Mazen Malek Shiaa

Mobility Management in Adaptable Service Systems

Doctoral thesis
for the degree of doktor ingeniør

Trondheim, June 2005

Norwegian University of Science and Technology
Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Telematics

NTNU
Innovation and Creativity

NTNU
Norwegian University of Science and Technology
Doctoral thesis
for the degree of doktor ingeniør
Faculty of Information Technology, Mathematics and
Electrical Engineering
Department of Telematics

ISBN 82-471-7156-2 (printed ver.)
ISBN 82-471-7155-4 (electronic ver.)
ISSN 1503-8181

Doctoral theses at NTNU, 2005:135

Printed by NTNU-trykk

Abstract

Telecommunication service systems have been developing rapidly during the last five decades. The service architectures as well as the technologies for design, implementation, deployment, execution, and management of the services have been under continuous development. The focus of this thesis is mobility management in adaptable service systems. *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 and *mobility management* is the handling of movements of the various components that can potentially move. As examples persons, services, terminals, nodes, capabilities, data and programs can move. Mobility management allows services to find locations, and to deliver certain content to the users or terminals regardless of their location. This thesis is focusing on the movement of persons, services, programs and terminals.

The thesis is related to TAPAS (Telematics Architecture for Play-based Adaptable Service Systems) research project. This project started in 1997 and has been founded by the Norwegian Research Council and the Department of Telematics at NTNU.

The thesis has four main parts: 1) a generic terminology framework, 2) a mobility management architecture, 3) a design model for the basic mechanism used to specify and realize the services, i.e. the role-figure model, and 4) a formal model and analysis of the role-figure model.

The terminology framework is the basis for the mobility management architecture. Three main mobility types are handled. These types are personal mobility, role-figure mobility, and terminal mobility. For each of these mobility types a set of generic concepts, definitions, and requirement rules are presented.

The mobility management architecture defines the structure and the functionality of the entities needed to handle the various mobility types. The mobility management architecture is worked out within the context of TAPAS.

The role-figure model is an abstract model for the implemented role-figure functionality. It has parts such as behaviour, capabilities, interfaces, messages, and executing methods. By using an ODP (Open Distributed

Processing) semantic framework and the rewriting logic, the structure of the cooperating role-figures and their behaviour is defined.

This model will be used as the basis for a formal model specified in Maude, which is a language and tool supporting specification and analysis of rewriting logic theories. It is used to reason about the structure and the behaviour of the role-figures and the proposed solution for role-figure mobility.

Acknowledgement

The work presented in this thesis has been done at the Department of Telematics, Norwegian University of Science and Technology, where Prof. Finn Arve Aagesen was my supervisor.

I am truly grateful to Finn Arve for the scientific guidance he assisted me with throughout the entire period of my PhD studies. He has always been a support, both professionally and spiritually, whenever I faced difficulties. I am thankful for the inspirations he used to give me to conduct my research. Our intensive discussions during the last several months have been really comprehensive and exhaustive, sometimes painful. However, he guided me in a most friendly, competent, and patient manner. He, beyond doubt, taught me how to be comprehensive, consistent, and, most importantly, efficient in my work. So thank you Finn Arve.

I am equally grateful and thankful to my soul mate, the most wonderful human being in my heart, my lovely wife. Erika has always been there for me. The amount of support and backing she gave me over the years was always needed. She never let me down, and always made me feel home even when I am thousands of miles away from my homeland, family, and friends.

I am greatly indebted to my colleague Cyril Carrez. Cyril has really been a great help. His patient readings of my thesis, and his valuable comments were always supportive. I would also thank Paramai Supadulchai. Discussions with Paramai provided me with needed feedbacks, and he really was a nice roommate during the last two years.

Also, I would thank Shanshan and Humberto for their comments on my early draft versions of this thesis. Equally, I would like to thank Randi for her administrative help and advices, not only regarding my PhD studies and teaching assistance work, but also for her kind help during my early days here in Trondheim. Thanks also to all my colleagues at the department with whom I shared wonderful and productive five years of my life.

The thesis is dedicated to my parents, whose love was always of the best. To the memory of my father who always dreamed about the moment I get this far, and my mother who has always been praying for me. Also to my older sister Maysoon, my brother Muhannad, and my little sister Maali, all have been supporting me with their wishes and thoughts. They

have been the fuel that got me started and kept me going to the finish line, although they all live far away from me in three different countries in three different continents.

Contents

List of Figures

CHAPTER 1

Introduction

1.1 Background

A TELECOMMUNICATION SERVICE is a functionality offered to a service-user by a service-provider. A service user can be a human user, software functionality, or hardware functionality. A service provider can be an enterprise offering services, but it can also be software functionality or hardware functionality. Services offered to a human user are denoted as the user services, and an enterprise offering services to a user is denoted as a telecommunication service provider.

During the last five decades the service architectures as well as the technology for design, implementation, deployment, execution, and management of the services have been under continuous development. But it is also important to notice that the relational structure of business actors involved in telecommunication service provision has changed dramatically. In addition to telecommunication service providers we have actors such as network providers, service brokers, service creators, and service subscribers. A service subscriber can order a service from a service creator that is offered by a telecommunication service provider. A telecommunication service provider can use the service of a network provider for the connectivity between users, subscribers, and service creators. Telecommunication services are already an important infrastructure for the society. However, the business as well as the technology landscape is more complex than ever.

In the 60’s the main user service was the fixed line telephony service. This service was only concerned with the establishment of connections between two users. In the 70’s the Signalling System No. 7 was developed. One aim was to separate the call setup information and the talk path. But this signalling system also was a basis for more profound services. In the 80's, more advanced telecommunication services were developed. These services were classified as basic services, related to the basic call, and supplementary services, e.g. call forwarding and call waiting. The problem faced, however, was the time needed to implement a new service. Typical time to implement a new service could be one year, because a change involved new versions of software to be installed in all switches of the network.

To handle this problem, the concept of Intelligent Networks (IN) was developed from the middle of the 80's [IN92]. The aim for IN was to separate the service-related functionality from the basic switching and transmission functionality and to consider the service-related functionality as a programmable tool box for easy and fast implementation of new services. The IN architecture, nevertheless, was politically biased in the sense that the switches still played an important role with respect to the service design and execution.

Telecommunications Information Networking Architecture (TINA) was developed between 1995 and 1997, [TINA95] and [TINA97]. It was initialised to develop the politically biased IN model further. It was aimed to put together the best of telecommunications and information technologies aiming at providing solutions to the challenges of developing network information services combining the fixed line as well as mobile telephony and data terminal services [BDD99]. TINA included platform for distributed processing such as CORBA [CORBA98].

An important technology component used in the IN architecture is the database. The database is also an important technology component in the systems for provision of mobile telephony services that was developed from the late 70's. The database was a crucial component for realising various aspects of flexibility. One such aspect is the handling of mobility. Mobility is the ability for a component to move or for a component to be moved. Mobility management is the handling of movements of the various components that can potentially move. In this mobility context components can be persons, services, terminals, nodes, capabilities, data and programs. Mobility management allows services to find locations, and to deliver certain content to the users or terminals regardless of their location.

During the 90's the volume of the Internet traffic exceeded the volume of the telephony traffic. The use of the Internet exploded and initiatives within the Internet community resulted in concepts such as Active Networks, Mobile IP, and later Semantic Web. These are all initiatives that increase the power of flexibility within the Internet.

Starting from the fixed line telephony service the nature of the intelligence of the service providing system has changed. During the era of the automation of the telephone switches, the intelligence was associated with the amount of automatic tasks performed responding to varying user demands. Later it was associated with the network operator's ability to develop, deploy and manage services more rapidly and efficiently. Nowadays, one important aspect of intelligence is adaptability.

Adaptability is a concept widely used in the science of Biology to describe the organism's ability to adapt: "Adaptation or Adaptability is the process by which populations of organisms respond to long-term environmental stresses by permanent genetic change. Populations adapt by evolving" [DO04]. 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.

In 2004 the European Commission call for initiatives within the field of Autonomic Communication [Smi04] was launched. The work in this area was started by the IBM Autonomic project [IBM05]. The vision is a service providing system with "its own life" like a biological system, but made up of independent distributed components. While autonomic communication both put objectives regarding the external behaviour of the system and also restriction on how this functionality is made possible, in this thesis we only consider objectives regarding the external behaviour of the system. In that sense autonomic communication is a subset of adaptable service systems.

This thesis addresses mobility management in adaptable service systems. The thesis is related to TAPAS (Telematics Architecture for Play-based Adaptable Service Systems) research project. This projected started in 1997 and has been founded by the Norwegian Research Council and the Department of Telematics at NTNU.

1.2 Technologies related to Mobility Management

There are several technologies that can be related to mobility management. Figure 1-1 illustrates the history of some technologies related to mobility management.

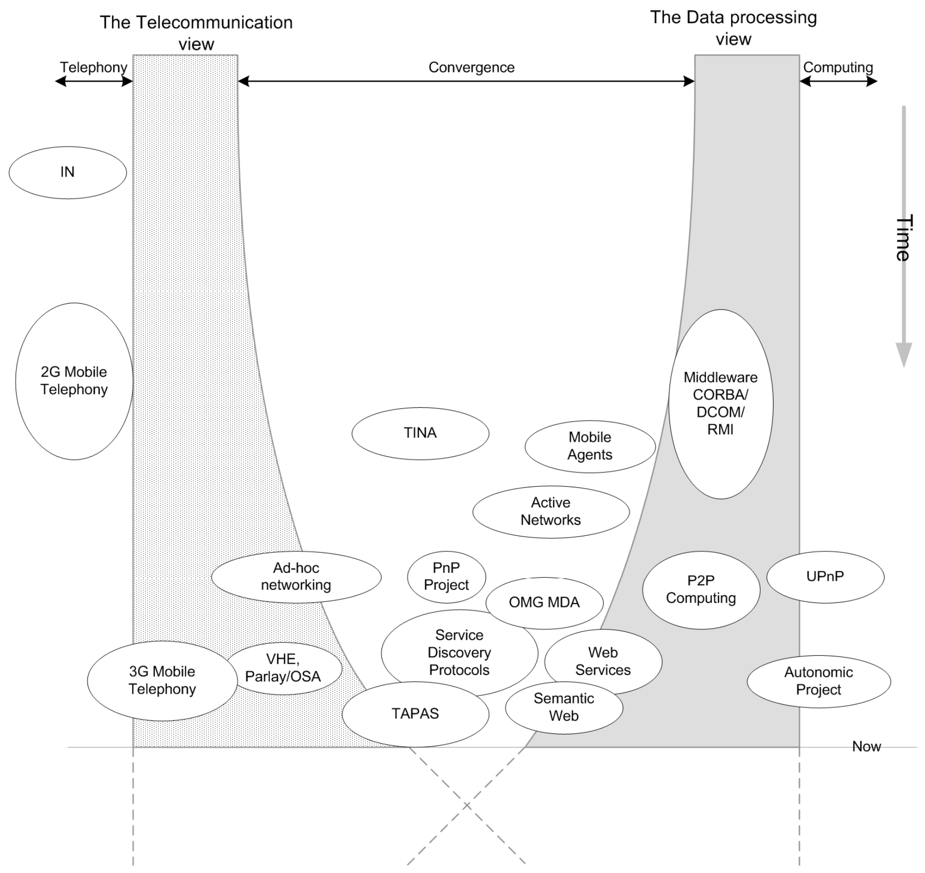


Figure 1-1 Some technologies related to mobility management

Some of the technologies in the figure are discussed in the following subsections with respect to various mobility types and contexts. The mobility types addressed are code, terminal, node, service and personal mobility.

The precise definition of the context and the mobility types to be used in this thesis will be given in Sec. 1.3. TAPAS will be presented in Sec. 1.4. The technologies VHE, OSA and Parlay, which also are in Figure 1-1, will be discussed in Sec. 2.5, because this discussion needs concepts that will be defined in Chapter 2.

1.2.1 Code mobility technologies

Code mobility is the mobility of the instantiated code among different execution environments. In the following, we briefly discuss mobile agent systems, code-on-demand, process migration, and remote evaluation.

Mobile Agent systems

A mobile agent is a program, script or package that physically travels around a network, and performs operations on hosts that have agent capabilities. These agents, which operate autonomously, usually have very specific tasks, such as fetching prices of merchandise from on-line stores. Apart from interacting with all sorts of operating systems, databases and information systems, mobile agents can also interact with other agents, meeting in agent-gathering places to exchange information. There are many different mobile agent architectures, e.g. SOMA architecture [BCS00] and KQML architecture [Mar04]. One research project [WPB99] has been dealing with the installation, extension, and modification of services based on mobile agents. In this architecture, with the capability to receive code from other parties, any service can be installed and made available for subsequent requests. The self-repairing aspect has also been addressed here. The idea is to inject all sorts of mobile code into the network that would be intelligent enough to take care of most of the problems by activating recovery routines or by planning other required activities.

Mobile agents have also been used in resource and networks management. One of the promising applications of agents is their employment in distributed resource allocation. An example is MIT Challenger system [MIT04]; which is a multi-agent system that performs distributed resource allocation.

With mobile agents, the flow of control actually moves across the network, instead of using the request/response architecture of the client/server communication. In effect, every node is a server in the agent network, and the agent (program) moves to the location where it may find the services it needs to run at each point in its execution. However, Mobile agents face many problems such as security concerns, bandwidth saving, limited agent capabilities, high development costs, dependency on expensive and local communications hardware or resources, high costs of infrastructure management, and the lack of integration with legacy applications.

Code-on-demand

Code-on-demand is a technique by which a code is downloaded and executed whenever a need for it arises. Although code-on-demand does not involve the mobility of the instantiated code, it achieves code mobility by re-instantiating the code. In [RPM00] it is argued that code on demand is probably the most widely used code mobility concept at this time. They call the mobility type achieved by code-on-demand logical mobility (which involves moving units of code and state) as opposed to physical mobility (which involves moving hosts or nodes). Web based services as well as Web Services [GGK02], and based on the widespread usage of XML, may be exploited to further enhance the concept of code-on-demand. Service descriptions and service requests can all be specified in XML and be sent to service clients and servers no matters what is the target environment, underlying communication platform, etc.

Process Migration
Process migration is a technique by which a currently executing process is transferred from one computer to another. There have been many efforts to handle process migration, e.g. the paper [TH91] uses the so-called heterogeneous process migration concept to achieve process migration. Heterogeneous means that the two computers can have different architectures, that is different CPU instruction sets and different data formats. The paper [TH91] also covered many issues related to dynamic configuration of networks using process migration.

Remote EValuation (REV)
Remote EValuation (REV) [SG90] is a technique by which a computer can send another computer a request in the form of a program. A computer that receives such a request executes the program in the request and returns the results to the sending computer. [SG90] discusses the flexibility of REV as compared to remote procedure calls. [SG90] argues that REV reduces the amount of communications between computers, and as a result improves the performance of distributed systems.

1.2.2 Active Networks and Programmable Networks

Active and programmable networks are aimed at simplifying the deployment of services. These technologies also support service creation and service deployment by moving code among network routers and switches. The paper [TSS97] gives a survey of active networks, while the paper [CKV01] discusses the main principles of these networks.

Active and programmable networks have been a research area for long time, and may be classified in several ways. We follow the classification presented in [CMK99], which states that active and programmable networks may follow one of two approaches: *open signalling* and *active networks*. Open signalling focuses on Programmable Switches, while active networks focus on the so-called Capsules. Open signalling supports the idea that new service is provided by a virtual network. All kinds of primitives are installed on routers in the networks, i.e. nodes with needed primitives are visited by packets of a stream in a certain order. Active routers process the data in packets and then re-wrap processing results and pass it to the next active node for further computation.

Active networks think that packets transport not only data, but also code itself; what they call “capsule”. In active nodes, the content of capsules is evaluated to decide whether the code should be executed at run time. The first capsule of a stream might change the processing environment of this stream. This environment will be kept until the end capsule of the stream is processed [CMK99]. Several active network and programmable network architectures have been developed, e.g. the Genesis project [GEN05], Smart Packets project [BS00], DARPA [DoD05], NetScript [YSF99], and SwitchWare [PEN05].

There are however major concerns with respect to efficiency and security of the network. Efficiency in active networks is compromised as executing a program contained in a packet at a router would typically take longer than processing a fixed format packet header. Also, with programs being injected into the network on the fly, another concern is the safety and security of the network. A program can cause disproportionate consumption of resources at a router, or can disrupt/crash a router thus affecting the core network function [VS04].

1.2.3 Mobile Telephony Systems

The history of mobility management can be traced back to the early days of the mobile telephony. The first analogue cellular system was the Nordic Mobile Telephone (NMT) in 1981. European Telecommunications Standards Institute (ETSI) targeted the mobility management by the Global System for Mobile communications (GSM) in 1991. It also developed the Universal Mobile Telecommunications System (UMTS). ETSI also tried to adopt the concepts of IN in its standards for mobility. These generations of mobile telephony systems have evolved from the basic mobile voice service, to the advanced mobile telephony services, and now to the mobile data communication services. The shift has been from supporting *terminal* mobility (as in GSM), to supporting *terminal* and *personal* mobility (as in UMTS). Personal mobility within the UMTS context handles both the movement of the user across different terminal devices, as well as the personalization of the user operating environment.

Mobile telephony systems can provide a variety of voice and multimedia services to the mobile users. The GSM system delivers advanced voice services, and to some extent data communication services. The GSM system provides mobility for the users and their terminals in and through enterprise-based domains. In GSM, the Home Location Register (HLR) and the Visitor Location Register (VLR) ensure the terminal mobility between different domains [GSM92]. HLR and VLR are databases at the home network and at the visiting network to maintain subscription, location, and charging data.

Mobility management in GSM is achieved by allowing the users to change terminals by means of a Subscriber Identity Module (SIM) card. In GSM, each terminal also has an IMEI (International Mobile Equipment Identity), and each mobile subscriber has two identifiers: IMSI (International Mobile Subscriber Identity) and TMSI (Temporary Mobile Subscriber Identity). The mobility management in GSM leads to the coupling between the user or subscriber and its terminal. There is no possibility to address the user or the terminal separately in GSM. As a result, GSM only supports terminal mobility.

1.2.4 Mobile IP

In the internet architecture prior to the mobile IP, a computer is allocated an IP address when it is connected to a specific network. When that computer moves to another network it gets a new IP address. Other nodes that are not aware of the computer’s new address cannot, for instance, access files or resources on that computer.

Mobile IP provides a transparent scheme, so that computing continues as normal when a host is moved from one subnet to another. Mobile IP identifies a node uniquely by using distinct IP addresses, while allowing it to seamlessly change its point of attachment on the internet. The mobility type supported by mobile IP is node mobility.

Mobile IP achieves node mobility by applying routing mechanisms between two agents, the home agent (HA) and the foreign agent (FA). When the mobile host leaves its home domain, the HA is informed of this move, and the FA of the visited domain relays back to the HA that the host is available in that domain. The HA then operates as a proxy, relaying all traffic to the mobile host through the visited domains FA.

Mobile IP is described in RFC2002 [PER96] and RFC3344 [PER02]. Mobile IP defines mobility functionality for IPv4 [POS93] and IPv6 [DH95]. Mobile IP functionality for IPv4 and IPv6 are very similar, see Figure 1-2.

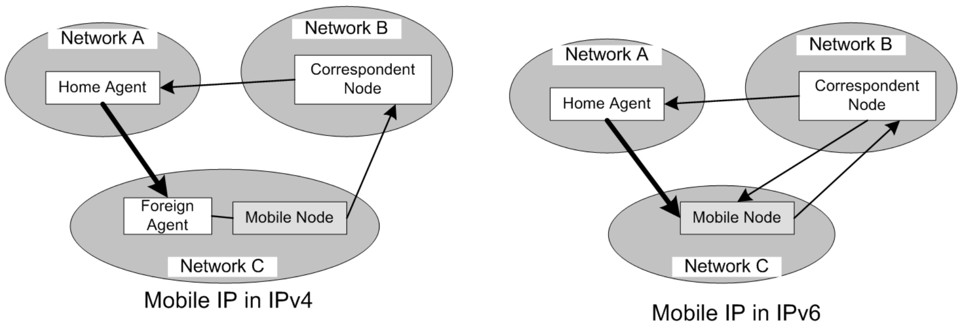


Figure 1-2 Mobile IP architecture in IPv4 and IPv6 [PER97]

The addressing scheme for IPv4 is topologically hierarchical and an IP address represents a physical location in the network topology. IPv6 provides several additional features that enhance and simplify mobility. Similar to IPv4, two IP addresses and a home agent are needed, but not a foreign agent. Through neighbourhood discovery and auto configuration of IP addresses, a mobile node can obtain a local care-of address, and informs the correspondent nodes of its new location through the source routing headers present in the IPv6.

1.2.5 Mobility management in TINA

A TINA deliverable [TINA98] highlighted the issue of mobility management in TINA. It discusses that service session mobility has been supported by allowing service sessions to be suspended and resumed. It is also argued that the personal mobility is provided by the TINA service architecture. Personal mobility in TINA is however defined as the ability of end users to originate and receive calls and access subscribed telecommunication services on any terminal in any location, and the ability of the network to identify end users as they move. Regarding terminal mobility, [TINA98] concludes that TINA does not support terminal mobility, as the terminal mobility aspects are hidden from the service. Therefore designing a service that requires explicit terminal mobility and location management is not trivial, e.g. by incorporating legacy networks in TINA. Beside this deliverable, some other efforts have been conducted to address other issues of the mobility management in TINA, e.g. [Tir98], [TLL99], and [Tha97].

1.2.6 Technologies and mobility types

Figure 1-3 illustrates mobility types related to some of the technologies considered. TAPAS mobility management architecture, to be elaborated in this thesis, supports multi mobility types. It is important, however, to notice the context for the mobility type when comparing different technologies. As an example, the definition of personal mobility is different in TINA, UMTS and TAPAS.

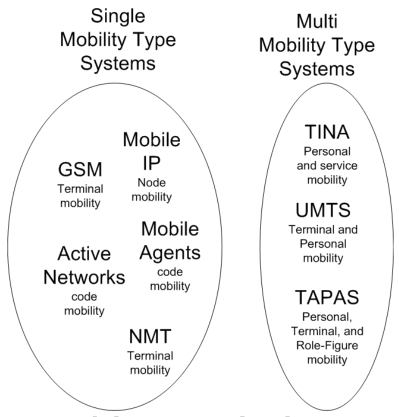

Single Mobility Type Systems
GSM
Terminal mobility
Mobile IP
Node mobility
Active Networks
code mobility
Mobile Agents
code mobility
NMT
Terminal mobility
Multi Mobility Type Systems
TINA
Personal and service mobility
UMTS
Terminal and Personal mobility
TAPAS
Personal, Terminal, and Role-Figure mobility

Figure 1-3 Mobility types related to system types

1.3 The Considered Context for Mobility Management

We are considering service systems consisting of services realized by service components. A *service* is realized by the structural and behaviour arrangement of *service components*, which by their inter-working provide a service in the role of a *service provider* to a *service user*. These definitions comprise any network based application service, including Web based services, telecommunication services as defined by TINA and ITU, network management services, and also other services provided by the OSI layers 2-7.

