are formalised, stored and made available using the concepts of *status* and *capability. Status* is a measure for the situation in a system concerning the number of active entities, traffic, and Quality of Service (QoS). *Capability* is the properties of a node or a user defining the ability to do something.

Telematics Architecture for Play-based Adaptable Service systems (TAPAS) is a research project which aims at developing an *architecture* for adaptable service systems. *Adaptable* means that the service systems will adapt dynamically to changes in both time and position related to users, nodes, capabilities, status, and changed service requirements. In TAPAS, adaptability is modelled as a 3-classes property: *1)* Rearrangement flexibility, *2)* Failure robustness and survivability, and *3)* QoS awareness and resource control [1,2,3]. One objective is to gain experiences by implementing those various features. Parts of the specified functionality have been implemented based on java and web services platforms. The TAPAS architecture has been specified using various UML diagrams.

However, it has been realized that the behaviour parts of the architecture lacks a formal foundation. The implementation software only contains program code, while the UML diagrams only specify parts of the functionality informally. We need a model that can be used as a basis for the formal verification of the various issues related to adaptability. Mainly, this means that when a service is trying to adapt to a change in the service system, it will change some of its composing parts (for example by moving or creating new service components). We would like the formal model to ensure that the actions taken by the service will achieve its goal, and without harming the whole architecture. In this paper we present a formal model of the main component of the TAPAS architecture. This component is the role-figure and the formal model is denoted as the *role-figure model.* The model will be used to verify the behaviour of the role-figures, and will be the basis for the formal verification of certain properties of the system.

Related works and TAPAS are discussed in Sec. 2 and 3, respectively. The semantics of the role-figure model is presented in Sec. 4, while its properties are discussed in Sec. 5. Section 6 concludes the paper.

2. RELATED WORK

The role-figure model must capture the features and properties of service adaptability. Various formal frameworks have been considered as candidates. Process Algebras such as π-calculus [4] have very powerful notations which abstract system elements in terms of processes and communication channels, focusing on the sequence of inputs/outputs. Specifying the TAPAS architecture and role-figures using process algebras is possible, but the specification would be very detailed and lengthy. Moreover, we are more concerned with the constructive states of the system than the input/output sequences.

The ODP framework and the ODP formal model presented by Najm and Stefani in [5], and further elaborated with Dustzadeh [6] is also very interesting. ODP computational objects have states and can interact with their environment through operations on interfaces. The object interfaces and operations provide an abstract view of the state of the object. Access to the object is only possible through invocations of its advertised operations on a designated interface. ODP computational objects and role-figures have several similarities, e.g. the definition of interfaces and their dynamic creation, as well as the distributed operation invocation.

The ODP formal model was based on rewriting logic theory [7]. The semantics of the Rewriting Logic is based on the models of rewrite systems: it is applied to terms which are rewritten based on rewriting rules of the form $t \rightarrow t'$ (meaning the term t is rewritten to t'). This theory has been used for the formal specification and verification of many other systems, such as the Actor semantics [8], and the formalization of active network [9,10].

Our role-figure model is based on the ODP formal semantics, and Rewriting Logic.

3. TAPAS ARCHITECTURE

In accordance with TINA architectural framework, TAPAS is separated into two parts: the *computing* architecture and the *system management* architecture. The *computing* architecture is a generic architecture for the modelling of any service system. The *system management* architecture, not detailed in this paper, is the structure of services and management components.

The computing architecture has three views: the *service* view, the *play* view, and the *network* view (*Figure 1*). The *service* view concepts are generic and should be consistent with any service architecture. Basically, a service system consists of several service components.

The *play* view concepts are the basis for implementing the service view concepts. The concepts of actor, role, role figure, manuscript, capability and status have already been defined. Additional concepts are *director*, *role-session* and *domains*. The *director* acts according to a special role and manages the performance of different role-figures involved in a certain *play*. It also represents a *play domain*. *Role-session* is the projection of the behaviour of a role-figure with respect to one of its interacting role-figures.