Service components are executed in nodes, which are different kinds of physical processing units such as servers, routers, switches and user terminals. A terminal is a node operated by a human user. Human users are associated with a node when using a service. A user is connected to a node at an access point and has a user session for each service in use.

Service components are executed as software components. The software components are generic components, which are able to download and execute different functionality depending on the need. Such generic components are denoted as *actors*. The name *actor* is chosen because of the analogy with the actor in the theatre metaphor where an actor plays a *role* in a play defined in a *manuscript*. We use the *role-figure* as a generic concept for the actor which is playing a role. So services and service components are mapped to role-figures.

To utilise the flexibility potential of this metaphor, the attributes of services, service components and nodes must be appropriately formalised, stored and made available. As a first step towards this formalisation, 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, traffic situation and Quality of Service (QoS). *Capability* is the inherent properties of a node or a user, which defines the ability to do something. An actor executes a manuscript. However, to execute this manuscript, capabilities in the node may be needed. A capability in a node is a feature available to implement services. A capability of a user is a feature that makes the user capable of using services. Capabilities are classified into:

- *functions* (pure software or combined software/hardware components, which perform particular tasks, e.g. encryption functions or special programs available for general use),
- *resources* (physical hardware components with finite capacity, e.g. processing resources, storage resources, and communication resources), and
- *data* (just data, the interpretation, validity and lifespan of which depend on the context of the usage, e.g. user login and access rights).

Adaptability is modelled as property consisting of three property classes [AHJ01]:

(a) Rearrangement flexibility
(b) Failure robustness and survivability
(c) QoS awareness and resource control

Rearrangement flexibility means that the system structure and the functionality is not fixed (adding, moving, removing nodes, users, resources, services and service components according to the needs.) Also, there is a continuous adaptation to changed environments and operation strategies/policies (new services and service components functionality, new management functionality, new policy functionality.)

Failure robustness and survivability means that the architecture is dependable and distributed (Replicated resources and functionality, inhibiting malicious and unauthorized components.) This means also that the system can reconfigure itself in the presence of failures, and that the system can provide continuous operation.

QoS awareness and resource control means that there is a functionality for negotiation about QoS and optimum resource allocation. Monitoring of resource utilization and actions and reallocation of resources is also part of this property class.

Mobility management was defined in Sec. 1.1 as the handling of movements of the various components that move. Mobility management is by definition one aspect of the rearrangement flexibility property class defined above.

This thesis focuses on the mobility of persons, services, service components, role-figures and terminals. Because services and service components are realised by role-figures, role-figure mobility is the basic mechanism to attain service component mobility. So the focus will be on the following three types of mobility:

- *Personal* mobility
- *Role-figure* mobility
- *Terminal* mobility

These mobility types handle the mobility of the three main components of the architecture: the person, the service and service components, and the terminal. Personal mobility is further decomposed into *user* mobility and *user session* mobility.

Personal mobility is the utilization of services that are personalized with end user's preferences and identities independently of both physical location and specific equipment. This type of mobility comprises the mobility of the user together with all of its data and information across different terminals. These data and information can be related to the *user* or can be related to the *user session*, defined as the representation of the user interactions with the service system. *User* mobility is the seamless access of the subscribed services at different user interfaces and terminals, while *user session* mobility is the re-instantiation and resumption of the user suspended sessions.

Role-figure mobility is the movement of the instantiated service functionality. In adaptable service systems, service functionality can be moved, due to the move of the user, or to optimize the system resource utilization. This requires that the service functionality be re-instantiated at a new location.

Terminal mobility is the movement of terminals while maintaining access to services and applications.

Notice that *security* is not defined as a property of an adaptable service system. This does not imply that it is claimed that an adaptable service system can be insecure. Security must be included in a real-world adaptable service system, but is out of the scope of the work presented in this thesis.

1.4 The TAPAS architecture

1.4.1 General

TAPAS aims at fulfilling the adaptability properties defined in Sec. 1.3. In accordance with TINA architectural framework, TAPAS is separated into a *computing architecture* and a *system management architecture* as follows:

- The *computing architecture* is a generic architecture for the modelling of any service system.
- The *system management architecture* is the structure of services and service management components.

These architectures are not independent and can to some extent also be seen as architectures at different abstraction layers. The system management architecture has focus on the functionality independently of implementation, while the computing architecture has focus on the modelling of functionality with respect to implementation. The nature of the computing as well as the system management architecture is described briefly in the following.

1.4.2 The Computing Architecture

1.4.2.1 Overview

The computing architecture is illustrated in Figure 1-4 and Figure 1-5. It has three views: the *service view*, the *play view*, and the *network view*. Figure 1-5 is a more detailed version of the play and the networks views. TAPAS Core Platform supports the play view concepts by a set of procedures and is described in Sec. 1.4.2.2.

The service view concepts are rather generic and should be consistent with any service architecture. A service system consists of service components that further can be service systems. A service component that is not longer decomposed into new service components is denoted a *leaf service component*. Likewise, the network view concepts are generic and should be consistent with any corresponding physical architecture. The network view concepts are the basis for implementing the play view concepts, which again are the basis for implementing the service view concepts.

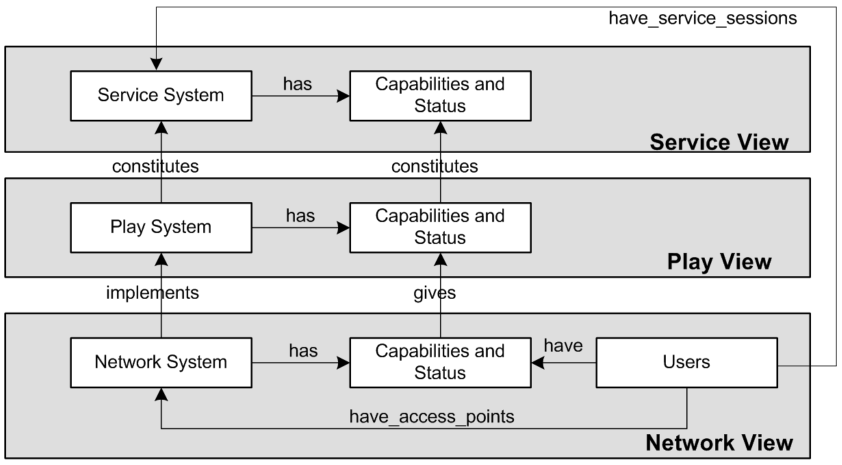


Figure 1-4 The TAPAS Computing Architecture

The play view concepts are founded on a theatre metaphor as already described in Sec. 1.3. The play view intends to be a basis for the designing functionality that can meet the requirements related to Rearrangement flexibility, the Failure robustness and survivability, and the QoS awareness and resource control.

The play view has the concepts already presented with some additions such as *director*, *role-session* and *domains*. The *director*, which acts according to a special role, is an actor managing and controlling the overall performance and execution of different role-figures that are involved in a certain *play*. A *director* 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 different *roles* have different requirements on *capabilities* and *status* of the executing system. The ability to play roles depends on the defined required capability and the matching offered capability in a node.

A service system is constituted by a *play*. A leaf service component is constituted by a *role-figure*.

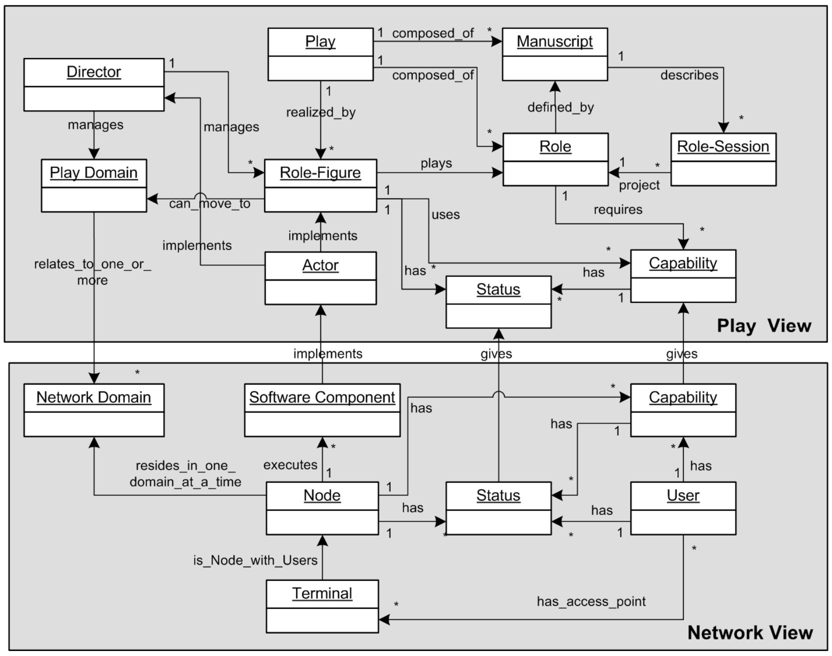


Figure 1-5 *Play View* and *Network View* – some details

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

The play view concepts allow service components to be installed into the network and execute services according to the available capabilities in the network. The play view concepts also facilitate the handling of dynamic changes in the installed service components, which occur due to changing capabilities, changing functionality, changing locations, etc. Examples of the handling of dynamic changes can be replacing the complete play definition by another, changing a specific role played by an actor, moving a role-figure from one node to another, etc. The handling of dynamic changes is needed to cope with the dynamic, heterogeneous, and rapidly evolving service systems. Also the handling of dynamic changes enable installed service components to adapt to changes in their environment without changing the definitions of the service system. The play view concepts introduce an extra complexity. However, this added complexity achieves adaptability.

1.4.2.2 TAPAS Core Platform

TAPAS core platform is a platform supporting the play view concepts of the computing architecture by a set of support procedures. These procedures are classified as *play management procedures* and *actor basic support procedures*. A third class of procedures will be introduced in Chapter 3 for mobility management. These are denoted as *mobility management procedures*.

The *play management procedures* are procedures for managing the availability of plays. These are: *PlayPlugIn*, *PlayChangesPlugIn*, and *PlayPlugOut*. These procedures will not be used in the thesis. It will be assumed that the play definitions are always available at the director. The *actor basic support procedures* are procedures for:

1) managing the existence of actors,
2) dynamic redefinition of role-figure behaviour,
3) role-figure movement, and
4) interactions between role-figures, actor capability change and monitoring.

There are quite many actor basic support procedures that cover these four purposes. We will only focus on certain procedures. The support procedures that we will use in this thesis are the following: *PlugInActor, PlugOutActor*, *CreateInterface, BehaviourChange, CapabilityChange,* and *RoleFigureMove*. These procedures are implemented as public methods in the software components in the prototype implementation of the TAPAS architecture. These methods can be invoked by the following invocation requests:

- *pluginActor* (name, location, behaviour, capability): is used to plug in an actor with a Role-Figure name, at a certain location, with a behaviour specification, and with required capabilities;
- *plugoutActor* (name): plugs out an instantiated Role-Figure;
- *createInterface* (interfaces): creates interfaces in a Role-Figure (interfaces will be discussed later);
- *behaviourChange* (behaviour, state): instantiates a new behaviour for a Role-Figure, and sets its current state;
- *capabilityChange* (capabilities): changes the capability definitions, or adds new capability definitions to a Role-Figure;
- *rolefigureMove* (location): initiates the move of a Role-Figure to a new location.

The arguments of these requests are included within the parentheses. The detailed specification of these requests is not included in this thesis.

Figure 1-6 gives an example view of the TAPAS core platform executing in three nodes and a web server. The core platform executes a node execution support in every node to facilitate the communication between nodes through the communication network. A web server is used to store the TAPAS support system and the play repository. The Actor Environment Execution Module (AEEM) is a system process or thread that can execute a collection of actors (AEEM is further explained in the terminology framework in Chapter 2).

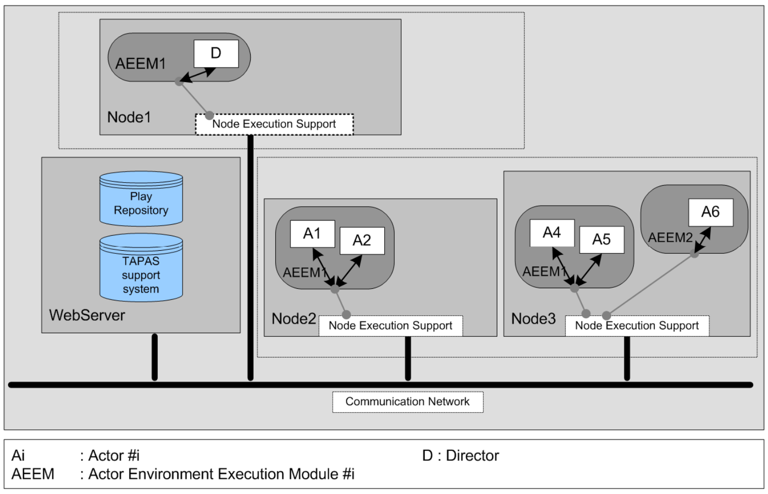


Figure 1-6 Example view of the TAPAS core platform

1.4.3 The System Management Architecture

The TAPAS system management architecture comprises several functionality components. Figure 1-7 illustrates the main functionality components that must be part of the system management architecture. Beside the executing service functionality, there are:

- *Service management:* is deployment and invocation of services and service components.
- *Configuration management*: is the optimization of service systems initial configurations and reconfigurations with respect to capabilities and status
- *Capability management*: is the installation and de-installation of capabilities, updating a view of the offered capabilities, and the dynamic capability allocation.
- *Status monitoring*: is the provision of a view of the offered status in the system.

- *Mobility management:* is the functionality of handling the movements of the components of the service systems.

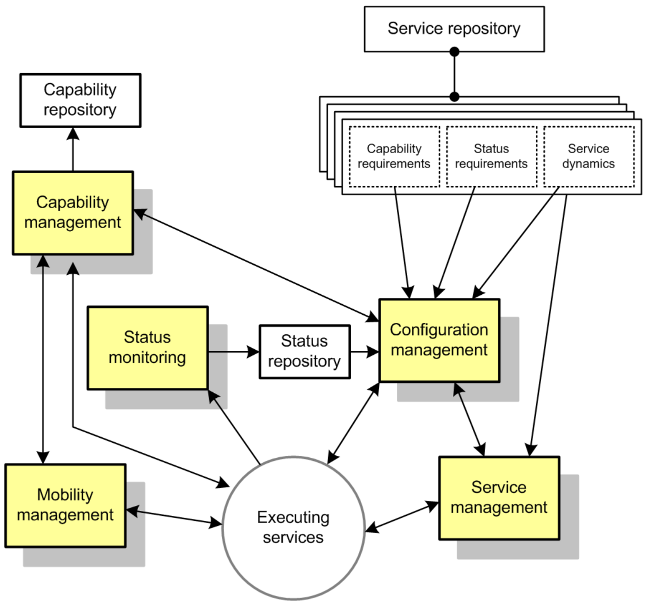


Figure 1-7 TAPAS system management architecture components

To fulfil the failure robustness and survivability requirements, the architecture must be dependable and distributed. This means that replication of resources and functionality is needed. The dependability aspect is beyond the scope of the illustration given in Figure 1-7 and also this thesis. The various functionality components are defined as being part of a centralized architecture.

1.4.4 TAPAS status related to the thesis work

The TAPAS architecture has undergone numerous revisions, and several extensions. The original concept of the architecture itself, and its core platform, were first presented in [AHW99] and then in [AHJ01]. A status of the project was presented later in [AHA03]. The configuration management architecture was introduced in [AAS02] and later in [ASA05]. The service management architecture was presented in [SJS04]. [JA03] gave an implementation of the architecture based on a state machine role specification. The mobility management architecture has been presented in several increments, [SA102], [SL02], [SA202], and [Shi03], respectively. These increments presented the evolution of the concept and its functionality. Most of the TAPAS mobility management architecture has been implemented (see [Lil03], [Luh03], [Hen04] and [Smi04]).

1.5 Outline of the Thesis

The thesis has four main parts: 1) a generic terminology framework, 2) a mobility management architecture, 3) a design model for the basic mechanism used to specify and realize the services, i.e. the role-figure model, and 4) a formal model and analysis of the role-figure model.

The terminology framework is presented in Chapter 2. This terminology provides a generic and platform-independent framework for mobility management. The mobility management architecture is presented in Chapter 3. It is elaborated based on the terminology framework. In this architecture all types of mobility are considered as a whole without applying external systems or additional middleware supporting mobility. The design model for the implementation of the role-figure is presented in Chapter 4. This model will provide the means to formalise and analyse role-figure mobility. The formal analysis of the role-figure model, which is aimed to increase the confidence of the proposed solution for the role-figure mobility, is presented in Chapter 5

Figure 1-8 illustrates the lifecycle of the research contributions resulting from the work presented in this thesis.

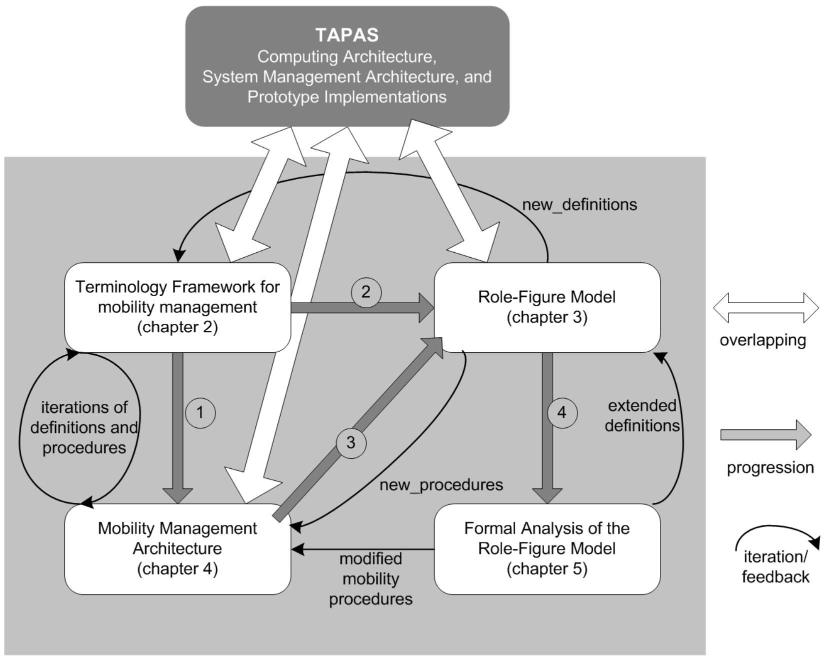

TAPAS
Computing Architecture,
System Management Architecture, and
Prototype Implementations
new_definitions
Terminology Framework for mobility management (chapter 2)
2
Role-Figure Model (chapter 3)
overlapping
iterations of definitions and procedures
1
3
4
extended definitions
progression
new_procedures
Mobility Management Architecture (chapter 4)
Formal Analysis of the Role-Figure Model (chapter 5)
modified mobility procedures
iteration/ feedback

Figure 1-8 Research contributions and research cycle

In this figure there are five contributions. There are four main contributions within the context of the mobility management in accordance with the four main parts of the thesis. Besides, there is a contribution to the TAPAS computing architecture, system management architecture, and prototype implementation.

1.6 List of Publications

During the fulfilment of the PhD study, I have contributed to the following papers as an author or co-author. The corresponding contributions presented in Figure 1-8 are indicated at each of these papers:

1. Mazen Malek Shiaa, Finn Arve Aagesen, and Cyril Carrez, “Formal Modelling of an Adaptable Service System”, *IFIP International Conference, INTELLCOMM 2005*, Montreal, Canada, October 2005 (submitted). [Contribution: *role-figure model, formal model* and *analysis*]

2. Finn Arve Aagesen, Paramai Supadulchai, Chutiporn Anutariya, and Mazen Malek Shiaa, “Configuration Management for an Adaptable Service System”, *IFIP International Conference on Metropolitan Area Networks MAN’05,* Ho Chi Minh city, Viet Nam, April 2005. [Contribution: *TAPAS Computing Architecture* and *System Management Architecture*]

3. Mazen Malek Shiaa, Shanshan Jiang, Paramai Supadulchai and Joan J. Vila-Armenegol, “An XML based Framework for Dynamic Service Management”, *IFIP International Conference, INTELLCOMM 2004,* Bangkok, Thailand, November 2004. [Contribution: *TAPAS System Management Architecture*]

4. Shanshan Jiang, Mazen Malek Shiaa and Finn Arve Aagesen, “An Approach for Dynamic Service Management”, *IFIP WG 6.3 Workshop and EUNICE 2004 on "Advances in fixed and mobile networks",* Tampere, Finland, June 14 - 16, 2004. [Contribution: *TAPAS System Management Architecture*]

5. Finn Arve Aagesen, Chutiporn Anutariya, Mazen Malek Shiaa, Bjarne E. Helvik and Paramai Supadulchai, “A Dynamic Configuration Architecture”, *IEEE/IFIP Network Operations and Management Symposium NOMS'2004,* Seoul, South Korea, April 2004. [Contribution: *TAPAS Computing Architecture* and *System Management Architecture*]

6. Mazen Malek Shiaa, “Mobility Support Framework in Adaptable Service Architecture”, *IFIP - IEEE Conference on Network Control and En-*

gineering for QoS, Security and Mobility, NetCon'2003, Muscat-Oman, October 2003. [Contribution: *Terminology Framework for mobility management, Mobility Management Architecture,* and *Role-Figure Model*]

7. Finn Arve Aagesen, Bjarne E. Helvik, Chutiporn Anutariya and Mazen Malek Shiaa, "On Adaptable Networking", *The First International Conference on Information and Communication Technologies, ICT'2003,* Assumption University - Thailand, April 2003. [Contribution: *TAPAS Computing Architecture*]

8. Mazen Malek Shiaa and Finn Arve Aagesen. "Architectural Consideration for Personal Mobility in the Wireless Internet", *Personal Wireless Communication PWC2002,* Singapore, October 2002. [Contribution: *Mobility Management Architecture*]

9. Mazen Malek Shiaa and Lars Erik Liljeback. "User and Session Mobility in a Plug-and-Play Architecture", *IFIP WG6.7 Workshop and Eunice Summer School on Adaptable Networks and Teleservices,* Trondheim, Norway, September 2002. [Contribution: *Mobility Management Architecture*]

10. Finn Arve Aagesen, Chutiporn Anutariya, Mazen Malek Shiaa and Bjarne E. Helvik. "Capability Specification and Selection in TAPAS", *IFIP WG6.7 Workshop and Eunice Summer School on Adaptable Networks and Teleservices,* Trondheim, Norway, September 2002. [Contribution: *TAPAS System Management Architecture*]

11. Mazen Malek Shiaa and Finn Arve Aagesen. "Mobility management in a Plug and Play Architecture", *IFIP TC6 Seventh International Conference on Intelligence in Networks,* Saariselka, Finland, April 2002. [Contribution: *Mobility Management Architecture*].

CHAPTER 2

Terminology Framework

2.1 Introduction

THE TERMINOLOGY FRAMEWORK provides generic concepts and definitions for mobility management for the three main mobility types defined in Chapter 1. The definitions in this chapter are organised in groups. Each group contains the definitions related to one mobility type. These groups are:

– Personal mobility related definitions
– Role-figure mobility related definitions
– Terminal mobility related definitions

These groups of definitions are presented in Sec. 2.2, Sec. 2.3, and Sec. 2.4, respectively. In these sections we also give propositions and requirement rules for every group of definitions. Propositions will state which definitions are required in the handling of the defined mobility types. The requirement rules will give the overall requirements and conditions regarding these definitions. Mobility management solutions for the defined mobility types must satisfy these requirement rules.

The definitions, propositions, and requirement rules in these sections will be generic. However, certain discussions will be presented within the context of TAPAS. In these discussions the definitions that are specific to TAPAS concepts will be explicitly mentioned.

Sec. 2.5 gives a discussion on the terminology framework. In this section we give a discussion on some concepts of the terminology framework related to similar concepts and standards.

2.2 Personal mobility related definitions

2.2.1 Conceptualisation of personal mobility

In any service system the user is a very essential concept. Services are based on a service-provider to a service-user relationship. In this section the user is the human user or the person.

In Chapter 1, we defined the **personal mobility** as the utilization of services that are personalized with end user's preferences and identities independently of both physical location and specific equipment. This type of mobility comprises the mobility of the user together with all of its data and information across different terminals. These data and information can be related to the user or can be related to the user session. A user can move with or without its user session. In this section we give the personal mobility related definitions in two groups:

- User mobility related definitions
- User session mobility related definitions

In this respect, we decompose the personal mobility into user mobility and user session mobility. The issues of the personalization of the user environment, user applications, user profile, etc. are parts of the user mobility. The issues related to the suspension and resumption of user sessions are parts of the user session mobility.

Figure 2-1 illustrates concepts related to personal mobility.

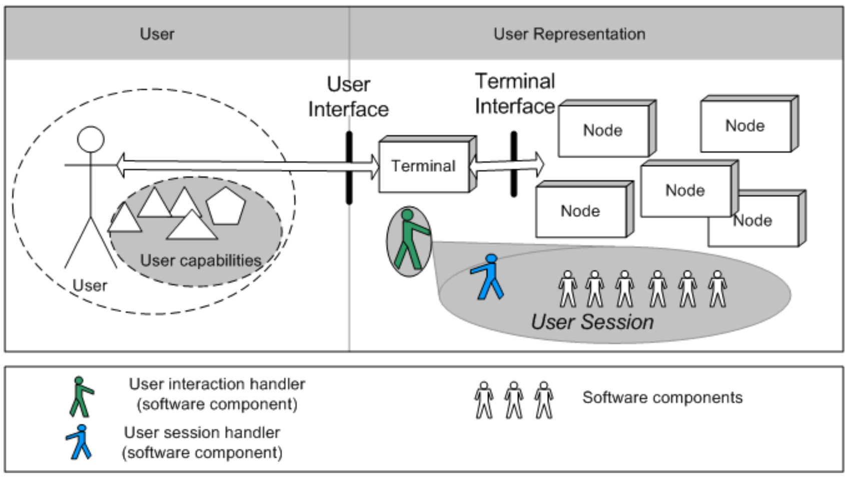


Figure 2-1 Concepts related to personal mobility

In this figure, a *user* has *user capabilities*, e.g. its identification, passwords, personal settings, preferences, personal data, phone books, buddy lists, etc. A user is related to a *terminal* (a user can also be related to a set of terminals instead of one single terminal). A user has a *user session* through the terminal (A user also can have multiple user sessions through the same terminal). Figure 2-1 shows that a user is represented within the architecture by a set of *software components*, which execute in user sessions that execute in terminals and nodes. This set of software components is called the *user representation*. A software component, called *user interaction handler*, controls this user representation and handles the user interactions with the service system. A user may interact with the system, or services, within a specific user session. A software component, which is called *user session handler*, controls this user session. A user can access services using different terminals. The user capabilities can be moved among these terminals. There are two physical access points or interfaces:

– *User interface*
– *Terminal interface*

These concepts provide a flexible mechanism to represent users and terminals independently of each other. A user may be identified by a name, while a terminal may be identified by a network address.

In Sec. 2.2.2 we give the user mobility related definitions and requirement rules, while in Sec. 2.2.3 we give the user session mobility related definitions and requirement rules.

2.2.2 User mobility related definitions and requirement rules

2.2.2.1 Definitions

User mobility is the seamless access of the subscribed services at different user interfaces and terminals. This comprises the mobility of the data and information that are related to the user. As a starting point, the handling of the user mobility requires the handling of the user capabilities defined earlier. These capabilities may be partly included in the user terminals, but mainly they are stored and maintained centrally in the service system. A user profile may be used to include the user’s capabilities. Besides, a user profile may include other user-related information. Such information can be the user location, terminal-related data, subscribed services, access permissions, and authorization constraints. The user profile may also include the set of optional preferences and setting attributes of the services associated with the user.

User profiles are stored in databases, or user profile bases. One main assumption regarding the management of user profile bases is needed. In the service system where user profiles are used and maintained a component of the service system must be responsible for the management functionality of these bases. We call such a component a *supervisory object*.

Figure 2-2 illustrates a user with a terminal, its corresponding user profile, and a user profile base. The data and information of this user profile may be updated and accessed throughout the service execution. Beside this basic arrangement of user profiles it is possible to apply any other organization of user profiles, e.g. there can be terminal-based, application-based, or network-based user profiles.

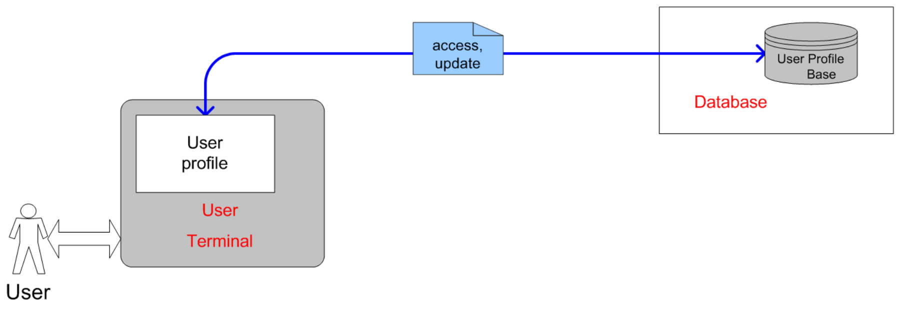


Figure 2-2 Illustration of user profile and user profile base

User mobility requires user-terminal independence. For this purpose a software component that handles the user login is needed, so that different users can use the same terminal. A user starts interacting by sending a login request to this software component. The login request must contain the user identification and it is password. We call such a component *Login Agent*.

Furthermore, the user representation illustrated in Figure 2-1 must be different for users who have user profiles in the system, and for users who do not have such profiles. The main difference between these two representations is concerning the access rights and privileges assigned to the users. Accordingly, there can be two user interaction handlers:

– *Visitor Agent* to handle users having user profiles in the service system, or being at home domain (we call such a service system the use's home domain, where a user profile for the user exists);

– *User Agent* to handle users without user profiles, or being at visitor domain (we call such a service system the user's visitor domain, where a user profile for the user doesn't exist).

In the discussion above, we presented concepts related to user mobility. The following list gives precise definitions:

User is the human user that accesses the services via *terminals*. Each user has service subscription information that is stored in a database in the system (*user profile base*). A user of a service, who is performing the interactions with the service instance, is the subscriber of the service, who owns the subscription contracts.

User Capabilities are the set of user-related data and resources that are not part of the service system, but can be used or needed in the service interactions.

User Representation is the way a user is represented and the way its interactions are handled in the service system.

User Interface is the access point between a *user* and a *terminal*.

Terminal Interface is the access point between a *terminal* and the other *nodes*.

User Session is the representation of the user interactions with the service system.

Person is a *user* with *user capabilities* and *user sessions*.

Login Agent is the software component that handles the user login to the service system. It also handles the user initial interactions with the service system. It is responsible for the initialization of the service instances under managed permissions, constraints, and optional preferences, as described in the *user profile*. Login agent executes in the user terminal.

User Agent is the user interaction handler (software component) responsible for controlling the user interactions with the user's home domain.

Visitor Agent is the user interaction handler (software component) responsible for controlling the user interactions with the user's visitor domain.

Supervisory Object is a component (software component) that has a central role and is responsible for a management functionality in the service system.

User Profile is the representation of the *user capabilities* and the user information relevant to the provision of services.

User Profile Base is the informational or knowledge base where the *user profile* information is maintained. User profile base is managed by a *supervisory object*.
User profile Update is a request to update the user's maintained profile.
User profile Access is a request to ask for the user's maintained profile.

These definitions constitute a self-contained set. The following proposition states the definitions required for the handling of user mobility:

Proposition-1 The handling of user mobility needs the definitions of the following: *user, user capabilities, user representation, user interface, terminal interface, user session, person, login agent, user agent, visitor agent, supervisory object, user profile, user profile base, user profile update,* and *user profile access.*

2.2.2.2 Requirement rules

The definitions related to the user mobility need to be related to each other. For example the definition of user profile must be related to the definition of user, i.e. it must be specified whether or not a user is required to have a user profile to execute services. The requirement rules presented below, *UR1-UR6*, show the overall requirements and conditions regarding the user mobility definitions. They are based on the concepts presented in Sec. 2.2.2.1. These rules must be satisfied to achieve the management functionality associated with the user mobility. Further requirement rules, however, may be specified based on the used application platforms and service systems. In the rules "*MUST*" is used to signify that these rules are required.

UR1 requires the relationship between the user and its user profile. *UR2* specifies a requirement on the update of the user profile. *UR3* and *UR4* are used for requirements on the user profile bases. *UR5* and *UR6* describe how the user must interact with the service system, and how the user interaction must be handled.

UR1. A user with subscribed services in the service system *MUST* have at least one user profile.
UR2. A user in its home domain *MUST* be able to update its profile.
UR3. User profiles *MUST* be maintained in user profile bases.
UR4. A supervisory object *MUST* exist in the service system, which is responsible for administrating the user profile bases.
UR5. A user in its home domain *MUST* start interacting by sending a

login request that contains its user identification and password. The supervisory object *MUST* check if the user identification is valid and if the password is correct.
UR6. Interactions of any user *MUST* be controlled by a user interaction handler (which can either be a *UserAgent* or a *VisitorAgent*).

2.2.3 User session mobility related definitions and requirement rules

2.2.3.1 Definitions

User session mobility is the re-instantiation and resumption of the user suspended sessions. This means the movement of the user session from one access point to another possibly at different time instances. User session mobility comprises the mobility of the data and the information that are related to the user session. Similarly to the user mobility, in order to maintain a user session, a user session profile must be properly described and maintained. User session profiles are maintained to facilitate the resumption of suspended sessions. User session profiles are stored in databases, or user session bases. Similarly to user profile bases, we assume user session bases to be managed by a supervisory object in the service system.

Figure 2-3 demonstrates a user with a terminal, its corresponding user sessions, and a user session base. Beside this basic arrangement of user session profiles it is possible to apply any other organization of user session profiles, e.g. there can be terminal-based, application-based, or network-based user session profiles.

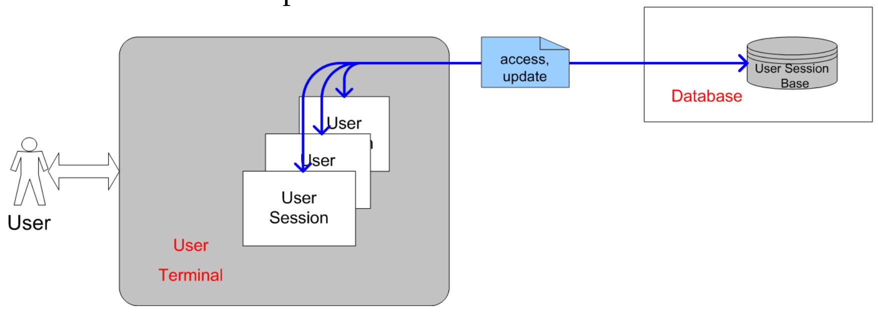


Figure 2-3 Illustration of user sessions and user session base

In the brief discussion above, we used user session profile, user session base, access and update of user sessions to describe the concept of user session mobility. In the following we give precise definitions for these terms:

User Session Profile is the representation of the user session information relevant to the maintenance of the *user sessions*.
User Session Base is the informational or knowledge base where the user session information is maintained. User session base is managed by a *supervisory object*.
User session Update is a request to update the user's maintained sessions.
User session Access is a request to ask for the user's maintained sessions.

These definitions constitute a self-contained set. The following proposition states the definitions required for the handling of user session mobility:

Proposition-2 The handling of user session mobility needs the definitions of the following: *user session profile, user session base, user session update,* and *user session access.*

2.2.3.2 Requirement rules

The definitions related to the user session mobility need to be related to each other. We must specify whether a user must have a user session profile maintained in the service system. Also we must specify the relationship between a *UserAgent* or a *VisitorAgent* and the user session, i.e. which agent must handle the suspension and resumption of user sessions. The following requirement rules, *SR1-SR8*, must be satisfied to achieve the management functionality associated with the user session mobility. Further requirement rules may be specified if a more elaborated concept for the user session is used.

SR1 describes the relationship between the user and the user session profile. *SR2* and *SR3* specify requirements on the maintenance of user session bases. *SR4* specifies a requirement on the control of user sessions. *SR5* requires the handling of the suspension and resumption of user session. *SR6*, *SR7*, and *SR8* are used to require the handling of suspension and resumption of user sessions for users at home domain, considering that users at visitor domain may or may not be offered such functionality.

SR1. A user accessing services in the service system *MUST* have at least one user session profile.
SR2. The user session profiles *MUST* be maintained in user session bases.
SR3. A supervisory object *MUST* exist in the service system that adminis-

trates the user session bases.
SR4. User sessions *MUST* be controlled by either a *UserAgent* or a *VisitorAgent*.
SR5. Suspension and resumption of user sessions *MUST* be handled by the *UserAgent*.
SR6. A user in its home domain *MUST* be able to resume a session, and suspend a session.
SR7. If a user in its home domain suspends its session the *UserAgent MUST* save the user session profile of the user.
SR8. If a user in its home domain resumes a suspended session the *UserAgent MUST* access the user session profile of the suspended session.

2.3 Role-figure mobility related definitions and requirement rules

2.3.1 Definitions

The service components collaborate to achieve the goals associated with the service functionality. These service components will be realized by software components denoted as role-figures. Role-figure is a generic concept and this section focuses on the mobility of role-figures. The software components – actors – that execute role-figures are also generic components capable of executing in the network nodes and terminals. **Role-figure mobility** is the movement of instantiated role-figures. To handle this type of mobility we need a model with appropriate concepts. We will to some extent use TAPAS concepts for this elaboration, and recall the following definitions from the TAPAS computing architecture:

Actor is a generic object with a generic behaviour.

Role-figure is an actor with a specific behaviour. It realizes service-components. It behaves and performs according to the role it is assigned. Each role-figure executes a particular role as part of an overall functionality. This role is described in a manuscript.

Director is a supervisory actor that is responsible for the management of a domain. It interacts with actors and role-figures, and keeps an update of their availability.

Role-session is the projection of the behaviour of a role-figure with respect to one of its interacting role-figures. It represents a relationship between two role-figures.

Interface is the means to gain access to object instances from other objects. Interfaces define also the access point between the objects and their environments.

Role-figure Capability is a property of the node where the role-figure executes. It is consumed or used by the role-figure.

Play Domain is a domain managed by one director. It relates to one or more network domains and includes the nodes and terminals where actors and role-figures execute. Play domains are used to manage the federation of responsibility between different directors (the play domain is denoted as ***domain***). ***Sub-domain*** is a subset or part of a specific domain.

Actor Environment Execution Module (AEEM) is the run-time system process or thread that is capable of instantiating actor instances, and is eventually responsible for the execution of the architecture functionality. AEEMs can execute concurrently in the same node.

Role-figure plug in is the instantiation of an actor with specific behaviour. Only role-figures can behave according to roles, consume capabilities, and have role-sessions with other role-figures. Figure 2-4 demonstrates the relationship between an actor and a role-figure (or *RF*).

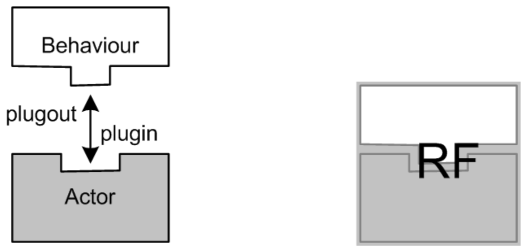


Figure 2-4 Demonstration of the actor role-figure relationship

Figure 2-5 gives an illustration of concepts related to role-figure in. In this example an instantiated role-figure executes in *Node1*. It consumes certain part of the node's capability set, e.g. domain name, terminal location, memory, processing power. The role-figure has its own session, denoted as child-session. This means that it instantiates role-figures to perform certain tasks. The role-figure interacts with other role-figures via role-sessions. It interacts also with the domain's director and some special-purpose role-figures, which are certain role-figures that have domain-specific functionality, e.g. domain servers, name servers, directory servers, capability managers.

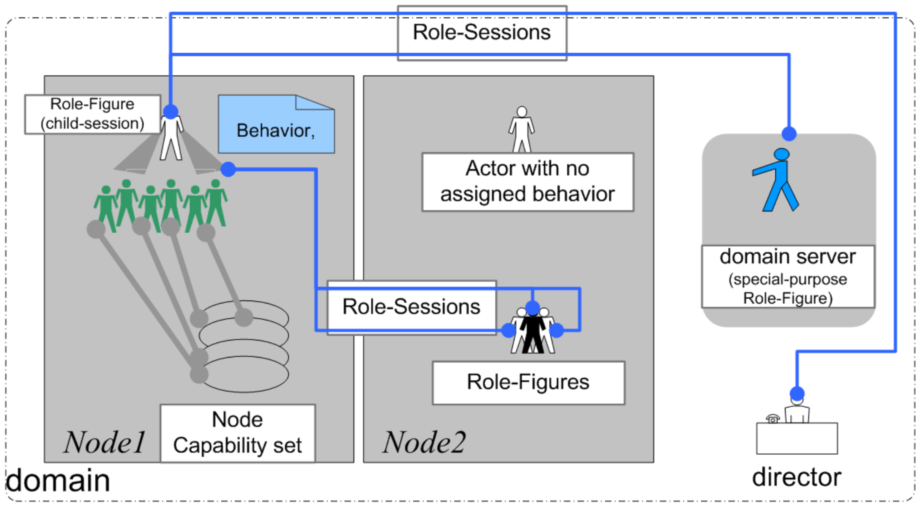

Role-Sessions
Role-Figure (child-session)
Behavior,
Actor with no assigned behavior
domain server (special-purpose Role-Figure)
Role-Sessions
Role-Figures
Node Capability set
Node1
Node2
domain
director

Figure 2-5 An illustration of concepts related to role-figure

Role-figures move between execution environments, e.g. between two terminals. Role-figures need to move due to several reasons, e.g. changed capability requirements, change in the role-figure functionality, changes in the service functionality, etc. It is crucial to ensure that the role-figure is capable of continuing its execution after the movement. Intuitively, certain characteristics of the role-figure need to be preserved, e.g.:

– The role-figure’s instantiated behaviour and current state
– The role-figure’s consumed capabilities
– The role-figure’s established role-sessions and interfaces
– The role-figure’s child-session if any

The structure or the model of the role-figure needs to be defined. This structure or model will be the basis for decisions on the characteristics of the role-figure that will be preserved during the movement. As a consequence, there can be different role-figure mobility realizations based on how these characteristics will be preserved. Additionally a set of domain-based rules indicating the conditions and requirements of moving role-figures must also be defined.

When a role-figure moves from one execution environment to another a supervisory object in the system is responsible for managing this move. The role-figure must report its new location information to this supervisory object upon the move. Other role-figures can discover the new location of the moving role-figure by consulting the supervisory object.

The movement of a role-figure needs to be transparent to other role-figures. Role-figures must be identified and addressed as they execute in terminals and nodes. These terminals and nodes have identities and addresses. It is important at this stage to discuss certain requirements on the definitions of the identity and address of role-figures, terminals, and nodes. Given that one of these entities (role-figure, terminal, and node) has an identity or address, given also that this identity/address belongs to a range of identities/addresses, the following requirements should hold in any adaptable service system:

- The entity should be identifiable by other entities;
- The entity's identity/address should be unique in its range of identities/addresses.

It is possible, however, to introduce flexibility measures as well. As an example, a set of identities/addresses can be invisible by other sets, reusability can be applied in identities/addresses if they are freed up, and several entities may share identities/addresses if an additional mechanism is provided to resolve them.

In the following precise definitions for the concepts related to role-figure mobility are given:

Role-figure Child-session is the representation of the role-figure's instantiated *role-figures* with their respective data, *role-sessions*, settings, etc.

Special-purpose Role-figure corresponds to a *role-figure* whose availability and accessibility is only defined and relevant in a given *node, sub-domain,* or *domain*.

Role-figure Model is the design model to represent the structure and the behaviour of the *role-figure*. A *role-figure*, according to this model, comprises the following parts: the behaviour described by a specification, the consumed capabilities in the node, the role-sessions with other role-figures, the queue of incoming messages, and the executing methods (or the role-figure active tasks). [1]

Mobility Strategy is a set of domain specific rules and conditions that govern the role-figure mobility procedures.

Role-figure State-information is the information of the role-figure instantaneous run-time conditions including program variables, stack of call requests, and queued messages.

[1] The role-figure model and its parts will be discussed in detail later in Chapter 4.

Execution Environment is the environment where a *role-figure* executes and consumes capabilities. Execution environment is defined by a domain, a node, and a system process.

Role-figure Location-of-performance is the location information that is associated with a *role-figure,* and reflects its *execution environment*.

Role-figure Identity is the information that uniquely identifies a *role-figure* in an execution environment.

Role-figure Address is the information that facilitates the communication from and to a *role-figure*. Role-figure address is derived from the *role-figure identity* and its *location-of-performance*[2].

State-information Mobility is the ability to move the state-information of a moving role-figure.

Role-session Mobility is the re-instantiation of the *role-sessions* of a moving role-figure.

Capability Mobility is the reclamation of the acquired or consumed set of capabilities by a moving role-figure.

Child-session Mobility is the re-establishment of a moving role-figure's child-session, including the instantiation of all role-figures of its child-session with their respective data and role-sessions.

These definitions constitute a self-contained set. The following proposition states the definitions required for the handling of role-figure mobility:

Proposition-3a The handling of role-figure mobility needs the definitions of the following: *role-figure child-session, special-purpose role-figure, role-figure model, mobility strategy, role-figure state-information, execution environment, role-figure location-of-performance, role-figure identity, role-figure address, state-information mobility, role-session mobility, capability mobility,* and *child-session mobility*.