The *network* view concepts are the basis for implementing the play view concepts. In the network view, *capability* is provided by a *node* or is owned by a *user*. *User*, *node*, and *capability* have *status* information. A *play domain* may be related to one or more *network domain* (a set of *nodes*), as a play may be distributed over several network domains.

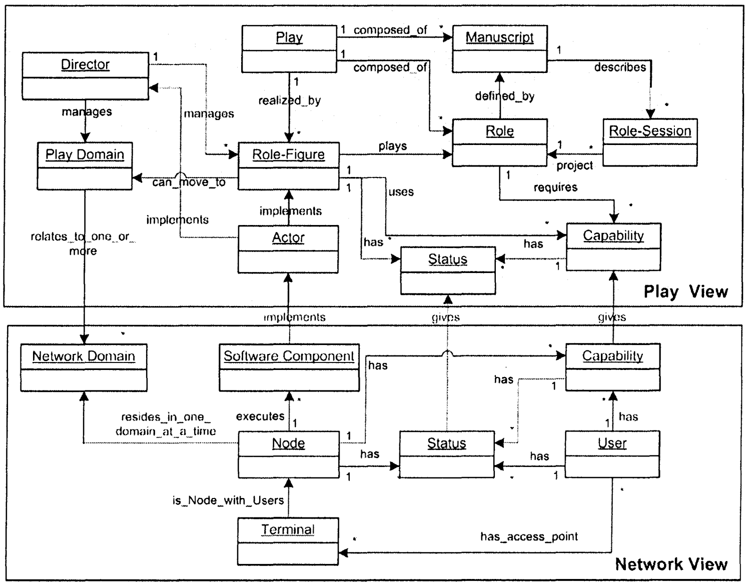

Figure 1. Computing architecture – Play View and Network View

The play view intends to be a basis for designing functionality that can meet the requirements related to rearrangement flexibility, failure robustness and survivability, and QoS awareness and resource control. The play view concepts allow service components to be instantiated according to the available capabilities and the status in the network. They also facilitate the handling of dynamic changes in the installed service components, which occur due to changing capabilities, changing functionality, changing locations, etc.

An important concept related to the *role-session* is the *interface.* Two role-figures can only communicate if they are connected via interfaces. A role-figure creates an interface locally and connects it to another interface in another role-figure. Sending a message is performed by the local interfaces of a role-figure. TAPAS core platform is a platform supporting the functionality of the play view by offering a set of methods. Role-figures will also interact with each other via *signals* that are used to interact with the behaviour of the role-figure, and thus performing the service.

The role-figure model is a formal model of the role-figure implementation. The following aspects need to be included:

- Role-figures are realized by actors and can be dynamically created;
- Role-figures interact via role-sessions and are connected via interfaces. Messages are asynchronous;

- Role-figures comprise behaviour (an extended finite state machine), and methods used for management and control of actor objects;
- Messages are: signals (used to interact with the role-figure behaviour), requests to invoke methods, and returns (results of the invoked methods);
- The main role-figure methods are:

– *PlugInActor*	instantiates role-figures
– *PlugOutActor*	terminates role-figures
– *CreateInterface*	creates interfaces in role-figures
– *BehaviourChange*	changes role-figure behaviour
– *CapabilityChange*	adds or modifies capabilities
– *RoleFigureMove*	moves role-figure to new locations.

The *RoleFigureMove* procedure is used to implement the role-figure mobility. We believe this mobility is one of the keys to adaptability. To ensure that the moving role-figure will continue its execution after the movement, the following parts of the role-figure will be moved as well:

- *behaviour* described by a specification
- *role-sessions* and *interfaces* with other role-figures
- consumed *capabilities* in the node
- *queue* of incoming messages
- executing *methods* (or the role-figure active tasks)

In this paper we only handle the first three parts: behaviour, interfaces, and capabilities. Role-figure mobility management is further detailed in [11].