The role-figure mobility is a special type of mobility that is partly defined based on the concepts of TAPAS. Certain definitions that are specific to role-figures are not needed for the handling of service component mobility in general, e.g. role-figure model and role-figure state-

[2] A role-figure is given a location-of-performance that reflects its AEEM process. A role-figure identity in this case is the name given to that role-figure that reflects its role. This identity distinguishes it from other role-figures within the same play. The address of this role-figure is composed from the address of the actor instance that executes it and the location of the node it resides in.

information. Role-figure location-of-performance, role-figure identity, and role-figure address can be substituted by service component location that shows where a service component resides. A generic terminology for the service component mobility can be given by the following proposition:

Proposition-3b The handling of service component mobility needs the definitions of the following: *service component child-session, special-purpose service component, mobility strategy, execution environment,* and *service component location.*

2.3.2 Requirement rules

The requirement rules presented here, *RR1-RR6*, show the overall requirements and conditions regarding the definitions of role-figure mobility. These rules are based on the role-figure definitions in Sec. 2.3.1.

The requirement rules *RR1*, *RR2*, and *RR3* specify requirements on the supervisory object that handles the role-figure mobility. *RR4* specifies a requirement on the mobility strategy, but no requirements are given on the content of this strategy. *RR5* and *RR6* specify requirements regarding role-figure mobility between several domains or sub-domains.

RR1. A domain or a sub-domain *MUST* have at least one supervisory object responsible for managing role-figure mobility.

RR2. Role-figures that can move *MUST* be supervised by at least one such supervisory object.

RR3. The supervisory object managing role-figure mobility *MUST* maintain a record of the moving role-figures under its supervision.

RR4. Mobility management architecture for role-figure mobility *MUST* have a mobility strategy.

RR5. After a role-figure move to another domain or sub-domain the moving role-figure *MUST* be supervised by another supervisory object that exists in the other domain or sub-domain.

RR6. If more than one supervisory object manages the mobility of role-figures in one domain or in one sub-domain the relationship between these supervisory objects *MUST* be defined.

2.4 Terminal mobility related definitions and requirement rules

2.4.1 Definitions

Terminal mobility is the movement of terminals while maintaining access to services and applications. This implies the change of the terminal location as well. Terminal mobility can be conducted within one domain or across different domains.

Terminal moves with the human user that is associated with it. Terminal mobility introduces implications on the user mobility. From the personal mobility related definitions we have concluded that services can only be accessed when the logged in user has the appropriate subscriptions and access rights. However, when moving the terminal used by the user to a different domain these subscribed services may or may not be accessible. The terminal must be identified, addressed, and accepted by the new domain so that the user can access the subscribed services. Taking this argument into account, the user and the terminal are not independent anymore.

Terminal mobility also has implications on the role-figure mobility. Terminal mobility could influence the functionality of the instantiated role-figures in a moving terminal, and initiate the mobility of some of these role-figures. Two typical examples of this influence are:

- The limited capabilities of a terminal in a new location.
- The limited access rights of a user in a new domain.

For the purpose of handling terminal mobility, terminals must be identified and addressed in the service system. The identification and the addressing need to have validity over a period of time. Also, terminals have locations, and when terminals move their locations change.

A terminal executes a software component that is responsible for tracking its location, e.g. its network address or its geographical location. We call this software component *mobility agent*. The domain the terminal resides in executes another software component that is responsible for updating the terminal's location. We call this software component *mobility manager*. The *mobility manager*, also considered as a supervisory object of the service system, needs to be known by all terminals in a domain. If there exist several mobility managers in a domain, the delegation of responsibility should be clarified. Also the domain-to-domain relationships must be defined. The update and the discovery of terminals locations need also be handled.

The brief discussion above showed the concept of terminal mobility. Our focus is on terminal mobility issues that implicate and affect the other mobility types, i.e. personal and role-figure mobility. The handling of terminal mobility is limited to the following set of definitions:

Terminal Location is the location information that is associated with a *terminal*, and reflects its physical location.

Terminal Identity is the information that uniquely identifies a *terminal*, either globally or within a specific domain.

Terminal Address is the information that facilitates the communication from and to a *terminal*. Terminal address can be derived from the integration of the *terminal identity* and its *location*[3].

Mobility Agent is a software component responsible for managing the *terminals location*.

Mobility Manager is a software component responsible for managing the mobility of terminals.

The handling of terminal mobility can be described by these definitions. These definitions are self-contained. The following proposition states the definitions required for the handling of terminal mobility:

Proposition-4 The handling of terminal mobility needs the definitions of the following: *terminal location, terminal identity, terminal address, mobility agent,* and *mobility manager*.

2.4.2 Requirement rules

The set of terminal mobility related definitions we presented above need to be related to each other. For example it must be stated whether a terminal is managed by one *mobility agent* or several ones. The requirement rules presented below, *TR1-TR6*, show the overall requirements and conditions regarding terminal mobility definitions and must be satisfied to achieve the terminal mobility management functionality. These require-

[3] In the case of the mobile phone terminal, for instance, the terminal location is its geographical location. This is used to locate the terminal and assign it with a specific base station within a mobile operator domain. Its terminal identity could be the mobile phone Equipment Identity. However, the terminal address is usually extracted from the SIM card identity and the user phone number.

ment rules are based on the concepts presented in Sec. 2.4.1, and further rules may be specified to handle more elaborated terminal mobility issues.

In *TR1* we give a requirement on the terminal identification, location and address. *TR2*, and *TR3* specify requirements on the mobility manager that handles the terminal mobility. *TR4* specifies a requirement on the mobility agent. *TR5* and *TR6* specify requirements regarding the update of terminal locations. If any of these rules is not satisfied by a mobility management architecture, then the role-figure mobility functionality cannot be completely handled.

TR1. Every terminal *MUST* have a unique identification in the domain. Every terminal must have location and address information.
TR2. A domain or sub-domain *MUST* have at least one mobility manager.
TR3. Every terminal *MUST* be supervised by at least one mobility manager.
TR4. A terminal *MUST* execute a mobility agent.
TR5. Mobility agent *MUST* be responsible for sending updates regarding the terminal location information to the mobility manager.
TR6. The mobility manager *MUST* maintain a database with information on the terminals identifications and their locations.

2.5 Discussion

In this section we give a discussion on some concepts of the terminology framework related to similar concepts used in some standards related to mobility management. These standards are *VHE*, *OSA*, and *Parlay*, which are made by *3GPP* (3rd Generation Partnership Project), and *ETSI*.

2.5.1 Virtual Home Environment (VHE)

Virtual Home Environment (VHE), defined in [VHE02] and [GUP03], allows users to be consistently supported with their personalized features, customizable user interfaces, and service preferences regardless of the network terminal and location. VHE is meant for populating the use of future telecommunication services by pursuing the idea of service universality, which allows a user to transparently access services at anytime from anywhere [LYB02]. The key requirement of the VHE is to provide a user with a personal service environment, which consists of personalized services, personalized user data, and consistent set of services from the user's perspective. So far, VHE as a concept has gained a wide acceptance. However, a viable architectural solution to allow for flexible service development, deployment and management still lacks.

In the following we highlight the main definitions of VHE that have similar or counterpart definitions in the terminology framework.

VHE definitions	**Terminology Framework** definitions
Home Environment is the part of VHE responsible for the overall provision and control of the personal service environment of the subscribers.	*Home domain* and *UserAgent* represent the control of the user's interactions within its home domain and its operating environment, i.e. its terminal.
Personal Service Environment contains personalized information defining how subscribed services are provided and presented towards the user. Each subscriber of the Home Environment has its own personal service environment, defined in terms of one or more user profiles.	*User Profile* and *User Session Profile* represent the service-application and user customizability. User may personalize their services and applications in the *user profile*, as well can have different *user sessions* in different *user session profiles*.
User Service Profile contains identification of subscriber services, their status and reference to service preferences. This is part of the user profile information.	*User Profile* includes the service subscription information. However, the services themselves may have sessions, which can be maintained separately.
User Profile is a set of information necessary to provide a user with a consistent, personalized service environment, irrespective of the user location or the terminal used.	*User Profile* maintains the information necessary to provide a user with personalized services.

2.5.2 Open Service Access (OSA)

Open Service Access (OSA) [OSA03] (also [SSC02] and [SRO04]) is a service toolkit that enables applications to implement the services and access the network functionality. Network functionality offered to applications is defined in terms of a set of Service Capability Features (SCFs). These SCFs provide functionality of network capabilities, which is accessible to applications through the OSA interface upon which service developers can rely on when designing new services, as in Figure 2-6.

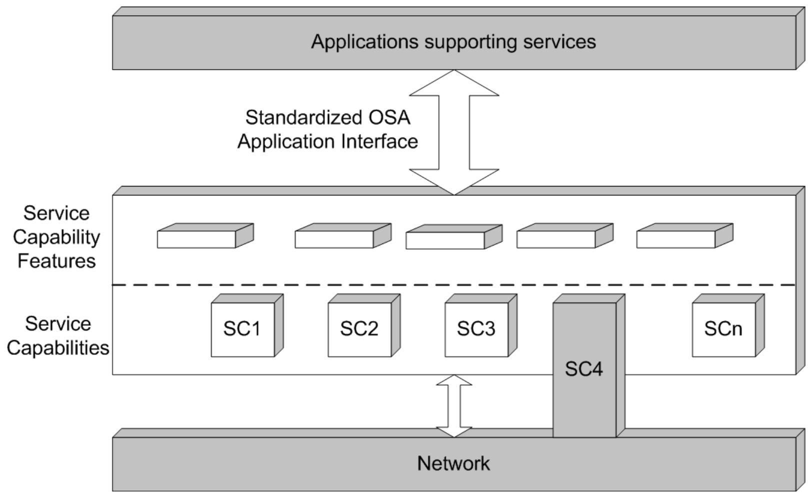

Applications supporting services
Standardized OSA Application Interface
Service Capability Features
Service Capabilities
SC1
SC2
SC3
SC4
SCn
Network

Figure 2-6 Application access Service Capability Features in OSA [OSA03]

The aim of OSA is to provide a standardized, extensible and scalable interface that allows for the inclusion of new functionality in the network with a minimum impact on the applications using the OSA interface. OSA aim at developing an open framework to provide services and applications controlled access of network resources based on the concept of service capability.

The OSA approach is based on the concept of service capability. Similarly the terminology framework is based on the concept of role-figure capability. The terminology framework discusses the role-figure mobility based on this capability concept. Role-figure mobility can be deployed to achieve better utilization of system resources when role-figure capabilities deteriorate.

There are also other similarities between the OSA and the terminology framework. For instance, OSA defines *policy* as a formalism that may be used to express business, engineering or management criteria, and is represented by a set of rules that may be created, modified, activated, deactivated, etc. In the terminology framework *mobility strategies* are the sets of domain-based rules that control the mobility procedures and can be similar to the policies of OSA. OSA also defines event notification function similar to the *user profile update* and the *user session update* events. Among the handled events by OSA are: "the user's status is changed", "the user's location is changed", and "terminal capabilities are changed" which are similar to the way how we handle the user profile and the concept of user representation in the terminal. Other 3GPP standards that are related to the OSA have addressed issues regarding the mobile equipment, which we denote terminal. In [MEE02] core requirements for a mobile execution environment are discussed, while in [PME02] functional specifications of features to personalise the mobile equipment are defined. These two specification handle similar issues to the one discussed in the personal mobility related definitions.

2.5.3 Parlay

The Parlay Group is an open multi-vendor consortium formed to develop open application programming interfaces (APIs), enabling enterprises to develop applications that operate across multiple networking platform environments [Par03].

It defines the Service Capability Server (SCS), and offers certain APIs to utilize pre-existing network-independent functionality such as those for call control and location management. Both, the SCS and the APIs, enable hosting applications outside specific networks while giving them controlled access to the network resources. Parlay is based on the main principles of the OSA, and its APIs standardize the application interfaces. Therefore Parlay is often called Parlay/OSA. The Parlay group has developed a set of Parlay specifications along their corresponding standardized interfaces. Examples of such specifications handle mobility management, terminal capabilities, data session control, and account management.

Similar to the OSA approach, the Parlay approach is in-line with many definitions of the terminology framework. However, both OSA and Parlay are aiming for the service development and service openness rather than the service adaptability. They provide services with open and technology-independent interfaces. Terminology framework, instead, focuses on providing services with mobility management support. On the other hand, Parlay/OSA has relevance in the Mobility Management architecture. The Parlay APIs for mobility and terminal capabilities provide similar support to service instance as those mobility management procedures for personal and terminal mobility.

CHAPTER 3

The Mobility Management Architecture

3.1 Introduction

THE TERMINOLOGY FRAMEWORK presented in Chapter 2 provided the definitions that we use as a basis for the mobility management architecture. The architecture consists of functional structure and mobility management functionality to handle the *personal, role-figure*, and *terminal* mobility.

In the functional structure we introduce functional entities to handle these mobility types. These functional entities are introduced in Sec. 3.2. The specific functionality of these is based on the needs of the various mobility types. The functionality is presented by *mobility management procedures, mechanisms*, and *design rules*.

The *mobility management procedures* are procedures supporting the management functionality for the defined mobility types. *Mobility management mechanisms* are combinations of TAPAS basic support procedures and mobility management procedures. The *design rules* give several options for the implementation of the mobility management functionality. The design rules complement the requirement rules, presented in the terminology framework, by giving various possibilities to handle the mobility types.

The procedures and mechanisms have been implemented. Concerning the design rules, some have not been implemented.

The mobility management architecture is divided into three main parts. The handling of the *personal mobility* is presented in Sec. 3.3, the handling of the *role-figure mobility* is presented in Sec. 3.4, and the handling of the *terminal mobility* is presented in Sec. 3.5. Sec. 3.6 gives a discussion on some of the implementation issues.

3.2 The functional structure

Entities are needed for the handling of the movements of *persons* (*users* and *user sessions*), *role-figures*, and *terminals*. We introduce the following functional entities: *UserSessionBase, UserProfileBase, MobilityManager*, *MobilityAgent*, *UserAgent*, *VisitorAgent*, and *LoginAgent*, which are highlighted in Figure 3-1. This diagram shows the functional structure of our mobility management architecture. It is an extension to the computing architecture presented in Figure 1-5 with emphasis on mobility management. The introduced functional entities are databases and software components (role-figures) needed to handle the mentioned mobility types.

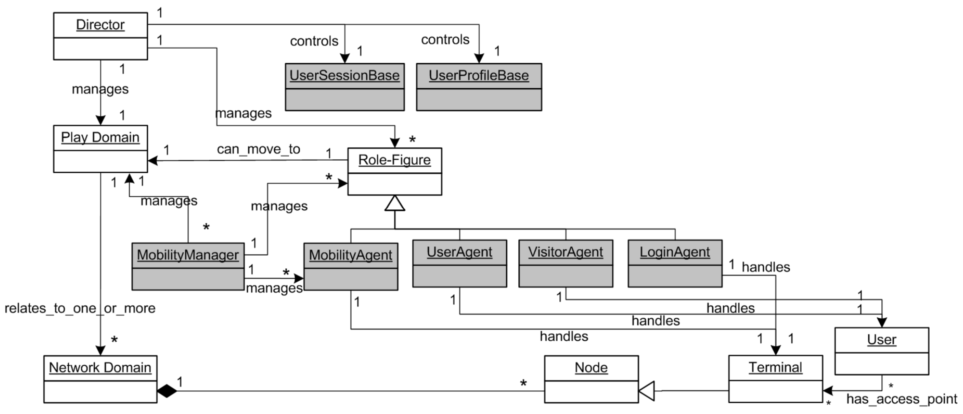


Figure 3-1 Functional structure of the mobility management architecture

In this figure we consider the following concepts from the computing architecture:

- *Director* manages one *play domain* and many *role-figures*
- *Play domain* relates to one or more *network domains* (the play domain is denoted as ***domain*** that may consist of ***sub-domains***)
- *Role-figure* can move to another *play domain*
- *Network domain* consists of *nodes*
- *Terminal* is one type of node
- *Users* have access points at *terminals*

We extend the computing architecture with the following functional entities:

- *UserSessionBase* is a database that corresponds to the *user session base* definition in the terminology framework. *UserSessionBase* is used to maintain the user session profile information.
- *UserProfileBase* is a database that corresponds to the *user profile base* definition in the terminology framework. *UserSessionBase* is used to maintain the user session profile information.
- *MobilityManager* is a software component (a supervisory object with a central role) that corresponds to the *mobility manager* definition in the terminology framework. *MobilityManager* is used to mange the mobility of role-figures, as well as, it manages the *mobility agents*. One or several mobility managers manage a play domain.
- *MobilityAgent* is a role-figure that corresponds to the *mobility agent* definition in the terminology framework. *MobilityAgent* is used to handle the mobility of a terminal.
- *UserAgent* is a role-figure that corresponds to the *user agent* definition in the terminology framework. *UserAgent* handles the interactions of a user at a home domain.
- *VisitorAgent* is a role-figure that corresponds to the *visitor agent* definition in the terminology framework. *VisitorAgent* handles the interactions of a user at a visitor domain.
- *LoginAgent* is a role-figure that corresponds to the *login agent* definition in the terminology framework. *LoginAgent* handles the login of users at a terminal.

Personal mobility will be handled by three role-figures: *LoginAgent* (one *LoginAgent* executes in every terminal), *UserAgent* (one *UserAgent* handles one user in a home domain), and *VisitorAgent* (one *VisitorAgent* handles one user in a visitor domain). *UserAgent* and *VisitorAgent* execute in the terminals associated with the user. In this diagram it is also shown that the domain director, which is a supervisory object in a TAPAS play domain, is responsible for the maintenance of the databases *UserSessionBase* and *UserProfileBase*. *Role-figure mobility* will be handled by *MobilityManager*. *Terminal mobility* will be handled by the *MobilityManager* and the *MobilityAgent* role-figures.

3.3 Personal mobility

3.3.1 Introduction

Personal mobility as defined in Chapter 2 comprises two types of mobility: user and user session mobility. The following assumptions are made:

- *User* is referred to by a *name* (user identification or user ID) and a *user profile*
- *User* is interacting with the system through a *user interface* (at a *terminal* through a *terminal interface*)
- Relationship between a user and a terminal is defined at login phase and controlled by a *LoginAgent*.
- A domain-based supervisory object (e.g. domain director) maintains the *user profiles* in the *UserProfileBase*
- A domain-based supervisory object (e.g. domain director) maintains the *user sessions profiles* in the *UserSessionBase*
- *UserAgent* and *VisitorAgent* control the user interactions with the system, and maintain a *user session* for each login phase
- *UserAgent* and *VisitorAgent* keep track of all role-figure instances that belong to a *user session*

The discussion on personal mobility is presented in five parts. In Sec. 3.3.2 we briefly discuss the login phase. Sec. 3.3.3 and Sec. 3.3.4 present the user mobility and the user session mobility, respectively. In these two parts the discussion is carried out in three steps: the concept, the mobility management procedures, and the design rules. Sec. 3.3.5 presents the databases for personal mobility. These are the *user profile base* and the *user session base*.

3.3.2 Login phase

The start-up of a user session is a central function in the personal mobility management. This user session can be terminal-based, application-based, or network-based user sessions. The login phase determines if a user session can be initiated or not. This phase also determines the access rights a user is assigned during the session, e.g. what services and file systems a user is allowed to access. In the mobility management architecture *LoginAgent* that executes in every terminal handles all the issues related to the login phase. This role-figure ensures that users can login to the terminals and can access their subscribed services.

During the login phase the service system must verify and accept the identification of the user in order to start a user session. An authentication process usually takes place during this phase. This process may various methods to authenticate users: password, pin code, magnetic card, or context-information (e.g. location information), etc.

The authentication process may be followed by the so-called authorization process, where a user may or may not be granted the rights to proceed with the session establishment. Even though a user is authenticated, it is not necessarily authorized to access certain services. The access rights of the user may be limited by time, payment, or security restrictions. As part of the session establishment process, certain information about users and terminals can also be registered, e.g. their location.

The login phase is not presented in detail to allow for different handling mechanisms in different service systems and application platforms, which may have quite different interpretations for the login phase. We limit our discussion on this phase to three main cases: *login at home domain*, *login at visitor domain*, and *login remotely to home domain*. Figure 3-2 gives an overview of the functionality of the implementation class of *LoginAgent*. It contains one public method to handle the login request, and this class must be *initiated* at every terminal.

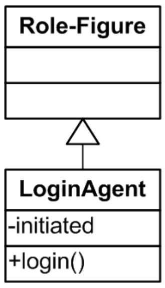


Figure 3-2 Implementation class of *LoginAgent*

Figure 3-3 illustrates three use cases handling the behaviour of *LoginAgent*. The use cases *loginHomeUser*, *loginVisitorUser*, and *loginHomeRemote* describe the following cases: *login at home domain*, *login at visitor domain*, and *login remotely to home domain*, respectively. The first two cases will be handled here, while the third login case (*login remotely to home domain*) will be presented in the following section as it is considered part of the user mobility.

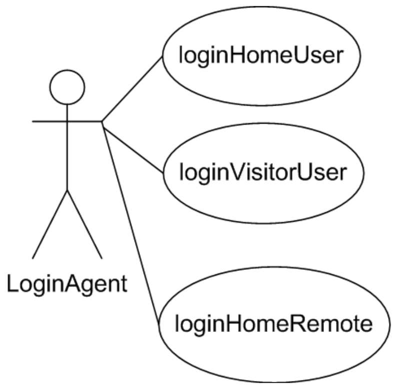


Figure 3-3 Use cases for *LoginAgent*

Two possibilities for user login to the system via the *LoginAgent* are shown in Figure 3-4, for the login at home domain, and Figure 3-5, for the login at visitor domain. In these two figures a user logs in to the system via the user interface with a *userID* and a *password* that arrives at the *LoginAgent* as a *login* request. Accordingly, *LoginAgent* sends a *loginRequest* to the director of the domain *domainDirector*. Until this point Figure 3-4 and Figure 3-5 are similar.

The director will decide on granting the required access to the user. In the case of the home domain user with user profile in *UserProfileBase*, a *UserAgent* is instantiated at the user's terminal, as shown in Figure 3-4. If there is no *user profile* in the *UserProfileBase*, a *VisitorAgent* is instantiated instead, as shown in Figure 3-5.