4. THE ROLE-FIGURE MODEL

This section presents the semantics of the role-figure model. These are semantic rules defining the structure and the behaviour of role-figures. These rules are inspired by the semantics of the ODP computational model [5], based on the rewriting logic [7]. We will use the notations:

$\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{f}, \boldsymbol{g}, \boldsymbol{h}$	role-figure names;
A,A′,... B,B′,...	role-figures $\boldsymbol{a}$ and $\boldsymbol{b}$ as they evolve, respectively;
i: α, j: α'	interface names i and j, with their types α and α';
$r=\langle w_1=v_1, w_2=v_2\rangle$	record with fields w_1 and w_2, having values v_1 and v_2; $r.w_1$ will be used to access the value of w_1 in r;
$\|\|$	asynchronous parallel operator;
$\lhd$	insert operator; "a$\lhd$b" only executes if a is not in b.
$\rhd$	remove operator. "a$\rhd$b" only executes if a is in b.

The operators $||$, $\lhd$ and $\rhd$ are associative and commutative, with $\varnothing$ as neutral element.

4.1 Role-figure components

The semantics for the role-figure model is based on a *Role-Figure Configuration (RFC)*, which is a set of role-figures interacting asynchronously:

$$
\begin{aligned}
&\text{RFC} && ::= \varnothing \mid \text{RFCE} \mid \text{RFC} \parallel \text{RFC} \\
&\text{RFCE} && ::= \text{RF} \mid \text{MSG} \\
&\text{RF} && ::= \langle \text{Name} = \text{string}, \text{Int} = \gamma, \text{Beh} = \beta, \text{Cap} = \pi \rangle \\
&\text{MSG} && ::= \text{Req} \mid \text{Sig} \mid \text{Ret}
\end{aligned}
$$

A role-figure configuration *RFC* is composed of parallel RFC Elements (*RFCE)*, which is either a role-figure *RF*, or a message *MSG.* Three kinds of message exist: a method invocation request *Req*, a communicating signal *Sig*, and a method return result *Ret.* A role-figure *RF* has a name (*Name*) and is defined by a set of interfaces (*Int*), a behaviour (*Beh*), and a set of capabilities (*Cap*). These parts (except *Name*) may evolve as the role-figure consumes messages. Role-figure names are used to distinguish different role-figures; however, as a simplification, we will omit this name in the rest of the article and assume $A, A', \ldots$ always stand for role-figure ***a***. The definitions of the role-figure are the following:

Interface	γ	$::= \varnothing \mid \gamma \lhd [j{:}\alpha] \mid \gamma \rhd [j{:}\alpha]$
Behaviour	β	
Capabilities	π	$::= \varnothing \mid \pi \lhd [c{:}cn] \mid \pi \rhd [c{:}cn]$
Invocation req.	Req	$::= \langle tar = j{:}\alpha,\ src = a,\ met = m{:}mn,\ ret = r,\ arg = p \rangle$
Signal	Sig	$::= \langle tar = j{:}\alpha,\ src = a,\ name = sig,\ arg = p \rangle$
Return	Ret	$::= \langle tar = j{:}\alpha,\ src = a,\ arg = p \rangle$
Argument list	p	$::= (p_1{:}t_1, \ldots, p_n{:}t_n)$

Where:

γ	list of interface definition $[j{:}\alpha]$ where j is an interface reference of type α. Types of interfaces are discussed in the next paragraph;
β	behaviour, based on an EFSM specification (see next paragraph);
π	list of capabilities $[c{:}cn]$. cn is a name denoting the type of capability. c denotes the capability identifier which is an instance or value of cn.
Req	method invocation request sent by the role-figure *src* to the target interface *tar,* invoking method *met* with arguments *arg* and return *ret.*
Sig	signal called *name*, sent by the role-figure *src* to the target interface *tar* with argument list *arg*.
Ret	return from a method invocation sent by the source role-figure *src* to the target interface *tar*, with argument list *arg*.
p	argument list of parameters $p_1, \ldots, p_n$ with types $t_1, \ldots, t_n$, respectively.