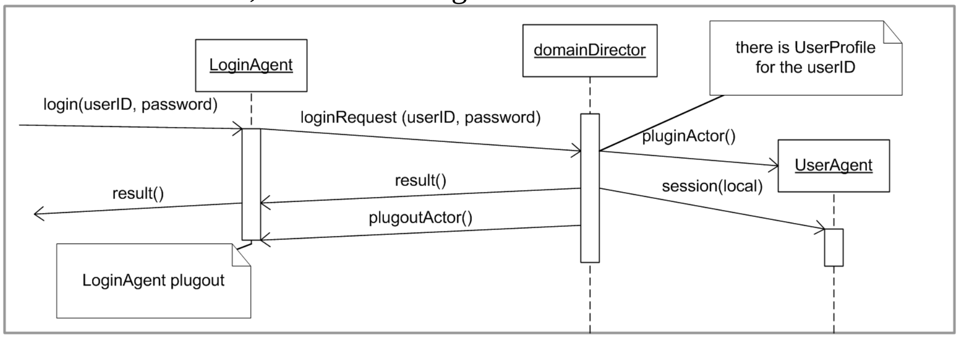


Figure 3-4 Sequence diagram for a *loginHomeUser*

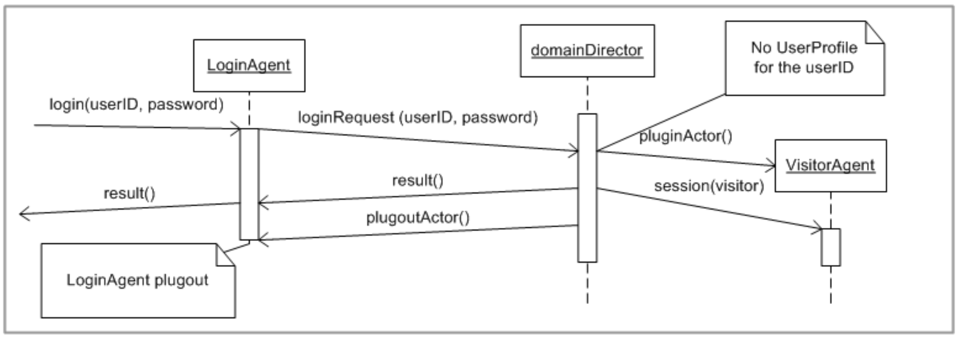


Figure 3-5 Sequence diagram for a *loginVisitorUser*

In both figures, the director proceeds with sending a *result* request to the user, which is forwarded to the user interface. The director also issues plug out to the *LoginAgent* from the user terminal. It is assumed that a *LoginAgent* plugs in again in the following cases: when the user session is terminated, when the user logs out, when the terminal is reset, etc. This ensures that there is always a *LoginAgent* that runs in the terminal.

The *UserAgent* and the *VisitorAgent* will then start a session. The session is started by the request *session* issued by the director, which indicates the access type granted to the user: *local* for home domain users, or *visitor* for visitor domain users. Note that the requests *pluginActor* and *plugoutActor* in the figures in this section are simplified, i.e. no arguments are shown.

3.3.3 User mobility

3.3.3.1 Concept

User mobility, as defined in the terminology framework, is the seamless access of the subscribed services at different user interfaces and terminals. It provides users with greater flexibility in terms of roaming among different terminals and domains.

Figure 3-6 illustrates the realization of the user mobility. This figure is an illustration of the user mobility between two TAPAS domains. It uses the definitions described in the user mobility related definitions in the terminology framework. The numbered steps give a logical order of actions to understand the concept of user mobility.

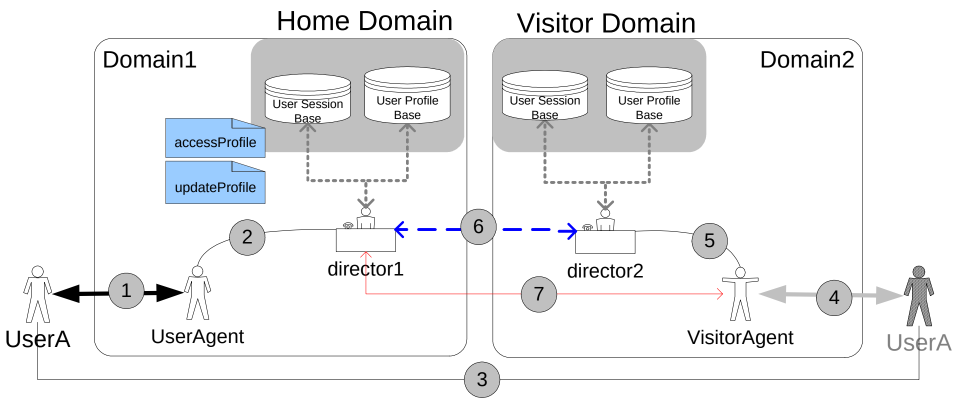


Figure 3-6 Illustration of the realization of user mobility

Assume the following scenario:

– In (1) *UserA* has logged in to its home domain, *Domain1*, and it has been assigned access to the subscribed services. This implies that a login phase has been carried out successfully (the user terminal and the *LoginAgent* executing in the terminal are not shown for simplicity).

– In (2) *UserAgent*, which is instantiated in the user's terminal, handles the user's interactions in its home domain. It contacts the director of the domain, *director1*, to get all the user profile information from *UserProfileBase*[1]. The requests *accessProfile* and *updateProfile* are issued to ask for and change the user's profile, respectively. *accessProfile* and *updateProfile* correspond to the *user profile access* definition and the *user profile update* definition in the terminology framework, respectively.

– In (3) the user will move and try to access the same set of services from a visitor domain, *Domain2*. When the user moves, its *UserAgent* in *Domain1* can be terminated, but it also can continue executing. Our handling of the user move will not be affected by the existing *UserAgent* in the home domain.

– Upon login phase, (4), this user is assigned a *VisitorAgent*, which is instantiated in the user's terminal. In this domain there is no user profile for this user, and hence it is granted a limited access to certain services – assigned usually to visitor users.

[1] It is possible at this point to resume suspended user sessions, as we will describe in the user session mobility in Sec. 3.3.4.

- to access home domain subscribed services, (5) *VisitorAgent* needs to indicate to director *director2* that a home domain access is required;
- If possible, a director-to-director negotiation in (6) can result in granting this type of access in (7).

As indicated by this logic, the login phase determines the access type a specific user is given, and therefore whether a *UserAgent* or a *VisitorAgent* is instantiated. Further details on remotely accessed home domain services, access rights, permissions, director-to-director negotiation, etc. are all left out to provide a general approach to user mobility.

User profiles should be highly customizable to reflect the personalization of services and applications. The user profile management is highly dependent on the used application platform and communication system. Moreover, user profile content can vary in different service systems. In the terminology framework we listed several information types that might be stored, e.g. user location, terminal-related data, subscribed services, access permissions, authorization constraints, service preferences, and setting attributes. User profile can also contain user specific data, e.g. favourite links and address book.

It is important to notice that Figure 3-6 illustrates the concept of the user mobility between two domains, possibly, in the simplest way. Further complexity may be added to handle issues related to domain security, personalized user environments, etc.

3.3.3.2 User mobility management procedures

The user mobility has been implemented as part of the TAPAS mobility management architecture prototype. We use the user mobility concept discussed above to develop the user mobility management procedures, i.e. we use the following assumptions:

- A domain is managed by a director that manages the databases for user profiles (also it manages the user session profiles)
- Users can either be in a home domain or in a visitor domain

Figure 3-7 gives an overview of the functionality of the implementation classes of *VisitorAgent* and *UserAgent.*

The parameters of these classes handle the access rights of the user, the user session, the child sessions of the role-figures, and the instantiated actors. The public methods in these classes handle the tasks associated with these role-figures. In *VisitorAgent*, there are methods for login, logout, session handling (*session* and *plugoutSession*), handling role-figures (*pluginActor* and *plugoutActor*), handling applications (*startApplication* and *stopApplication*), and a general method to return results. In *UserAgent*, similar methods exist to handle logout, sessions, role-figures, applications, return results, as well as methods for session mobility handling (*suspendSessionReq* and *resumeSessionReq*). These methods will be used in this subsection except the session mobility handling methods, which are part of the user session mobility of the next subsection.

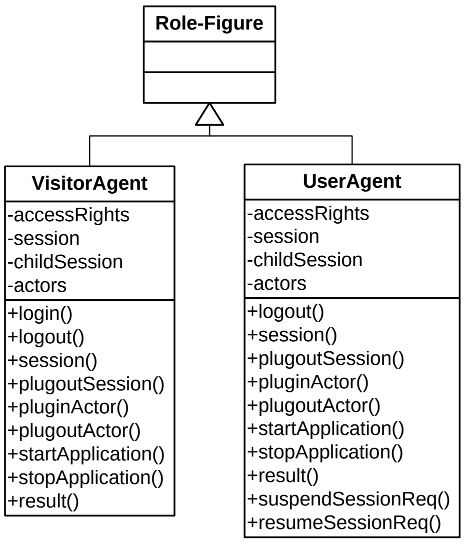


Figure 3-7 Implementation classes of *UserAgent* and *VisitorAgent*

Figure 3-8 illustrates different use cases handling the behaviour of *VisitorAgent* and *UserAgent*. These use-cases describe the mobility management procedures that will achieve the user mobility functionality. Each use case describes one mobility management procedure.

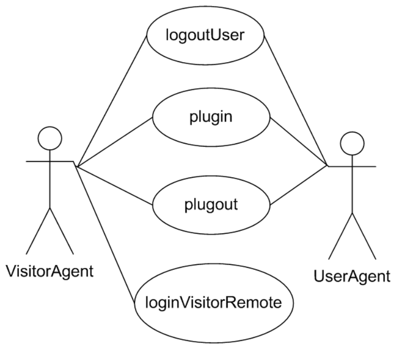


Figure 3-8 Use cases for user mobility

There are two possibilities for the remote login of a user to its home domain. The first possibility was mentioned in Sec. 3.3.2, which is the login remotely to home domain or *loginHomeRemote*. This is one possibility that is handled by *LoginAgent* and is presented in Figure 3-9.

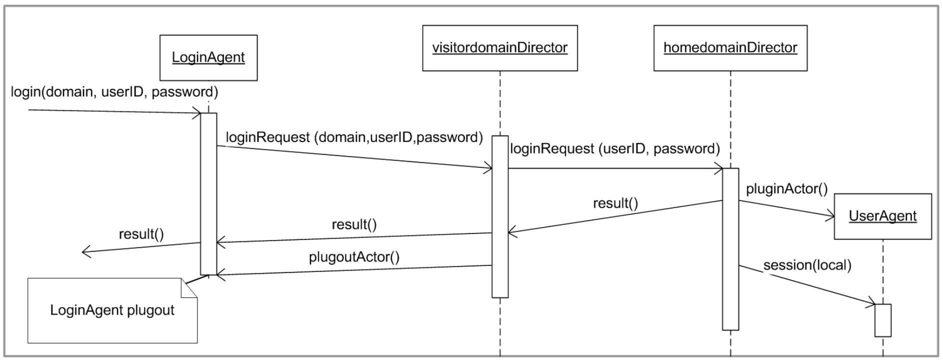


Figure 3-9 Sequence diagram for a *loginHomeRemote*

The other possibility corresponds to the login remotely to home domain that is handled by *VisitorAgent* and is presented in Figure 3-10.

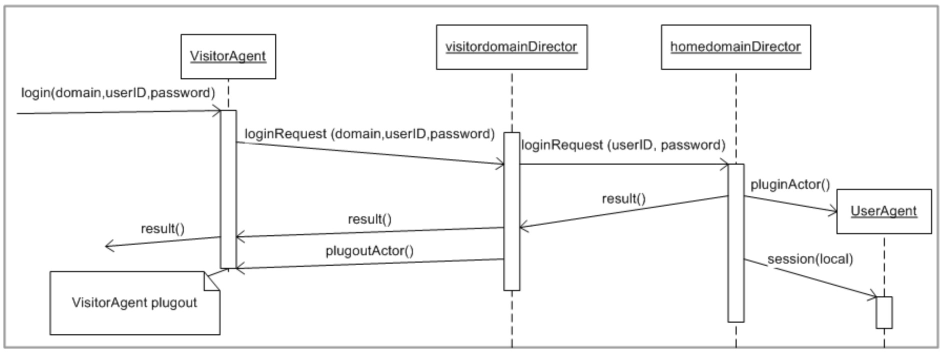


Figure 3-10 Sequence diagram for a *loginVisitorRemote*

In Figure 3-9 and Figure 3-10, the visiting user may be granted a remote login if it provides adequate information on its home *domain*, *userID*, and *password*. Following this *login* request is a director-to-director interaction and authentication process, which is indicated by the requests *loginRequest* and *result* between the directors of the two domains. This could result in providing the user with home domain access. As illustrated in Figure 3-9 and Figure 3-10, a successful remote login process results in plugging out the *LoginAgent* and the *VisitorAgent*, respectively. The reason is that a *UserAgent* must be instantiated to be capable of handling the user's interactions according to a home domain access type.

The logout phase can be part of the user move, and hence part of the user mobility. The logout phase can be handled by either *UserAgent* or *VisitorAgent*, and is shown in Figure 3-11 and Figure 3-12 for the local and the visitor cases, respectively. After receiving a logout request from the User Interface, *UserAgent* and *VisitorAgent* will plug out the user and its session. Both agents will be plugged out from the terminal after this.

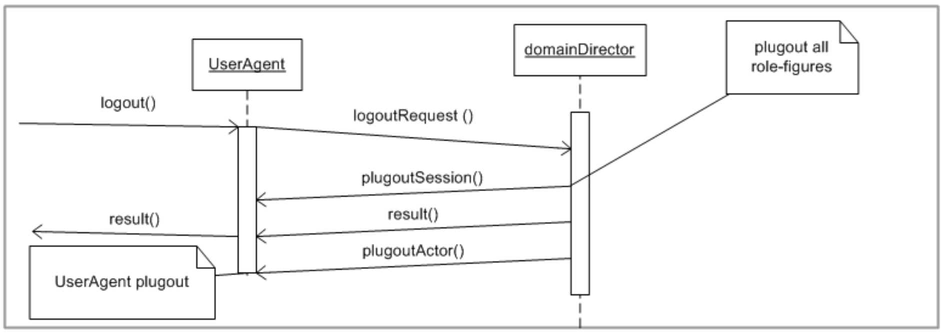


Figure 3-11 Sequence diagram for a *logoutUser* at home domain

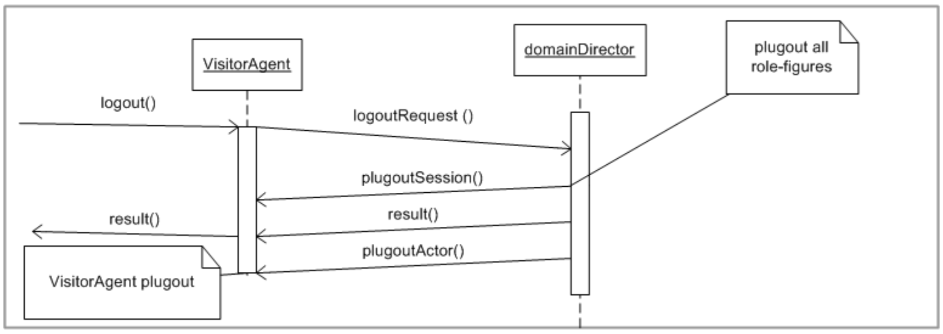


Figure 3-12 Sequence diagram for a *logoutUser* at visitor domain

In the proposed solution for handling the user mobility we introduced the *UserAgent* and *VisitorAgent* to be responsible for controlling the user interactions. Accordingly, if a user wants to start an application or terminate an application these agents must be informed. In our solutions we only allow the user to start or terminate applications through its assigned *UserAgent* or *VisitorAgent.* For example if an application started by a user requires a role-figure to be instantiated then *UserAgent* or *VisitorAgent* issue the plugin request for the role-figure that will execute the application. And similarly when an application is terminated, *UserAgent* or *VisitorAgent* will plugout the corresponding role-figure. Example of *plugin* and *plugout* sequence diagram is shown in Figure 3-13. *applicationStart* and *applicationStop* are two requests issued by the User Interface to start and stop an application. *UserAgent* responds by plug in and plug out the corresponding role-figure of the application.

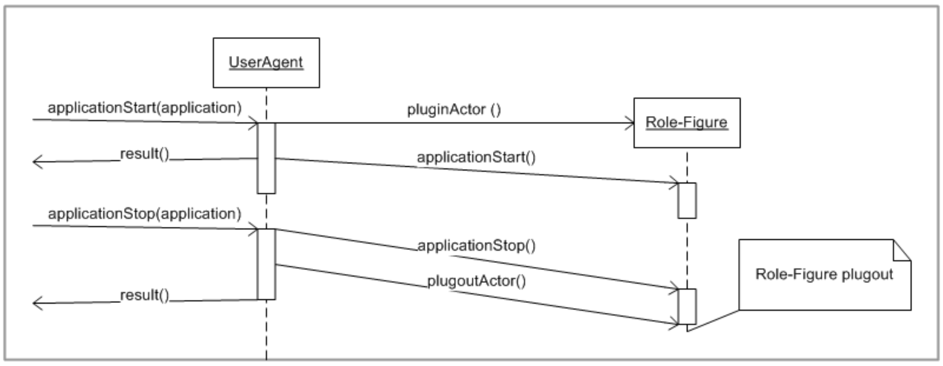


Figure 3-13 Sequence diagrams for a *plugin* and *plugout*

3.3.3.3 User mobility design rules

In this subsection we give design rules to show the overall behaviour of *LoginAgent*, *UserAgent*, and *VisitorAgent*, as well as, the handling of the user profile database with regard to user mobility. These design rules, *UD1-UD6*, complement the requirement rules for user mobility by showing different possibilities and options for the management functionality of the user mobility. Some of these rules have been implemented, while others have been encountered during the implementation and the decision was taken not to implement them.

These design rules will use "*MAY*" to signify that these rules are optional. *UD1* gives an option for the relationship between user and user profile. *UD2* shows how a user at visitor domain may interact. *UD3* and *UD4* are used as options on the user identification and user profile update. *UD5* and *UD6* give options for handling users at visitor domains.

UD1. A user, based on its service subscriptions, *MAY* have more than one user profile.
UD2. A user in a visitor domain *MAY* start interacting by sending its user ID in the login request.
UD3. A user in a visitor domain *MAY* be assigned a guest user ID.
UD4. A user in a visitor domain *MAY* be able to update its profile.
UD5. A user in a visitor domain *MAY* access home domain services from a visitor domain based on the policies applied in both domains.
UD6. A user *MAY* end its interactions with its home or visitor domain by sending a logout request to the supervisory object (a user can end its

interactions without a logout request).

Regarding our proposed solution presented by the user mobility management procedures we applied the following design rules: *UD3*, *UD4*, *UD5*, and *UD6*.

3.3.4 User session mobility

3.3.4.1 Concept

User session mobility, as defined in the terminology framework, is the re-instantiation and resumption of the user suspended sessions. Figure 3-14 illustrates how a *UserAgent* manages a user session, and how the director maintains the *UserSessionBase* as well as the *UserProfileBase*. In Figure 3-14 we use a domain of nodes and terminals.

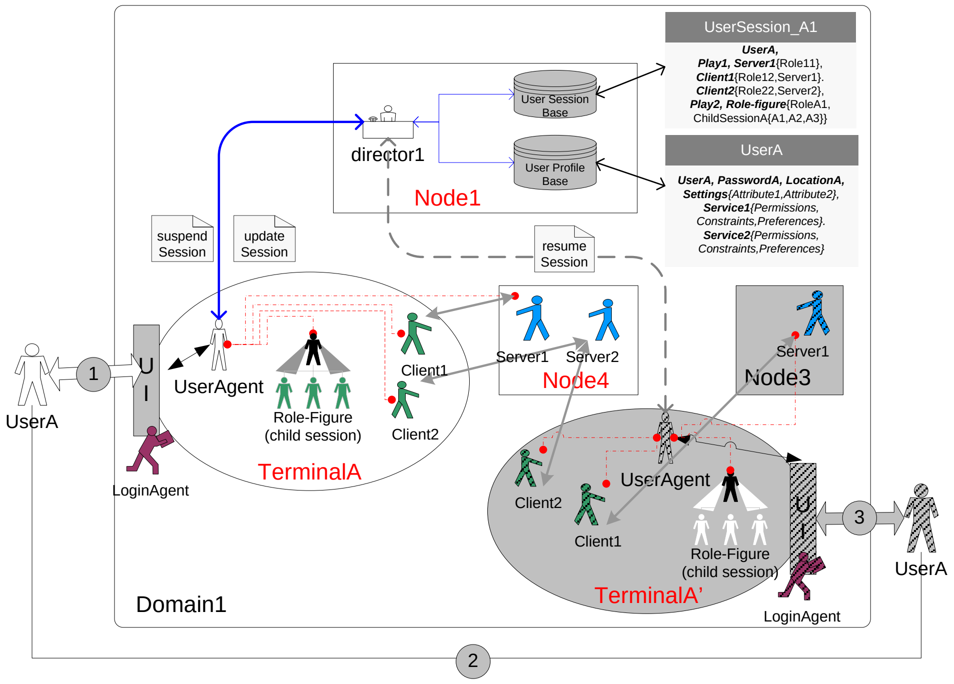


Figure 3-14 Illustration of user session mobility

Terminals execute *LoginAgents* and users can access these terminals at User Interfaces (UI). We assume *UserA* that access its subscribed services at two terminals, *TerminalA* and *TerminalA'*.

Figure 3-14 includes three main parts to show in logical order the suspension and resumption of a user session:

- Part (1): login phase (login *UserA* at *TerminalA*), which includes the establishment of a user session with role-figures, role-sessions, etc.
- Part (2): moving phase (*UserA* suspend its session and logs out from *TerminalA*. After that it moves to *TerminalA'* and resumes its session), where *TerminalA* and *TerminalA'* indicate that they both belong to the user *UserA*.
- Part (3): login phase (login *UserA* at *TerminalA'*), which includes the resumption of the suspended user session.

A user session is described by a user session profile, which is the representation of the user session information relevant to the maintenance of the user sessions. As such, a user session profile must contain full information on every instantiated role-figure, e.g. its data, role-sessions, and role-figure child-sessions. In Figure 3-14 we use dotted connectors between the *UserAgent* and other role-figures to indicate that they belong to one user session that is maintained by this *UserAgent*. E.g. the user session of *UserA* before the session mobility includes *role-figure* (with child session), *Client1*, *Client2*, and *Server1*. The figure shows an example of subscribed services: *Service1* and *Service2* defined by *Play1* (includes the Roles for *Client1*, *Client2*, and *Server1*) and *Play2* (includes the Roles for *role-figure* and its child session role-figures). *Server2* is considered a server maintained by the domain, and connected to *Client2*. Connectors in this figure indicate role-sessions between role-figures, e.g. *Server1* and *Client1*.

An example of a user profile and user session profile at the user profile base and the user session base are used (these profiles and the contents of these bases will be explained in Sec. 3.3.5).

The handling of user session mobility contains three main functions: updating, suspending, and resuming user sessions. A user session is updated by the *updateSession* request. This request may be sent regularly to the director to update the user session profile in the user session base. Also this request can only be sent when a user demands to update its user session. A user session may be suspended by the *suspendSession* request. *UserAgent* sends the *suspendSession* request to the director to indicate that this is a suspended session. *resumeSession* request is used to resume the sessions. *updateSession* and *suspendSession* correspond to the *user session update* definition, while *resumeSession* corresponds to the *user session access* definition in the terminology framework.

To resume a user session it is required to re-instantiate all the role-figures that have been maintained by the *UserAgent*, however certain role-figures (or their child sessions) may not be re-instantiated in the same manner as they have been used before the session suspension. In Figure 3-14 we indicate this by re-instantiating *Server1* in different node and re-instantiating a different child session for the used role-figure. These issues can only be addressed using the concepts and the management functionality of the role-figure mobility, which will be handled later in this chapter.