Interface types are defined as follow:

$\alpha \quad ::= \langle m_1{:}\ methsig, \ldots, m_n{:}\ methsig, sig_1, \ldots, sig_k \rangle$

$methsig ::= Nil \mid p \rightarrow return$ with p an argument list as defined earlier

Where:

$m_1, .., m_n$	Method names;
methsig	Method signature with arguments *argument* and a return *return*;
$Sig_1,..,sig_k$	signals with a name and arguments (i.e. like *Sig*, without *src, tar*)

The behaviour definition, β, is based on the operational semantics of the state machine model. We added to this EFSM the semantic the notion of stable states, which are states where a behaviour change is allowed:

$$\beta ::= \langle B = b : behaviour,\ St = st : state,\ Sg : g,\ Sc : s,\ Ss : s\rangle$$

s ::= $(state_1, ..., state_f)$ state names

g ::= $(sig, ..., sig_f)$ signal names

Where:

B	EFSM behaviour specification containing the state transition rules: triggering events, tasks performed, signals sent, and next states
St	current state
Sg	set of input signals (trigger events for state transition at current state)
Sc	set of successor states (next states after the firing of input signals)
Ss	Set of stable states (states where behaviour change is permitted)

As a role-figure behaviour evolves and transits from one state to another, *St, Sg, Sc,* and *Ss* change and reflect the status of the role-figure behaviour.

4.2 Behaviour evolution

This section describes the set of rewriting rules that handle the behaviour of a role-figure. The general form of the rewriting rules for role-figure ***a*** is the following[1]:

$$\text{l: } A \parallel T \parallel \Theta \parallel M \longrightarrow A' \parallel \Sigma \parallel T' \parallel \Theta' \parallel M' \quad \text{if } C$$

Where:

l is a label. *A* and *A'* stand for role-figure ***a*** that evolves from *A* to *A'*. Σ is the role-figures created in this rewriting rule (e.g. Σ can be *B* meaning that role-figure ***b*** was created). *T* and *T'* are *return* sets, Θ and Θ' are *signal* sets, *M* and *M'* are *request* sets. *C* is a condition.

This general rewriting rule, inspired from [5], is used to handle the transitions of any role-figure configuration. As such, a number of role-figures and messages (signals, requests and returns) can come together and participate in a transition in which some new role-figures and new messages may be created. Some restrictions apply:

- messages (*returns, signals*, and *requests*) are all consumed in a transition:
$$T \cap T' = \Theta \cap \Theta' = M \cap M' = \emptyset$$
- created role-figures are unique: $B \in \Sigma$ implies ***b*** is unique

[1] Recall that *A* and *B* define the role-figure elements RF whose names are ***a*** and ***b***.

- created messages have their *src* field set to the role-figure that sent them, and *tar* is connected to an interface of an existing role-figure:
  If $msg \in M' \cup T' \cup \Theta'$ with $msg.tar = [i{:}\alpha]$, then:
  $msg.src = \boldsymbol{a} \ \wedge\ [i{:}\alpha] \in A.Int$
  $\wedge\ \exists\ \boldsymbol{b} \in RoleFigures$[2], $j{:}\alpha \in B.Int$ such that connected(i,j)=TRUE
- Messages must be received by the proper interface indicated in *tar*:
  If $msg \in M \cup T \cup \Theta$ with $msg.tar = [i{:}\alpha]$, then:
  $\exists\ [j{:}\alpha] \in A.Int$ with connected(i,j)=TRUE

The predicate connected(i,j) checks that interfaces i, j are interconnected. This issue is left opened so no restriction is made on future implementations (for example, i can be made of the addresses of the local interface and the distant one j).

The rewriting rules will handle behaviour evolution, communications and adaptability functionality. From now on, the role-figure parts will remain constant when applying the rules unless mentioned otherwise.

The following set of rules handle behaviour evolution (internal_action) and communication between role-figures:

internal_action: $A \rightarrow A'$
with: $A.Cap \subseteq A'.Cap \ \wedge\ A'.Beh.St \in A.Beh.Sc \cup \{A.Beh.St\}$

send_request: $A \rightarrow A' \,\|\, req$
Assume $req = \langle\, tar = i:\alpha, src = a, met = m: mn, ret = r, arg = \tilde{p} \,\rangle$:
mn: $args_m \rightarrow return_m \in \alpha$ and $r = return_m \wedge \tilde{p} = args_m$

recv_request: $A \,\|\, req \rightarrow A'$
Assume $ret = \langle\, tar = i:\alpha, src = a, ret = r, arg = \tilde{p} \,\rangle$:
mn: $args_m \rightarrow return_m \in \alpha$ and $r = return_m \wedge \tilde{p} = args_m$

send_return: $A \rightarrow A' \,\|\, ret$
recv_return: $A \,\|\, ret \rightarrow A'$
send_signal: $A \rightarrow A' \,\|\, sig$
Assume $sig = \langle\, tar = i:\alpha, src = a, name = sig, arg = \tilde{p} \,\rangle$:
$\exists\ sig_k \in \alpha$ such that $sig_k.name = sig$

recv_signal: $A \,\|\, sig \rightarrow A'$
$sig \in A.Beh.Sg \Rightarrow A'.Beh.St \in A.Beh.Sc$

Explanation of the rules is the following.
internal_action: the role-figure can change its capabilities and perform a state transition.
send_request: a role-figure may invoke a method in another role-figure by sending a method invocation request via the appropriate interface. The

[2] *RoleFigures* denotes all the role figure names in the configuration.

method signature must be declared in the type α of the target interface, and the arguments and return set in the request must match this signature.
recv_request: when receiving a request, the method signature must be declared in the type α of the interface, and sent arguments and return type must match this signature.
send_return, **recv_return:** when sending or receiving returns, there is no additional restrictions: only basic type compatibility check is made[4].
send_signal: a role-figure may send a signal to a role-figure due to service functionality. The signal must be declared in the type of the target interface.
recv_signal: receiving a signal will trigger a state transition.

Note that a state transition is allowed only during an internal action or when receiving a signal.

Adaptability functionality is dealt with six special requests: plug in *pi*, plug out *po*, create interface *ci*, behaviour change *bc*, capability change *cc*, and role-figure move *mo*. The corresponding rewriting rules are specialisations of **send_request** and **recv_request**, with specific constraints:

Role-figure Plug in: $A \parallel pi \rightarrow A' \parallel \boldsymbol{b}$ $\quad pi.arg ::= (name, loc, beh{:}\beta, cap{:}\pi)$
$A.Int \subseteq A'.Int \wedge \boldsymbol{b} = pi.arg.name \wedge \text{location}(\boldsymbol{b}) = pi.arg.loc$
$\wedge\ B.Beh = pi.arg.beh \wedge pi.arg.cap \subseteq B.Cap$

Role-figure Plugout: $A \parallel po \rightarrow \emptyset$ $\quad po.arg ::= (name)$
$\forall B,\ A.Int \cap B.Int \neq \emptyset{:}\ B \rightarrow B'$ with $B'.Int = B.Int - A.Int$

Create Interface: $A \parallel ci \rightarrow A'$ $\quad ci.arg ::= (j_1 : \alpha_1, \ldots, j_n : \alpha_n)$
$A'.Int = A.Int \lhd_{i=1}^{n} ci.arg.j_i$

Behaviour Change: $A \parallel bc \rightarrow A'$ $\quad bc.arg ::= (beh : \beta, cSt{:}\ State)$
$A.Beh.St \in A.Beh.Ss \Rightarrow A'.Beh.B = bc.arg.beh \wedge A'.Beh.St = bc.arg.cSt$