3.3.4.2 User session mobility management procedures

Figure 3-15 illustrates different use cases handling the behaviour of *UserAgent* with regard to user session mobility. We assume that only *UserAgent* can handle user session mobility (a visiting user does not have a maintained session). These use-cases describe the mobility management procedures that will achieve the user session mobility functionality. Each use case describes one mobility management procedure.

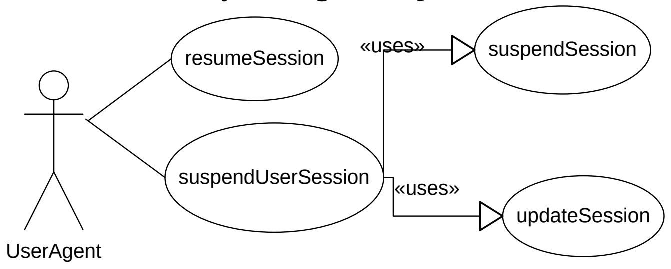


Figure 3-15 Use cases for user session mobility

Figure 3-16 describes a session suspension. A session suspension starts with a *suspendSessionReq* request from the user interface to the *UserAgent*, which accordingly sends a *suspendSession* request to all role-figures that have been plugged in by this *UserAgent*. These role-figures send descriptions of their execution status, in a *result* request. An example role-figure is shown in the message sequence diagram. The execution status of a role-figure is the full information that is needed to reconstruct the session, e.g. its data, role-sessions, and role-figure child-sessions. This specific information on individual role-figures is application and platform dependent. These role-figures are plugged out by the *UserAgent*. Any suspended session is registered in a *UserSessionBase* using the *updateSession* request, via the director.

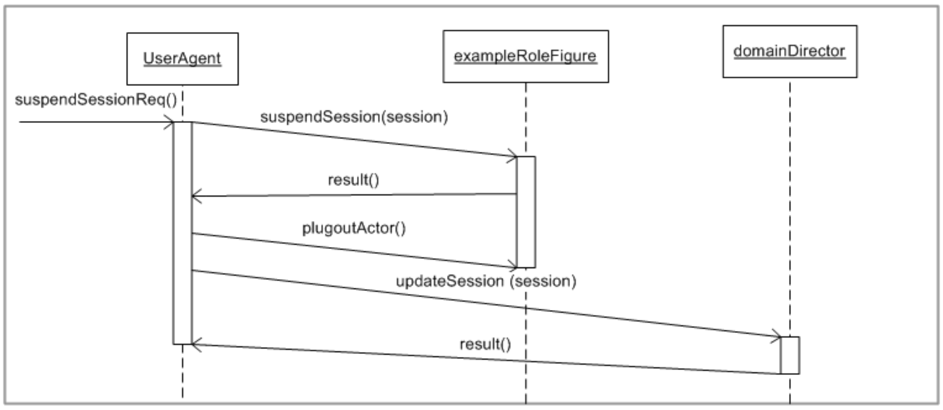


Figure 3-16 Sequence diagram for a *suspendSession* and *updateSession* use cases

Session resume is presented in Figure 3-17. Upon a new login of a user, who asks for a session resume by *resumeSessionReq*, the *UserAgent* will issue a *resumeSession* request to the domain director, which will return the session description to the *UserAgent*. *UserAgent* will manage the whole process of session reconstruction. The *UserAgent* will send different *pluginActor* requests to instantiate role-figures that are saved in the user session description. This process ends by sending a session request to these individual role-figures, with information on their execution status.

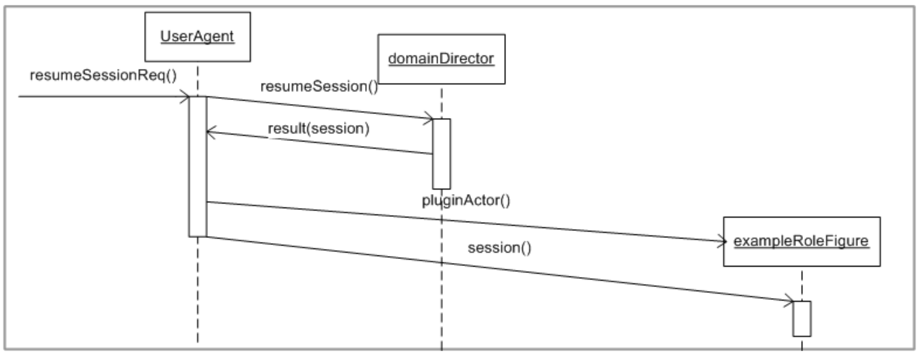


Figure 3-17 Sequence diagram for a *resumeSession* use case

3.3.4.3 User session mobility design rules

In this subsection we give design rules to show the overall behaviour of *UserAgent* and *VisitorAgent*, as well as, the handling of the user session base with regard to user session mobility. These design rules, *SD1-SD6*, complement the requirement rules for user session mobility, presented in the terminology framework. All of these rules have been implemented, except one rule that has been encountered and not implemented.

SD1 gives an option for the relationship between user and user session profile. *SD2* shows an option for the maintenance of role-figures child sessions. *SD4, SD5,* and *SD6* give options for handling visiting users.

SD1. A user *MAY* have more than one user session profile (in a domain it may be allowed to maintain more than one user session).
SD2. The user session profile *MAY* include information on the instantiated role-figures and their child-sessions within the user session.
SD3. Suspension and resumption of user sessions *MAY* be handled by the *VisitorAgent*.
SD4. A user in a visitor domain *MAY* be able to resume a session, and suspend a session.
SD5. If a user in a visitor domain suspends its session the *VisitorAgent MAY* update the user session profile of the user.
SD6. If a user in a visitor domain resumes a suspended session the *VisitorAgent MAY* access the user session profile of the suspended session.

Regarding our proposed solution for the user session mobility management we applied the following design rules: *SD2, SD4, SD5,* and *SD6.*

3.3.5 Databases for personal mobility

XML-based databases and XML-based requests can be proposed as an automated, modular, and extendable solution for the user session and the user profile bases. Figure 3-18 and Figure 3-19 show possible data structure models for the user session and the user profile bases, respectively.

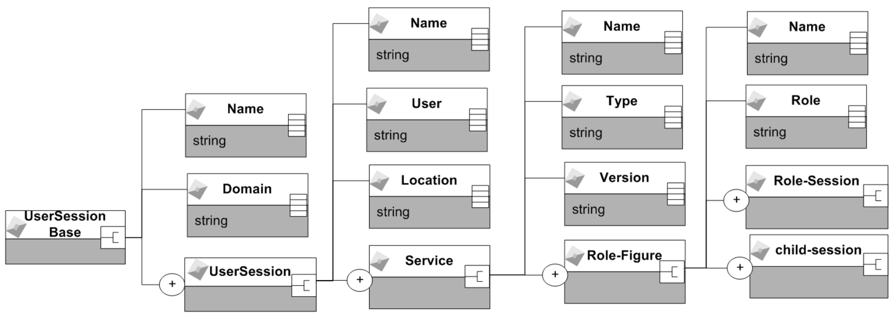


Figure 3-18 A possible data structure model for user session base

The graphical notation used in these figures is proprietary and it shows a possible XML schema for the two databases. Some parts of these data structures can be extended. A specific *UserSession* can be extended by adding several *Service*s, and a specific *Service* can be extended by adding several *Role-Figure*s. The *Role-Session* and *child-session* parts can be described in many different ways, e.g. every *role-session* and every *child-session* is described in an individual data structure. Other data structures, e.g. *Name*, *Domain*, *User*, *Location*, *Type*, *Version*, are modelled as strings. Similarly, in a user profile base a *UserProfile* have several *Service* parts, while *Setting* and *Preferences* can be described in different ways.

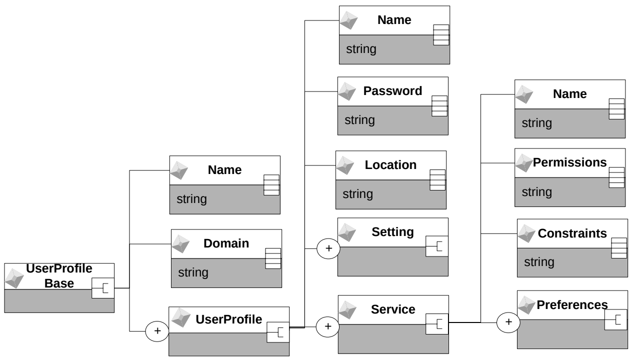


Figure 3-19 A possible data structure model for user profile base

Table 3-1 shows an example of *UserSessionBase* database that corresponds to the scenario presented in Figure 3-14.

Table 3-1 Example of *UserSessionBase* specified in XML

<USER_SESSION_BASE NAME="USBdomain1">		<VERSION>v1_2</VERSION>
<DOMAIN>domain1</DOMAIN>		<ACTOR_INSTANCE>
<USER_SESSION NAME="UserSession_A1">		<NAME>Role-figure</NAME>
<PROPERTY NAME="User">		<ROLE>RoleA1</ROLE>
<ID>UserA</ID>		<CHILD_SESSION>
<LOCATION>TerminalA</LOCATION>		<ACTOR_INSTANCE NAME="A1">
</PROPERTY>		<ROLE>RoleA11</ROLE>
<PROPERTY NAME="Play1">		<ROLE_SESSION>
<TYPE>Chat</TYPE>		<COOPERATOR>ActorA</COOPERATOR>
<VERSION>v1_1</VERSION>		</ROLE_SESSION>
<ACTOR_INSTANCE NAME="Server1" >		</ACTOR_INSTANCE>
<ROLE>Role11</ROLE>		<ACTOR_INSTANCE NAME="A2">
<ACTOR_INSTANCE>		<ROLE>RoleA12</ROLE>
<ACTOR_INSTANCE NAME="Client1">		<ROLE_SESSION>
<ROLE>Role12</ROLE>		<COOPERATOR>ActorA</COOPERATOR>
<ROLE_SESSION>		</ROLE_SESSION>
<COOPERATOR>Server1</COOPERATOR>		</ACTOR_INSTANCE>
</ROLE_SESSION>		<ACTOR_INSTANCE NAME="A3">
</ACTOR_INSTANCE>		<ROLE>RoleA13</ROLE>
<ACTOR_INSTANCE NAME="Client2">		<ROLE_SESSION>
<ROLE>Role22</ROLE>		<COOPERATOR>ActorA</COOPERATOR>
<ROLE_SESSION>		</ROLE_SESSION>
<COOPERATOR>Server2</COOPERATOR>		</ACTOR_INSTANCE>
</ROLE_SESSION>		</CHILD_SESSION>
</ACTOR_INSTANCE>		</ACTOR_INSTANCE>
</PROPERTY>		</PROPERTY>
<PROPERTY NAME="Play2">		</USER_SESSION>
<TYPE>Debug</TYPE>		</ USER_SESSION_BASE>
* An example of ***XML*** serialization of the **UserSession Base**, where UserA has a session named 'UserSession_A1' that takes part in two plays.		** UserA runs the following role-figures: Server1, Client1, Client2, Role-figure, A1, A2, and A3

Table 3-2 shows an example of *UserProfileBase* database that corresponds to the same scenario (the user profile for *UserA*, and the user session profile *UseSession_A1* in *Domain1*).

Table 3-2 Example of *UserProfileBase* specified in XML

```
<USER_PROFILE_BASE NAME="UPBdomain1">
 <DOMAIN>domain1</DOMAIN>
 <USER NAME="UserA">
   <PROPERTY NAME="Password">
     <VALUE>****</VALUE>
     <VALID>010106</VALID>
   </PROPERTY>
   <PROPERTY NAME="Location">
     <VALUE>local</VALUE>
   </PROPERTY>
   <PROPERTY NAME="Settings">
     <ATTRIBUTE NAME="BGColor">
       <VALUE>White</VALUE>
     </ATTRIBUTE>
     <ATTRIBUTE NAME="WSize">
       <VALUE>Large</VALUE>
     </ATTRIBUTE>
   </PROPERTY>
   <PROPERTY NAME="Service1">
     <PERMISSIONS>
       <VALUE>Owner</VALUE>
     </PERMISSIONS>
     <CONSTRAINTS>
       <VALUE>LocalAccess</VALUE>
     </CONSTRAINTS>
   <PREFERENCES>
       <VALUE>Empty</VALUE>
   </ PREFERENCES >
   </PROPERTY>
   <PROPERTY NAME="Service2">
     <PERMISSIONS>
       <VALUE>Temp</VALUE>
     </PERMISSIONS>
     <CONSTRAINTS>
       <VALUE>LocalAccess</VALUE>
     </CONSTRAINTS>
     <PREFERENCES>
       <VALUE>Empty</VALUE>
   </ PREFERENCES >
   </PROPERTY>
 </USER>
</ USER_PROFILE_BASE>
```

*** An example of ***XML*** serialization of the **UserProfile Base**, which includes the UserProfile for a single user, UserA.

*** UserA has a subscription to Service1 (specified by Play1), and Service2 (specified by Play2).

3.4 Role-figure mobility

Role-figure mobility, as defined in the terminology framework, is the movement of instantiated role-figures. The discussion on role-figure mobility will be carried out in four different parts. In Sec. 3.4.1 a general role-figure mobility scenario is presented. This illustrates the concept, and gives an overall approach to role-figure mobility. In Sec. 3.4.2, we discuss several issues regarding moving role-figures, mobility managers, and mobility strategy. In Sec. 3.4.3, we present implementation mechanisms for the role-figure mobility management. These mechanisms are based on choices taken on the mobility management procedures for moving role-figures and the applied mobility strategy. Finally, in Sec. 3.4.4 the design rules for role-figure mobility are presented.

3.4.1 General

A role-figure is moved by re-instantiating the role-figure in conjunction with its parts. Based on the discussion presented in the terminology framework, an instantiated role-figure comprises the following parts:

- *Role-sessions* with other role-figures
- Consumed *capabilities* in the node

- *Behaviour* described by a specification
- *Queue* of incoming messages
- Executing *methods* (or the role-figure active tasks)

These parts need be considered when the role-figure moves. Some of these parts must be moved with the role-figure to ensure that the moving role-figure will continue its execution after the movement. In the following sections we will handle in detail the first three parts, i.e. *role-sessions, capabilities,* and *behaviour*. We will briefly discuss the *queue* and executing *methods* and, later, will give general design rules to handle them in the mobility strategy design rules. Figure 3-20 illustrates role-figure mobility between two domains.

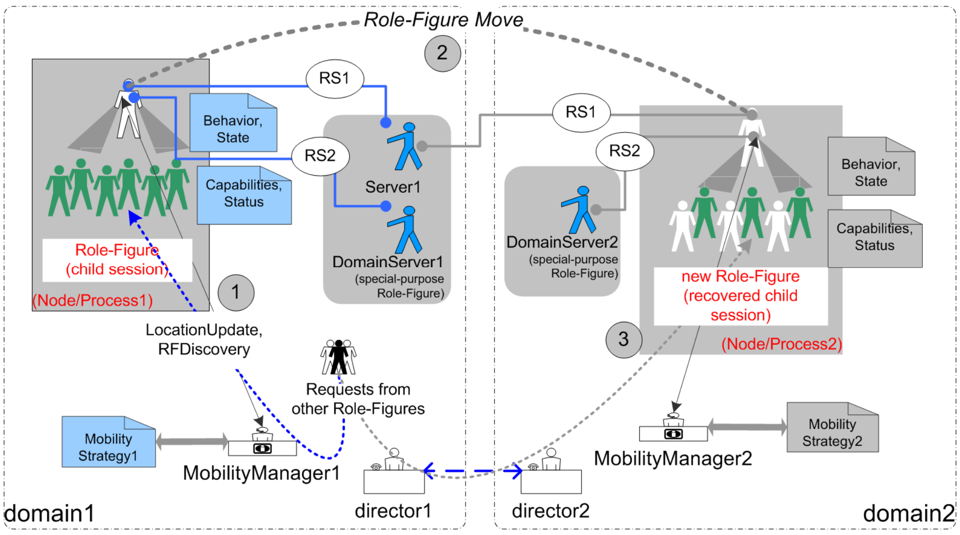


Figure 3-20 Illustration of role-figure mobility

In Figure 3-20 the role-figure, before its move, has behaviour that is at a current state, capabilities and status, two role-sessions (*RS1* with *Server1* and *RS2* with *DomainServer1*), and a child session that constitutes of several instantiated role-figures. We illustrate a movement of this role-figure from location *Node/process1* to location *Node/process2* (both are considered the role-figure's location or location-of-performance as we defined in the terminology framework). This figure includes three main parts to show in logical order a role-figure mobility scenario:

– Part (1): execution phase (*role-figure* executes in *Node/process1* at *domain1*).

– Part (2): moving phase (moving *role-figure*, with possible child-session consisting of several role-figures, from *Node/process1* at *domain1* to *Node/process2* at *domain2*).
– Part (3): execution phase (*role-figure* executes in *Node/process2* at *domain2* after the move).

In the terminology framework we stated that when a role-figure moves from one execution environment to another a supervisory object in the system is responsible for managing this move. In our proposed solution for the role-figure mobility we will use the mobility manager for this supervisory object. In Figure 3-20 it is *MobilityManager1* and *Mobility-Manager2*. We introduce three mobility management procedures to handle the mobility of the role-figure:

- *LocationUpdate*
- *RoleFigureMove*
- *RFDiscovery*

LocationUpdate procedure is initiated by a role-figure when it moves. This procedure results in sending the role-figure new location to the mobility manager. Using this procedure we make sure that mobility manager is always updated about the moving role-figure and its current location.

RoleFigureMove procedure handles the movement of the role-figure and its parts. Due to its crucial functionality, this procedure was also considered as a part of the support functionality of the core platform. In Sec. 1.4.2.2, we included this procedure in the actor basic support procedures and its invocation request was denoted as *rolefigureMove*. *RoleFigure-Move* procedure uses other actor basic support procedures to achieve the re-instantiation of a role-figure at a new location:

– *PlugInActor* (to instantiate a role-figure at a new location)
– *PlugOutActor* (to exit the role-figure at its previous location)
– *CreateInterface* (to create interfaces)
– *BehaviourChange* (to instantiate behaviour at a current state)
– *CapabilityChange* (to update the capability definitions)

RFDiscovery (*Role-Figure D*iscovery) procedure is initiated by other role-figures to get the location of the moving role-figure from the mobility manager. There are two different ways to use this procedure:

– Role-figures must initiate this procedure before every communication with a moving role-figure

- Role-figures may maintain a cache of addresses to store recently accessed moving role-figures, and only initiate this procedure when communicating with an unknown moving role-figure

These procedures and the requests used in these procedures will be explained later when we present our role-figure mobility management mechanisms in Sec. 3.4.3.

Mobility strategy is a set of domain specific rules and conditions that govern the role-figure mobility procedures. These sets of rules will be handled in detail in Sec. 3.4.2.3 and later in the design rules for role-figure mobility.

By moving role-figures certain parts may be irrecoverable. E.g. certain capabilities may not be available at the new location, or specific role-sessions are no longer relevant. In this scenario, for instance, the *RS2* at the new location has been recovered to connect to another *special-purpose* role-figure (*DomainServer2* that is available in *domain2*). Also, the moved role-figure child-session couldn't be fully recovered. Certain role-figures have been assigned different functionality and/or different set of capabilities.

3.4.2 Issues related to role-figure mobility

The simple mobility scenario, presented in the previous subsection, illustrated the main characteristics of the concept of role-figure mobility. However, many issues have not been covered and need further discussion. This subsection discusses issues related to the moving role-figure, the mobility manager, and the mobility strategy.

3.4.2.1 The moving role-figure

Role-session mobility

Role-session is the projection of the behaviour of a role-figure with respect to one of its interacting role-figures. A role-session between two role-figures means the agreement between the two role-figures on the terms of their interactions. As defined in the terminology framework, *role-session mobility* is the re-instantiation of the role-sessions of a moving role-figure. A role-session that cannot be re-instantiated mean that the communication with the role-figure referred to by this role-session is not possible.

Another concept that is related to the concept of role-session is the concept of interface. Interface is the means to gain access to object instances from other objects. Interfaces provide a role-figure with adequate information on how to reach other role-figures. Figure 3-21 illustrates how two role-figures interact with each other via a role-session. *RF_2* has a role-session *rs1* with *RF_1*, while *RF_1* has a role-session *rs2* with *RF_2*. The communication between these two role-figures, however, is realized via the actor objects through their interfaces, *i* and *j*.

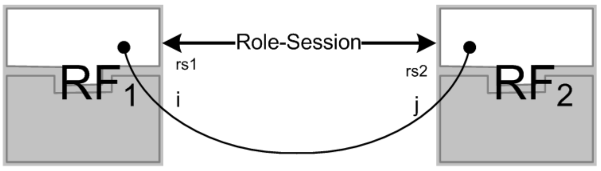


Figure 3-21 Illustration of role-sessions and interfaces

To re-instantiate the role-sessions of a moving role-figure we need to recreate their interfaces also. The special-purpose role-figures may introduce complexities in the role-session mobility. A role-session with a special-purpose role-figure may not be recreated after the move (a role-figure may move to another domain where such an interface is not allowed). However, it can be possible to find another special-purpose role-figure in the new domain that plays a similar role. In this case, the moving role-figure will create an interface to this special-purpose role-figure. Domains may have lists of special-purpose role-figures based on their functionality, e.g. name servers and capability managers. Mobility managers can use such lists to locate appropriate special-purpose role-figures for the moving role-figures. These issues of the interface recreation will be handled in the mobility management mechanisms and the design rules for mobility strategy.

To handle role-session mobility we need appropriate concepts for role-sessions, i.e. concepts for the instantiation, re-instantiation, and evolution of role-sessions. As these concepts have not been formulated within the context of TAPAS, we will only handle the recreation of the interfaces of a moving role-figure, and will not handle the re-instantiation of role-sessions of a moving role-figure.

Capability mobility

The reclamation of the consumed set of capabilities by a moving role-figure is denoted as *Capability mobility*. The discussion on the capability mobility requires a proper architecture that handles capabilities of the moving role-figure. Here we consider the TAPAS configuration management architecture (it was mentioned in Chapter 1 and explained in Appendix I). Similar to role-session mobility, certain capabilities of a moving role-figure may or may not be reclaimed. Here it is important to distinguish between two sets of capabilities for a moving role-figure:

- The required set of capabilities, which are specified in the specification of the role played by the role-figure.
- The requested set of capabilities, which are the capabilities consumed throughout the execution of the role-figure until the arrival of the move request.

The required set of capabilities must be available in the new location to instantiate the role-figure. The requested capabilities must be reclaimed to achieve the capability mobility, and hence the mobility of the role-figure. There are certain situations when some of the requested capabilities cannot be reclaimed.

Behaviour recovery

The role-figure behaviour is specified in an extended finite state machine specification, with states and state transitions. States are used to logical represent the evolution of the role-figure behaviour. The role-figure behaviour evolves by state transitions, in which the role-figure processes the arriving messages. A state transition is triggered by the consumption of a message that is defined as a triggering event at the current state.

When a role-figure moves its behaviour must be recovered to ensure that the role-figure will continue its execution after the move. Role-figure behaviour recovery has two elements: to capture the state information before the move, and to recover the state information after the move.