Capability Change: $A \parallel cc \rightarrow A'$ $\quad cc.arg ::= (p_1{:}\ c_1, \ldots, p_n{:}\ c_n)$
$A'.Cap = A.Cap \lhd_{i=1}^{n} cc.arg.j_i$

Role Figure move: $A \parallel mo \rightarrow A'$ $\quad mo.arg ::= (loc)$
$A'.Int \subseteq A.Int \wedge A'.Cap \subseteq A.Cap \wedge \text{location}(\boldsymbol{a'}) = mo.arg.loc$

Role-figure Plug in: this method plugs in a new role-figure named *name* at location *loc*. The created role-figure ***b*** will also receive its behaviour *beh* and capabilities π. Its interfaces will be added to *A*, the role-figure that received the request. We hide the complex process of director play management, capability allocation, etc. and describe it by a single rewriting rule.

[4] Concerning the sending, the arguments are not checked because they have already been matched by the method invocation request semantics.

Role-figure Plug out: a role-figure which receives this request disappears. All references to its interfaces are removed with additional rewriting rules.
Create Interface: all the interfaces passed as arguments of this *ci* request will be added to the role-figure's interface definition, *Int.* Interface creation between two role-figures means that they will agree on the terms and conditions of their future interactions.
Behaviour Change: a behaviour change assigns a new behaviour to the role-figure, with a current state. It is allowed only in stable states A.*Beh.Ss*.
Capability Change: this request changes the capability definition of the role-figure. Capabilities specified in the *cc* request are added to the role-figure's capability set, *Cap.*
Role-figure Move: this request moves a role-figure from one location to another. It is equivalent to a sequence of *pi, bc ci, cc,* and *po* requests: a role-figure is plugged in at the new location *loc*, with the behaviour, interfaces and capabilities of the original role-figure. The role-figure instance at the original location is terminated by a *po* method.

5. ROLE-FIGURE PROPERTIES

In this section we introduce requirements on role-figure configuration, and define properties to verify their correctness. This verification process takes place at the service system design phase to improve the service system at early design phases by identifying design errors.

The role-figure configuration, $rfc = \{a_1, \ldots, a_M, g_1 \ldots, g_N\}$, evolves through $rfc \rightarrow rfc^1 \rightarrow \ldots\ rfc^q \rightarrow \ldots$. In every transition a role-figure a_i evolves through A_i by either consuming a message g_j, generating a new message, or performing an internal action. Also, every interface in any of the role-figures is connected to another interface in another role-figure. The role-figure configuration is considered ***well-formed*** if and only if it obeys the rules and conditions constructed in the role-figure semantics. This role-figure configuration has three properties: *Plug ability*, *Consumption ability*, and *Play ability.*

Plug ability. This property proves that a role-figure has been plugged in at certain location. We do so by ensuring that the consumption of a plug in request has achieved the plug in of a role-figure at the appropriate location. The required capabilities and behaviour of the created role-figure must also satisfy the requirements of the plug in request. This property is defined by:

$$\boldsymbol{P}_{plug\ ability} = \quad \forall\ rfc = \{\mathbf{a}_1, \ldots, \mathbf{a}_M, \mathbf{g}_1, \cdots, \mathbf{g}_N, \mathbf{g}_{plugin}\},\ \mathbf{g}_{plugin} = pi(\mathbf{a}_{new}, loc_i, beh_i, capset_i),$$

$$rfc \xrightarrow{g_{plugin}} rfc' \parallel \mathbf{a}_{new}$$

$$\text{where} \begin{cases} rfc' = \{\mathbf{a}_1, \ldots, \mathbf{a}_M, \mathbf{g}_1, \cdots, \mathbf{g}_N\} \\ \mathbf{A}_{\mathbf{a}_{new}} = <\mathbf{Beh} = <\mathbf{B} = pi.beh_i>, \mathbf{Cap} = pi.capset_i>; \mathbf{location}(\mathbf{a}_{new}) = pi.loc_i \end{cases}$$