A role-figure operates on the main memory and uses the heap for processing data. Role-figure behaviour recovery does not mean to recover all sorts of data in the heap, or to recover the access rights and information of the local file system. These are only relevant at the role-figure’s current location, and have no meaning in another location.

The specification of the role-figure behaviour plays an important role in achieving role-figure behaviour recovery. The manner how a specification is written can highly affect the possibilities for the behaviour recovery and the state information mobility. We consider a set of states in the specification of the role-figure behaviour where a move is allowed. We call this set as *stable states*. This means that the behaviour change, and consequently the role-figure mobility, is only allowed when the role-figure behaviour current state is a state in this set.

Further elaboration on the issues regarding behaviour recovery will be handled in Chapter 4 and Chapter 5.

3.4.2.2 The mobility manager

The role of the mobility manager

The mobility manager needs to know which role-figures it is handling. Also role-figures need to know which mobility manager is handling their mobility. We consider two cases for relating role-figures to mobility managers:

- A role-figure is required to establish a role-session with mobility manager during its plug in
- A role-figure is not required to establish a role-session with mobility manager during its plug in, but may be instructed to do so later

In the first case, we require in the specification of the role played by the role-figure to connect this role-figure to a mobility manager. The second case allows the connection with a mobility manager to be carried out after the plug in. The difference between these two cases is that the first case does not allow role-figures to be instantiated without the connection with the mobility manager, while the second case allows that. Also, in the first case the role-figure is connected with the mobility manager of the domain or the sub-domain where the role-figure executes, while in the second case the role-figure may be connected with any specific mobility manager.

A mobility manager, in the first case, is responsible for all role-figures in its domain or sub-domain, while a mobility manager, in the second case, is responsible for some of these role-figures.

The organization of mobility managers

Several mobility managers may be needed within a single domain to increase the reliability and simplify the management of the domain. Several mobility managers in a single domain can cooperate to manage different sets of role-figures, distribute responsibility over different network domains, contain backup information, etc. For seek of simplicity, we assume a simple case; a domain is decomposed into sub-domains and each sub-domain is managed by one mobility manager. To delegate responsibility between mobility managers leads to a hierarchy of such managers.

Mobility manager to mobility manager relationship and their communications, e.g. inquiring about role-figures, are defined. In Figure 3-22 an example of a general mobility manager hierarchy is sketched. The relationships between these mobility managers determine the interactions and communications between these mobility managers. These relationships are denoted as: *cooperate*, *supervise*, *control*, *update*, and *co-supervise*. Note that several mobility managers may cooperate to supervise the same sub-domain, MM_1 and MM_{11}. Also several mobility managers, MM_{31} and MM_{32}, may supervise one mobility manager, MM_{321}.

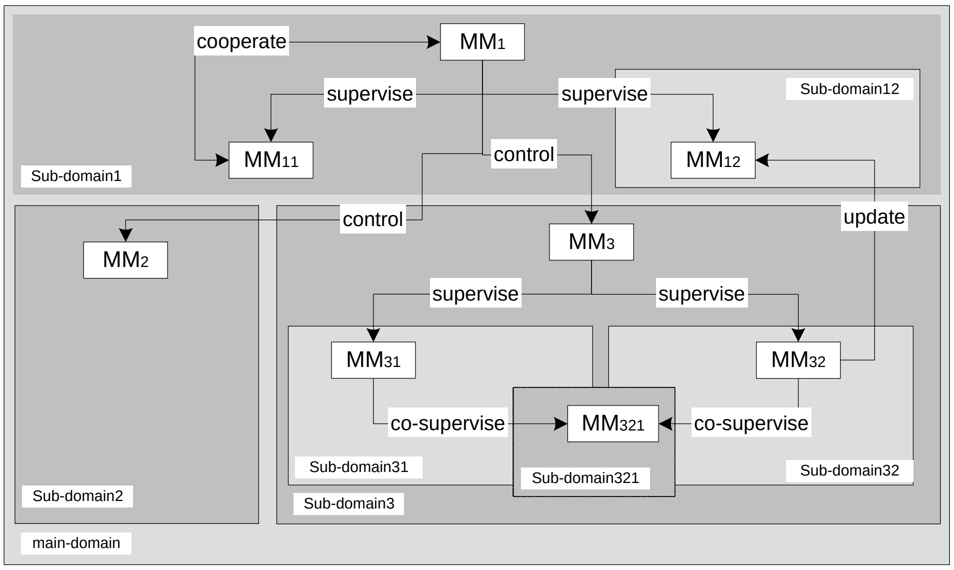


Figure 3-22 Example realization of a multi mobility management hierarchy

Following is a brief description of these relationships (the detailed description and realization of these relationships are not handled in the thesis):

- *Cooperate* stands for the collaboration and assistance among mobility managers to achieve certain goals. It may be used to split responsibility in large domains.
- *Supervise* means that a mobility manager supervises the operations of another mobility manager. This provides a general coordination and synchronization between the sub-domains.
- *Control* relationship allows for the close management of mobility managers own functionality and databases. This relationship is stronger than supervise, and indicates a full power on decision-making process.

– *Update* makes it possible to regularly update a mobility manager about the mobility aspects of another sub-domain. This may be used to assure security, or to realize a composite business model.
– *Co-supervise* is meant to provide a more reliability in supervising mobility managers of a highly dynamic or critical environment. It is also beneficial in cases of shared resources and infrastructure that necessitate a high degree of coordination.

There are several ways to inform a role-figure about the mobility manager. One possible way can be an initial setting for mobility managers in a domain (e.g. via a URL). However, sometimes it is needed to apply certain mechanisms to advertise and discover mobility managers. In Figure 3-23 we show two examples to advertise mobility managers.

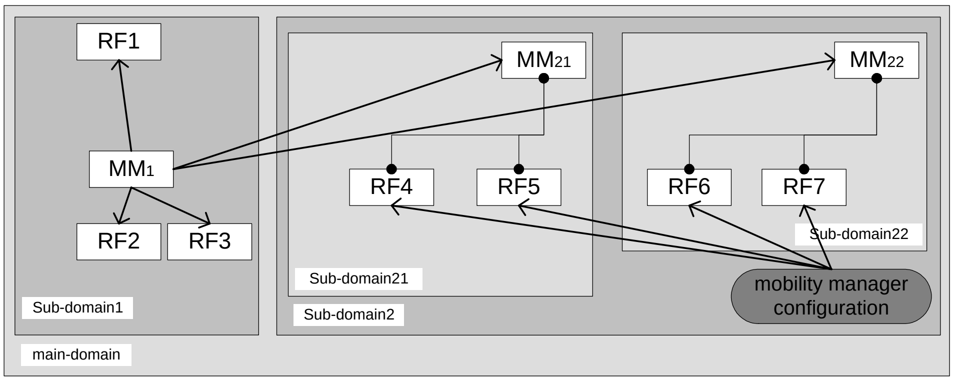


Figure 3-23 Advertisement examples of mobility managers

– In *Sub-domain1* the mobility manager broadcasts information regarding its availability to:
  ▪ All role-figures in its sub-domain (*RF1*, *RF2*, and *RF3*)
  ▪ Other mobility managers in its domain (MM_{21} and MM_{22})
– In *Sub-domain2* mobility manager configuration file is sent to role-figures in the sub-domain (*RF4*, *RF5*, *RF6*, and *RF7*). This configuration informs role-figures about their corresponding mobility managers. Role-figures obtain this file:
  ▪ During their plug in phase
  ▪ After their plug in phase if they request it

The lists of role-figures and mobility managers in a domain or sub-domain are maintained by the director of the domain. Mobility managers can have similar lists that may be used when a mobility manager resets.

Finally, there are several issues regarding mobility managers that must be resolved in order to achieve the role-figure mobility. Some of these issues are listed here:

– In case a role-figure moves between two domains or sub-domains, which have different mobility managers, how will this role-figure be registered at the new mobility manager?
– What are the alternatives for instantiating a moving role-figure that cannot establish a role-session with a mobility manager?

Such questions need to be addressed in the comprehensive set of rules that controls the mobility procedures, which we call the mobility strategy.

3.4.2.3 The mobility strategy

Mobility strategy is a set of domain specific rules and conditions that govern the role-figure mobility management procedures. The following gives a list of issues that will constitute the rules and conditions of the mobility strategy:

(a) Discard or handle the content of the message queue of the moving role-figure.
(b) Forward the content of the queue to the newly instantiated role-figure with or without further requirements on the arrival (i.e. order of arrival) and loss (i.e. forwarding with confirmation.)
(c) Allow or disallow two replicas of the same role-figure to function simultaneously, while the mobility process is handled.
(d) Keep or destroy replicas of the role-figure as it moves.
(e) In the case of *keeping* the replicas of a moving role-figure, how to forward the content of the queue to the moving role-figure, is it to the next one in the chain of replica or to the latest moved-to location.
(f) What is considered as an atomic event? Is it the whole sequence of events in the move process, or just a plug in request? This can result the role-figure to be totally suspended while it is moving.
(g) Priorities among strategy rules, so if there is more than one executable rule in the strategy, which one is to choose.
(h) What if a mobility strategy is not available in a domain?
(i) Many rules may be specified to cope with the question of special-purpose role-figures, e.g. if the moved role-figure cannot connect to a special-purpose role-figure.
(j) Many rules may be specified on the construction of role-sessions in general, how strict it should be, and what if certain role-sessions cannot be constructed.

(k) Many rules may be specified on the reclamation of capabilities, and what if certain capabilities cannot be reclaimed.

(l) Many rules may be specified on the re-instantiation of behaviour specification, and what if the current state cannot be recovered.

(m) Certain rules may be specified to handle the active tasks of a role-figure, whether to discard them or to wait until they execute. Active tasks of a role-figure may be remote method calls and local computations. Here we define a mobility management procedure to be either *preemptive* (to stop active tasks) or *non-preemptive* (to allow active tasks to execute).

Mobility strategies may be specified in a program-like manner, as a set of algorithms, or in a machine understandable format. These forms of specifications may be routines of a programmable language, algebraic expressions, or extensible notations, such as XML.

In the case of a moving role-figure between two domains or sub-domains, which possibly have two mobility strategies, it is important to define relationships between the strategies as well. The reason is that a mobility manager of one of these domains or sub-domains only accomplishes part of the job. The other part is accomplished by the other mobility manager. If the other mobility manager has a different set of rules in its mobility strategy then conflicts may occur. To resolve this issue we may apply the relationships between mobility managers that we handled in Sec. 3.4.2.2. For example role-figure mobility handled by two mobility managers with a *control* relationship between them may give a higher precedence to the rules of one mobility strategy over the other. Also, the mobility strategies of two mobility managers with a *co-supervise* relationship between them may require that these two strategies be exactly the same. This issue is a very interesting one, and could be handled in future development of our mobility management architecture.

3.4.3 Role-figure mobility management mechanisms

As we have mentioned earlier, the role-figure mobility is a special type of mobility and its management functionality encounters many issues. These issues are complex and dependent. Mechanisms are needed to deploy the presented role-figure mobility management procedures, and resolve those issues. Based on the discussions in Sec. 3.4.1 and Sec. 3.4.2, we assumed a general case of role-figure mobility that is controlled and managed by a mobility manager or several mobility managers. The mobility manager in this general case plays a central role so that every moving role-figure must be supervised by this mobility manager. This case turned out to be very restrictive and introduced a lot of traffic between the nodes. During the implementation and the experimentation of the different role-figure mobility scenarios, e.g. within the same node, within the same domain, among two domains, etc., we investigated the possibility of simplifying the role of the mobility manager. We encountered that the role of mobility manager can be simplified and associated with the moving role-figure.

We propose four implementation mechanisms to achieve role-figure mobility. They are denoted role-figure mobility management mechanisms, or RMM1 (the centralized mechanism), RMM2 (the proxy mechanism), RMM3 (the locator mechanism), and RMM4 (the persistent mechanism).

RMM1 implements the general case. RMM2, RMM3 and RMM4 introduce different mechanisms that simplify the role of the mobility manager. The mobility strategies have been incorporated as part of the role of the mobility manager. The role-figure mobility management mechanisms will take choices and decisions on the issues discussed in the mobility strategy. These mechanisms are specified using sequence diagrams. The abstraction will be kept at a high level, so that sequence diagrams don't get complex. The emphasis is on the interactions between the main components of the architecture.

In the role-figure mobility management mechanisms we will use the following invocation requests, as explained in Sec. 1.4.2.2:

- *pluginActor* (name, location, behaviour, capability)
- *plugoutActor* (name)
- *createInterface* (interfaces)
- *behaviourChange* (behaviour, state)
- *capabilityChange* (capabilities)
- *rolefigureMove* (location)

The role-figure mobility management mechanisms will also use the following invocation requests to initiate the *LocationUpdate* and *RFDiscovery* procedures mentioned earlier:

- *locationUpdate* (name, location), which is used to update the location of a role-figure
- *rfDiscovery* (name), which is used to get the location of a role-figure

Other requests will be used, and will be explained when they are used. The detailed description and semantics of these requests, as well as the corresponding public methods, are not included in the thesis.

3.4.3.1 RMM1: the centralized mechanism

The first mechanism is based on a centralized mobility manager, as shown in Figure 3-24. A role-figure responds to a move request, denoted *rolefigureMove* (location), by re-instantiating itself at the new location specified in this move request. The role-figure will transfer all the necessary information about its current capability set, role-sessions, behaviour to this newly instantiated role-figure, and update its location.

Figure 3-24 shows an example of a mobile role-figure (denoted *RF1-L1*) with the following information; it executes at location *L1*, performs according to behaviour specification *Beh* (denoted *RF1.beh = Beh*), its current state is *cState* (denoted *RF1.beh.st = cState*), has a capability set *Cap* (denoted *RF1.cap=Cap*), and has two interfaces corresponding to role-sessions *R1* and *R2* (denoted *RF1.int* = (*RS1*, *RS2*)). This role-figure will move to another location, which is L2. The locations L1 and L2 are abstraction of the role-figure executing environments.

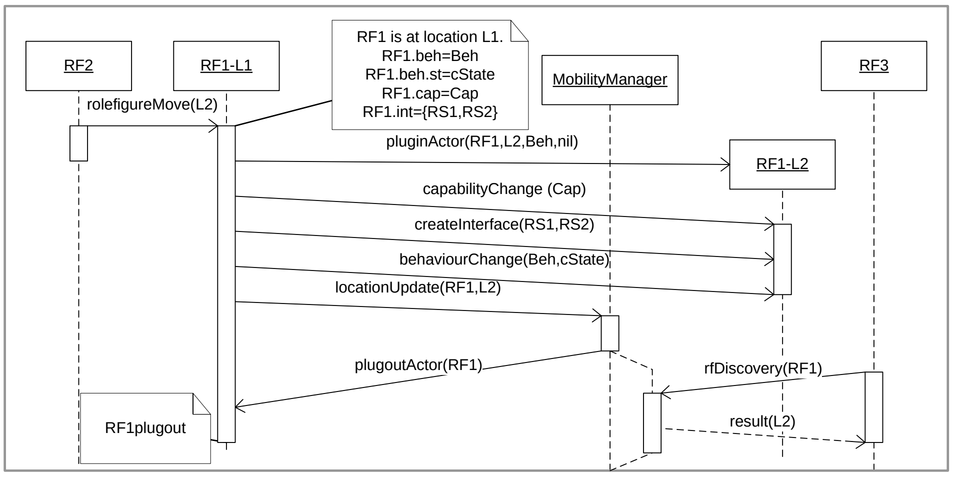


Figure 3-24 RMM1

Upon receiving a *rolefigureMove* (L2) from RF2 the role-figure starts the move to location L2. It initiates a *pluginActor* (RF1, L2, Beh, nil) request to plug in another role-figure at location L2, with behaviour Beh, and for simplicity no required set of capabilities. A role-figure is instantiated at L2 (denoted RF1-L2). The mobility mechanism continues with initiating the following requests: *capabilityChange* (Cap), *createInterface* (RS1, RS2), and *behaviorChange* (Beh,cState) to update the capabilities, role-sessions, and the behaviour of the role-figure at L2, respectively.

RF1-L1 then sends a *locationUpdate* (RF1, L2) request indicating the change of its location to the mobility manager, which plugs out the role-figure from *L1*. Other role-figures (e.g. RF3) aiming at communicating with the moving role-figure should send a *rfDiscovery* (RF1) request, and wait for a *result*(L2) in order to get the location of the role-figure. In this mobility mechanism, the discovery request must always be sent before any communication with a moving role-figure.

<u>*Advantages of RMM1*</u> To increase the robustness of this mechanism it is only needed to ensure that the centralized mobility manager is robust. It suits service systems that don’t require role-figures to move frequently. Also, role-figure discovery request can be sent to the mobility manager unaware of the fact that the role-figure is moving. A moving role-figure may signal the start of its movement, so that the mobility manager can mark this role-figure with a special flag, e.g. *moving*. By this manner the mobility manager delays sending information about the role-figure’s location until it is moved.

<u>*Disadvantages of RMM1*</u> This mechanism may imply heavy processing load for the mobility manager if there are many mobile role-figures with many movements. If the number of role-figures assigned to a mobility manager increases, or their movement becomes more frequent the performance of the mobility manager degrades. Also, as it is based on a centralized supervisory object a possible fail of the mobility manager would stop and role-figure mobility.

3.4.3.2 RMM2: the proxy mechanism

This mechanism is called the *proxy* mechanism as one role-figure acts as a proxy to a mobile role-figure. Basically, the first instance of a mobile role-figure acts as its proxy, and therefore it is not plugged out when this role-figure moves. This instance will handle all the requests of the role-figure, and forward them to the location where the role-figure moves to.

Figure 3-25 illustrates an example where a role-figure (RF1-L1) moves to a new location (RF1-L2), and then moves to another location (RF1-L3). It follows the same request used in RMM1 to initiate a move from L1 to L2. It uses *pluginActor* request to plug in a role-figure at L2. It uses the requests *capabilityChange* (to update capabilities Cap1), *createInter-*

face (to create the role-sessions (RS1, RS2)), and *behaviorChange* (to change the behaviour to (Beh1, cState1)) in locations L2. The move from L2 to L3 may be described in a similar manner.

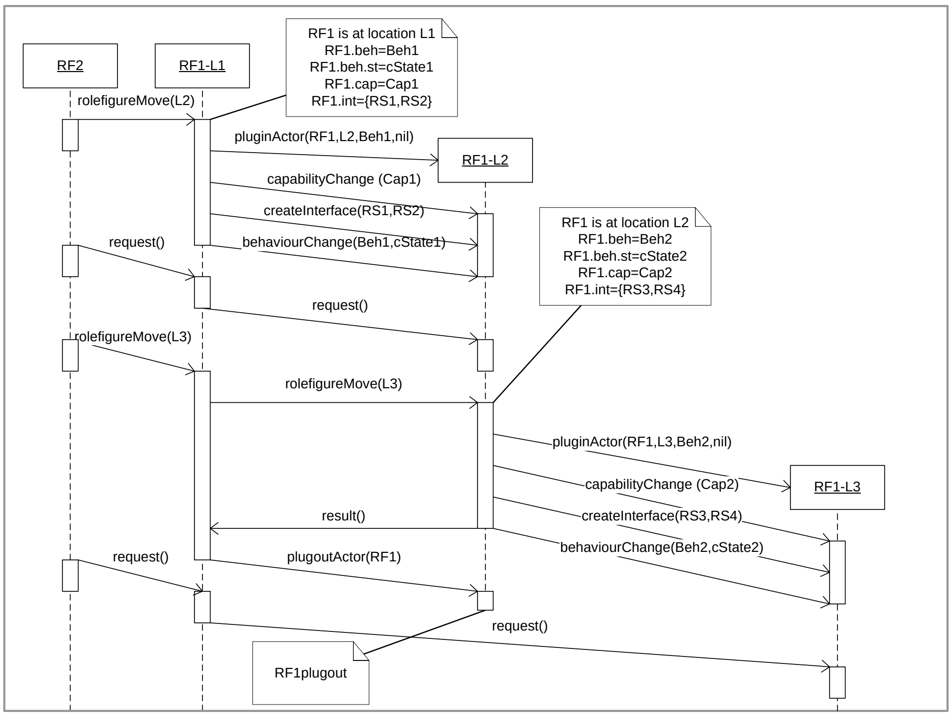


Figure 3-25 RMM2

When the role-figure RF1-L2 moves from L2 to L3 it performs the move and then RF1-L1 uses *plugoutActor* request to plug out the role-figure at L2 (this ensures that only a single replica of the role-figure exists besides its proxy). To show how other role-figures can communicate with this role-figure, RF2 sends a request to the role-figure proxy. The role-figure proxy, in this case, is the first instance of the role-figure, or RF1-L1. This proxy forwards the request it receives to RF1-L2 and to RF1-L3, based on where the role-figure exists at a given point. Also, Figure 3-25 demonstrates how other role-figures in the architecture are unaware of the movement itself.

<u>*Advantages of RMM2*</u> This mechanism does not rely on a centralized mobility manager as it was in RMM1. It suits service systems that are highly distributed, so the location updates don't need to be centrally maintained. It can support a large number of role-figures – as no role-figure discovery is required.

Disadvantages of RMM2 The scalability of this mechanism depends on the distribution nature of the architecture. If role-figures need to move among nodes that are at large distances, long delays may be generated. At the same time role-figures can only send requests to the proxy object, so there will always be an additional traffic in the network.

3.4.3.3 RMM3: the locator mechanism

This mechanism is based on dedicating a reliable entity to keep an updated role-figure location, while allowing the role-figure itself to move. We will call this entity as the role-figure's *locator*. Assume *role-figure locator* is realized by a special purpose object in the service system, which is a light weight implementation with very limited functionality. Each mobile role-figure will have a single *locator*. Other role-figures communicating with the mobile role-figure can obtain the moving role-figure's location from its locator. Figure 3-26 demonstrates the main aspects of this mechanism in an example.

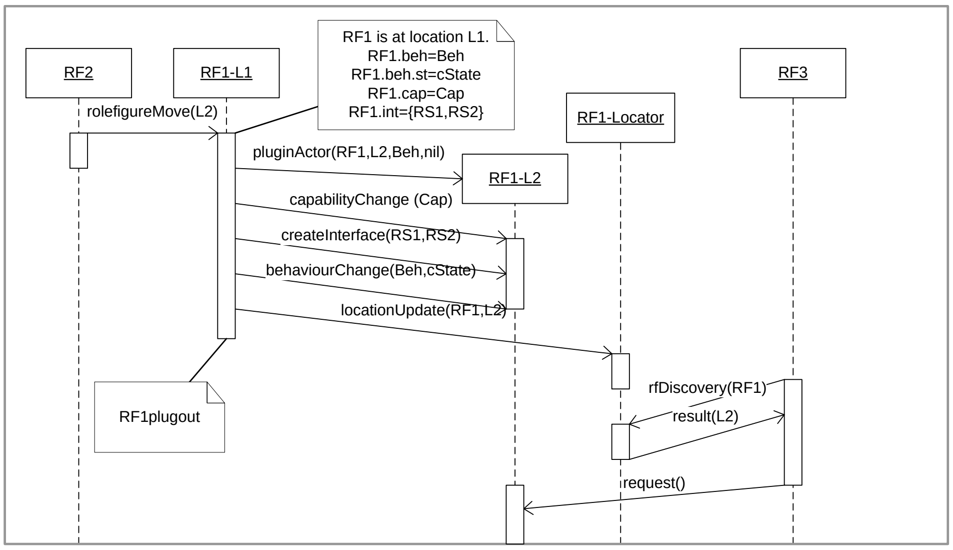


Figure 3-26 RMM3

Figure 3-26 uses exactly the same requests as in the example of RMM1. The only difference is that the mobility manager is substituted with the locator object. RMM3 differs also slightly from RMM2. While RMM2 maintains a fully functioning instance of the mobile role-figure to act as proxy, RMM3 creates only a *locator* object for it.