Consumption ability. This property proves that all messages generated by the role-figures during their execution will eventually be consumed:

$$P_{consumption\,ability} = \quad \forall\, rfc = \{\mathbf{a}_1,\ldots,\mathbf{a}_M,\mathbf{g}\}, \qquad rfc \xrightarrow{\mathbf{g}} rfc^1 \xrightarrow{\mathbf{g}_{rfc^1}} \cdots rfc^Q \xrightarrow{\mathbf{g}_{rfc^Q}} rfc_{\text{terminal}}$$
$$\text{such that} \begin{cases} \forall rfc^i : \quad rfc^i = \{\mathbf{a}_1,\ldots,\mathbf{a}_N,\mathbf{g}_1,\ldots,\mathbf{g}_P\}, & 1 < i < Q \\ rfc_{\text{terminal}} = \{\mathbf{a}_1,\ldots,\mathbf{a}_O\} \end{cases}$$

The configuration consumes messages and evolves based on the actions that will occur after the consumption: messages may be generated that will eventually be consumed. This process terminates when there will be no messages in the configuration (note the number of role-figures O in $rfc_{terminal}$ is different from M in rfc). This property examines all terminal states of a configuration and checks if they contain any unconsumed message (terminal states are states of the where no rewriting rule could be applied any further).

Play ability. This property proves that a role-figure, after its plug in phase, is playing or performing according to its predefined role. We have to verify that the role-figure behaviour is progressing, e.g. by marking certain states where something desirable happens as progress states, and examine if an execution of the system reaches such states. In play ability we only consider messages that are signals. There can be two types of this property: weak and strong Play ability. Weak Play ability proves that a role-figure has begun performing once it has been plugged in: at least one of the input signal $\boldsymbol{g}_k$ of the role-figure has been consumed. Weak Playability is defined by:

$$P_{wplay\,ability} = \quad \forall\, rfc = \{\mathbf{a}_1,\ldots,\mathbf{a}_i,\ldots,\mathbf{a}_M,\mathbf{g}_1,\ldots,\mathbf{g}_N\}, \qquad rfc \longrightarrow \cdots rfc' \xrightarrow{\mathbf{g}_k} rfc''$$
$$\text{such that} \begin{cases} rfc' = \{\mathbf{a}_1,\ldots,\mathbf{a}_i,\ldots,\mathbf{a}_O,\mathbf{g}_1,\ldots,\mathbf{g}_k,\ldots,\mathbf{g}_P\}, \quad \mathbf{g}_k \in \mathbf{A'}_i.Beh_i.Sg_i \\ rfc'' = \{\mathbf{a}_1,\ldots,\mathbf{a}_i,\ldots,\mathbf{a}_O,\mathbf{g}_1,\ldots,\mathbf{g}_{k-1},\mathbf{g}_{k+1},\ldots,\mathbf{g}_P\} \end{cases}$$

Strong play ability requires that a role-figure is proved to be free of non-progress cycles (a progression is achieved):

$$P_{splay\,ability} = \quad \forall\, rfc = \{\mathbf{a}_1,\ldots\mathbf{a}_i\ldots,\mathbf{a}_M,\mathbf{g}_1,\ldots,\mathbf{g}_N\} \quad rfc \longrightarrow \cdots rfc^q \xrightarrow{\mathbf{g}_{rfc^q}} \cdots rfc^{Q-1} \xrightarrow{\mathbf{g}_{rfc^{Q-1}}} rfc^Q$$
$$\text{such that} \begin{cases} rfc^q = \{\mathbf{a}_1,\ldots\mathbf{a}_i\ldots,\mathbf{a}_O,\mathbf{g}_1,\ldots,\mathbf{g}_P\}, & 1 \le q \le Q \\ \mathbf{A}_i \longrightarrow \mathbf{A}_i^1 \longrightarrow \cdots \mathbf{A}_i^q \rightarrow \cdots \mathbf{A}_i^Q, \\ \exists st_i \in \{\mathbf{A}_i^1.Beh_i.St_i,\cdots,\mathbf{A}_i^Q.Beh_i.St_i\}, \quad st_i \in \mathbf{a}_i|_{Beh.Progress} \end{cases}$$

The strong play ability requires knowledge of the role-figure state, which cannot be obtained by an external observation, as well as it requires knowledge of whether a state in the behaviour specification is a progress state or not. This property shows that a role-figure, which is assumed existing throughout a given execution of a configuration, evolves. Furthermore, the behaviour of the role-figure is said to have progressed at least once – one of its current states has been a progress state. The only difference to the weak play ability is the denotation, $\boldsymbol{a}_i|_{Beh.Progress}$, which stand for the progress states in the role-figure behaviour.

6. CONCLUSION

A formal model for the role-figures in the TAPAS architecture has been presented. Rewriting rules were used to describe role-figure behaviour as well as the three properties: *plug ability*, *consumption ability*, and *play ability*.

The plug ability property proves that a role-figure has been plugged in at certain location. The consumption ability property proves that all messages generated by the role-figures during their execution will eventually be consumed. The play ability property proves that a role-figure, after its plug in phase, is playing (i.e. its behaviour is progressing).

Although our experiences with modelling the role-figure and its properties are quite encouraging, the model we presented is just a first and preliminary step. The semantics and the dynamics of the role-figure model would benefit a more elaborated interface type theory: the behavioural types of Carrez et al.[12] can be used to describe the messages that are exchanged at the role-figure interfaces. Finally, the properties of the role-figure model may be extended to elaborate on the role-figure mobility management presented in [11], and used to verify the validity of mobility strategies (i.e. when and how to move).

References

1. F. A. Aagesen, B. E. Helvik, V. Wuvongse, H. Meling, R. Bræk, and U. Johansen, Towards a plug and play architecture for telecommunications, in *SmartNet'99* (1999)
2. F. A. Aagesen, B. E. Helvik, U. Johansen, and H. Meling, Plug&play for telecommunication Functionality: architecture and demonstration issues, in *IConIT'01* (May 2001)
3. F. A. Aagesen, B. E. Helvik, C. Anutariya, and M. M. Shiaa, On adaptable networking, in *ICT 2003* (April 2003)
4. R. Milner, J. Parrow, and D Walker, A calculus of mobile processes (parts I and II), in *Information and Computation*, 100:1-77 (1992)
5. E. Najm and J.B. Sstefani, A formal semantics for the ODP formal model, in *Computer Networks and ISDN systems 27*, pp.1305-1329 (1995)
6. J. Dustzadeh and E. Najm, Consistent semantics for ODP information and computational models, in *Proceedings of FORTE/PSTV'97* (Osaka, Japan, November 97)
7. N. Marti-Oliet and J. Meseguer, Rewriting logic as a logical and semantic framework, SRI International, Computer Science Laboratory Technical Report, August 1993
8. C. L. Talcott, An actor rewriting theory", in *ETCS*, 4 (1996)
9. G. Denker, J. Meseguer, and C. Talcote, Formal specification and analysis of active networks and communication protocols, in *DISCEX'2000* (January 2000)
10. B. Wang, J. Meseguer, and C. Gunter, Specification and formal analysis of PLAN algorithm in Maude, in *Workshop on Distributed system validation and verification*, (2000)
11. M. M. Shiaa, Mobility support framework in adaptable service architecture, in *Net-Con'2003* (Muscat Oman, October 2003)
12. C. Carrez, A. Fantechi, and E. Najm, Behavioural contracts for a sound assembly of components, in *Proc. of FORTE 2003, LNCS 2767* (Berlin, Germany, September 2003)