Advantages of RMM3 This mechanism suits service systems with large number of mobile role-figures with frequent moves.

Disadvantages of RMM3 The disadvantages of this mechanism are similar to the disadvantages of RMM2. Additionally, as locator objects get updates of the mobility of the role-figure only after the move, they may give non-updated location information while a role-figure is moving.

3.4.3.4 RMM4: the persistent mechanism

This mechanism is based on using a role-figure *locator* to get updated information about the mobile role-figures. Additionally, moving role-figures don't plug out their instances at previous locations. All instances of a mobile role-figure should exist even after receiving and performing a move request (be persistent over the lifespan of a role-figure). These instances of the mobile role-figure, however, stop executing any tasks. They only forward requests, which exceptionally arrive to them, to where the mobile role-figure exists at a given point. Another important aspect, when a role-figure is moved it will free up all resources it has seized, e.g. its capabilities. This increases the efficiency of the solution.

Figure 3-27 demonstrates this mechanism. It uses a combination of the requests used in the examples for RMM2 and RMM3. In this figure RF1-L1 executes in location L1, and its location is available in its locator RF1-Locator. RF2 issues a move request to RF1-L1 to move to RF1-L2. The most interesting part of this example is RF3. This role-figure gets the location of the mobile role-figure RF1 from its *locator*, while RF1 is moving. RF3 keeps sending requests to RF1's original location. This method makes this communication reliable by allowing all instances of RF1 to exist after the movements. The different instances of RF1 get a location update from the locator once a new movement is performed.

Advantages of RMM4 This mechanism can adapt to the highly dynamic network topologies, for instance ad hoc networks. In these networks parts of the network may split up, connect, disconnect, or be created or isolated dynamically. Such situations raise the issue of reliability in terms of the mobility manager maintained database, role-figure instantaneous movement status, and deliverability of requests to mobile role-figures. As such, it is highly possible that a mobility manager does not have up-to-date information about the role-figures, role-figures and mobility managers alike may be unaware of the mobility procedure current status, and requests aimed to mobile role-figures be sent to the wrong location.

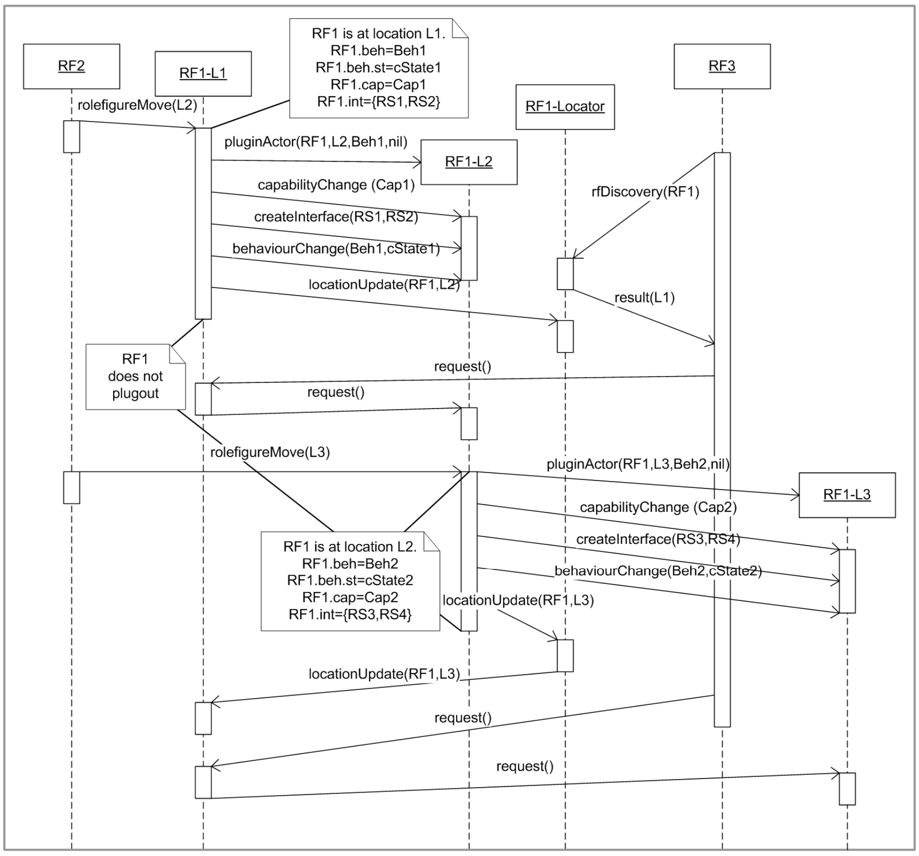


Figure 3-27 RMM4 using role-figure locator

Disadvantages of RMM4 This mechanism scales down to the movement of the individual role-figures, as we don't plug out old instances of a moving role-figure. Certain situations can decrease the efficiency of this mechanism. For instance, when role-figures have very frequent movements. Besides, as it is using *locator* objects, the limitations of RMM3 affect it as well.

3.4.4 Role-figure mobility management design rules

In this subsection we give design rules to show the overall behaviour of a moving role-figure and *MobilityManager* with regard to role-figure mobility. These design rules, *RD1-RD9*, complement the requirement rules for role-figure mobility, presented in the terminology framework. These rules will be divided into two groups: general design rules (*RD1-RD8*) and design rules for the mobility strategy (*RD9* (a) - *RD9* (*r*)).

The rules *RD1*, *RD2*, *RD3*, *RD4*, and *RD5* give options on the supervision of role-figures by mobility managers. *RD6*, *RD7*, and *RD8* show options for a moving role-figure between different domains that are managed by different mobility managers.

RD1. If a single mobility manager exists in a domain or sub-domain, then it *MAY* be required that a role-figure is supervised by the domain or the sub-domain mobility manager.
RD2. If several mobility managers exist in a domain or sub-domain, then it *MAY* be required that a role-figure is supervised by a specific mobility manager.
RD3. A role-figure *MAY* be supervised by more than one mobility managers at the same time.
RD4. Mobility manager *MAY* supervise all role-figures created in its domain or sub-domain.
RD5. Mobility manager *MAY* supervise only those role-figures that it was instructed to supervise (these instructions may be issued by the supervisory object of the domain that controls the plug in phase).
RD6. In the case of mobility between two domains or sub-domains with different mobility strategies, the relationship between these strategies *MAY* be provided.
RD7. After a movement a moving role-figure *MAY* register itself at another mobility manager.
RD8. After a movement a mobility manager where a mobile role-figure has moved to *MAY* register it.

With regard to *RD7* and *RD8*, these rules are options of the requirement rule *RR5*, which stated that a moving role-figure must be supervised after it moves to another domain or sub-domain. All the design rules presented here handle the role of the mobility manager. Therefore we only regard them to RMM1 that deploys this mobility manager. In RMM1 we implemented *RD1*, *RD4*, and *RD7*.

Regarding the content of the mobility strategy we give the rules *RD9* (*a*) – (*r*) as options to be included. (*a*), (*b*), and (*c*) give options regarding the queue handling. (*d*) and (*e*) handle the multiple copies of a moving role-figure. (*f*) shows the plug out of a moving role-figure, while (*g*) sets an option for priorities among mobility strategy rules. (*h*), (*i*), (*j*), and (*k*) give options for the handling of role-sessions. (*l*), (*m*), and (*n*) give options for handling the capabilities. (*o*) and (*p*) handle the behaviour, while (*q*) and (*r*) handle the executing methods of a moving role-figure.

(***a***) In mobility strategy rules, mobility manager *MAY* discard the queue contents of a mobile role-figure.
(***b***) In mobility strategy rules, mobility manager *MAY* forward the queue contents of a mobile role-figure to its new location without confirmation of the arrival (confirmation of the arrival or confirmation of the order of arrival).
(***c***) In mobility strategy rules, mobility manager *MAY* forward the queue contents of a mobile role-figure to its new location with confirmation of arrival (confirmation of the arrival or confirmation of the order of arrival).
(***d***) In mobility strategy rules, mobility manager *MAY* allow two copies of a mobile role-figure to exist simultaneously at two different locations.
(***e***) In mobility strategy rules, mobility manager *MAY* disallow two copies of a mobile role-figure to exist simultaneously at two different locations.
(***f***) In mobility strategy rules, mobility manager *MAY* plug out a mobile role-figure once it has been instantiated at another location.
(***g***) In mobility strategy rules, priorities *MAY* be set among mobility strategy rules to resolve conflicts.
(***h***) In mobility strategy rules, mobility manager *MAY* discard re-instantiation of all role-sessions.
(***i***) In mobility strategy rules, mobility manager *MAY* require the re-instantiation of all role-sessions.
(***j***) In mobility strategy rules, mobility manager *MAY* discard the recreation of role-sessions to special-purpose role-figures only.
(***k***) Mobility strategy *MAY* include a list of special-purpose role-figures and their functionality (this list may be used to locate appropriate special-purpose role-figures for moving role-figures).
(***l***) In mobility strategy rules, mobility manager *MAY* discard the reclamation of all capabilities.
(***m***) In mobility strategy rules, mobility manager *MAY* require the reclamation of all capabilities.
(***n***) In mobility strategy rules, mobility manager *MAY* require the reclamation of certain types of capabilities.
(***o***) In mobility strategy rules, mobility manager *MAY* require the recovery of the behaviour specification.
(***p***) In mobility strategy rules, mobility manager *MAY* require the recovery of the state-information.
(***q***) In mobility strategy rules, mobility management procedure *MAY* be

preemptive (to stop the role-figure active tasks and discard the executing methods).
(r) In mobility strategy rules, mobility manager *MAY* be *non-preemptive* (to allow the role-figure active tasks to execute).

As these design rules handle the role of the mobility manager, we will only regard them to RMM1. In RMM1 we implemented (*q*), (*d*), (*f*), (*i*), (*m*), (*o*), (*p*), and (*q*). RMM2, RMM3, and RMM4 have been implemented using similar design rules but they have been incorporated in the role of role-figure proxy and locator.

3.5 Terminal mobility

Terminal mobility as defined in the terminology framework is the movement of terminals while maintaining access to services and applications. This implies the change of the terminals location as well.

The terminal mobility will be discussed in three parts. Sec. 3.5.1 illustrates a terminal characterization to distinguish between nodes and terminals. Sec. 3.5.2 presents the mobility management procedures for terminal mobility. Sec. 3.5.3 gives a set of design rules.

3.5.1 Concept

A terminal has been defined as a node that is associated with human users. In the personal mobility management procedures, we have encountered that terminals and nodes both can execute services and instantiate role-figures. Terminals have typically limited set of capabilities (regarding resources, computational and processing power) when compared to nodes. Therefore role-figure functionality will execute and perform differently in node and terminal execution environments. In Figure 3-28 we illustrate a terminal characterization that will be used as a basis for the terminal mobility management procedures.

A terminal in our mobility management architecture is characterised by the following entities: User Interface (UI), Terminal Interface (TI), User Agent (UA), Mobility Agent (MA), and constrained role-figures (^{c}RF). ^{c}RF is a role-figure with constraints in terms of its capabilities, child-session, mobility, etc. These constraints are determined by the terminal's capabilities.

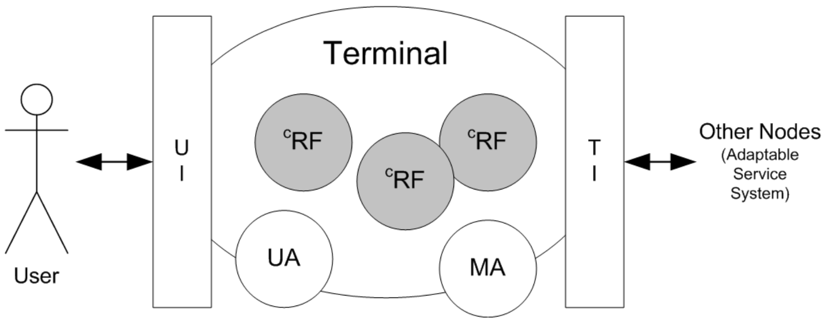


Figure 3-28 An illustration of terminal characterization

3.5.2 The mobility management procedures

In the terminology framework we discussed that terminal(s) execute *MobilityAgent*(s) responsible for tracking their location, while *MobilityManager* is responsible for updating these locations. Based on the topology of the network, domains and sub-domains may exist, and therefore several mobility managers may be required. To illustrate terminal mobility we assume a simple case of a single mobility manager per domain.

There are two main terminal mobility management procedures: *LocationUpdate* and *TerminalDiscovery*. A *MobilityAgent* initiates a *LocationUpdate* procedure when the terminal changes its location. Terminals and nodes initiate a *TerminalDiscovery* procedure when a communication with a terminal is demanded. *TerminalDiscovery* may be realized in two different ways:

- Nodes and terminals must initiate this procedure before every communication with any other terminal
- Nodes and terminals may maintain a cache of addresses that they use to store recently accessed terminals, and only initiate this procedure when communicating with an unknown terminal

Figure 3-29 illustrates a general case of terminal mobility. This figure includes three main parts to show in logical order a terminal mobility scenario:

– Part (1): access phase (*Terminal* at *domain1*);
– Part (2): moving phase (moving *Terminal* from *domain1* to *domain2*), *MobilityAgent* ensures that *MobilitManager* is updated on this movement. Meanwhile, when the terminal reaches the so-called *out-of-coverage* area, it is inaccessible;
– Part (3): access phase (*Terminal* at *domain2*).

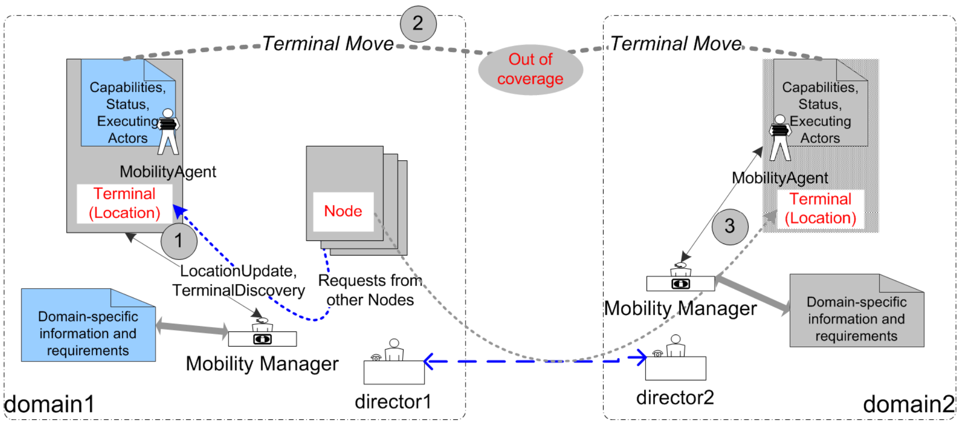

2
Terminal Move
Out of coverage
Terminal Move
Capabilities, Status, Executing Actors
MobilityAgent
Terminal (Location)
Node
1
LocationUpdate, TerminalDiscovery
Requests from other Nodes
Domain-specific information and requirements
Mobility Manager
director1
domain1
Capabilities, Status, Executing Actors
MobilityAgent
Terminal (Location)
3
Mobility Manager
Domain-specific information and requirements
director2
domain2

Figure 3-29 Illustration of the terminal mobility

In parts (1) and (3), requests from other nodes addressed to the terminal should use a *TerminalDiscovery* procedure, which is executed through the corresponding *MobilityManager* in the domain. Domain specific set of information and requirements might be used to control the privileges, access rights, roaming, etc. of users and terminals. For seek of generality, we don't consider these requirements in the terminal mobility procedures.

Figure 3-30 illustrates a sequence diagram for the terminal mobility.

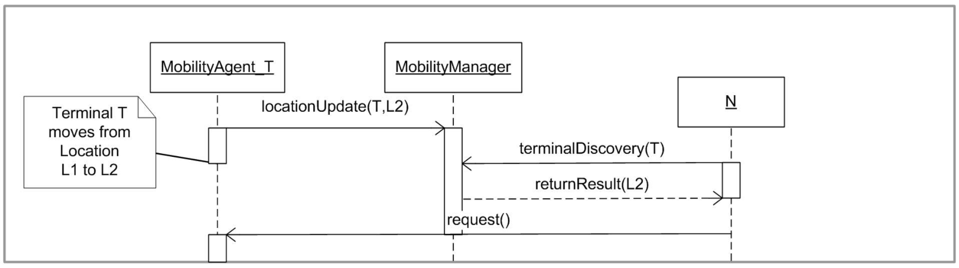


Figure 3-30 Sequence diagram for a terminal move

A node (*N*), a terminal (*T* represented by its mobility agent *MobilityAgent_T*), and locations (*L1* and *L2*) in this sequence diagram are shown as an illustration of a general terminal move within one domain. The node is used to communicate with the moving terminal. *MobilityAgent_T* updates the mobility manager about the terminal movement to the new location *L2* (using the invocation request *locationUpdate* (terminal, location)). *N* gets the location from the mobility manager by requesting a terminal discovery procedure (using the invocation request *terminalDiscovery* (terminal)).

3.5.3 Terminal mobility management design rules

Terminal mobility has been dealt with to a great extent by many mobile systems, e.g. mobile telephony systems. Issues such as mobile unit identification, user subscription, network handover and roaming are few examples of the issues that were deeply covered. We only focus on the issues of terminal mobility that implicate and affect *personal* and *role-figure* mobility types.

In this subsection we give design rules to show the overall behaviour of *MobilityManager* and *MobilityAgent* with regard to terminal mobility. These design rules, *TD1-TD7*, complement the requirement rules for terminal mobility, presented in the terminology framework. We only implemented few of these rules that were adequate to experiment with the functionality of terminal mobility. The other design rules have been encountered during the implementation as alternatives to our implementation.

For the terminal mobility management we limit our discussion to the following set of design rules. Options for terminal location information handling in the mobility agent are given in *TD1* and *TD2* (which are options of the requirement rule *TR5*), and options for terminal location information handling in the mobility manager are given in *TD3* and *TD4* (which are options of the requirement rule *TR6*). *TD5* and *TD6* specify options regarding the update of terminal locations. *TD7* give an option for handling user terminals in visitor domains.

TD1. The mobility agent *MAY* send the information location updates at regular intervals.
TD2. The mobility agent *MAY* send the information location updates whenever the location changes.
TD3. The mobility manager *MAY* maintain the information on terminals identifications and their locations locally in the node where it executes.
TD4. The mobility manager *MAY* maintain the information on terminals identifications and their locations locally in a central database.
TD5. A node or a terminal *MAY* always ask for the location of a terminal by sending a terminal discovery request to the mobility manager.
TD6. A node or a terminal *MAY* store locations of terminals and access these terminals directly, and only sends a terminal discovery request if no information on the location of a terminal is stored.
TD7. A terminal *MAY* be granted access to services when entering a visitor domain (if the visitor domain permits visitor terminals and if the

visitor domain authorizes this terminal to access services).

Regarding our proposed solution for terminal mobility management we have applied the following design rules: *TD2*, *TD3*, and *TD5*.

3.6 Mobility management architecture implementation

The major concern of the implementation of the mobility management architecture was the implementation of the role-figure. The issue at this point is the mapping of the role-figure parts into the real programming environment. In other words describing in detail how it is possible to: accomplish the inter-object communication, represent the registry elements, handle the queuing of messages, etc.

The implementation of the mobility management architecture has been based on the TAPAS core platform. It has a support functionality based on Java RMI, socket interface, and web services as means for service discovery and service execution. Figure 3-31 presents an illustration of the implementation of the mobility management architecture.

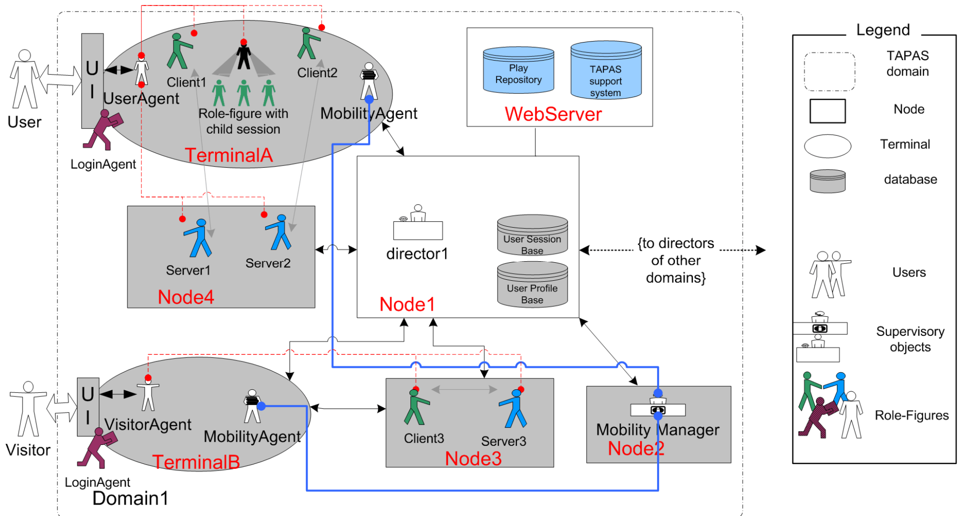


Figure 3-31 An illustration of the mobility management architecture implementation

Figure 3-31 shows a TAPAS domain, with *nodes*, *terminals*, *databases*, *users*, *supervisory objects*, and *role-figures*. The *director* controls the *user session base* and the *user profile base*. The *director* can have connections to other directors in other domains. The *play repository* and the *TAPAS support system* are available for downloading from a web server.

To experiment with the personal mobility, a platform for *personal mobility* was developed. In the implementation of the management functionality for p*ersonal mobility* we implemented the following: *LoginAgent*, *UserAgent*, and *VisitorAgent*. Also we have implemented the databases *UserSessionBase* and *UserProfileBase*. This management functionality implemented the mobility management procedures for the user mobility using the following design rules from the user mobility management: *UD3*, *UD4*, *UD5*, and *UD6*. Also it implemented the mobility management procedures for the user session mobility using the following design rules from the user session mobility management: *SD2*, *SD4*, *SD5*, and *SD6*. The main characteristics of this platform have been presented in [SL02]. The implementation details of this platform have been worked out by [Lil03]. In this platform experiments with moving *users* and *user sessions* have been conducted in different terminals and domains.

Another platform for *role-figure* and *terminal mobility* has been developed, which was based on the mobility management procedures and mechanisms presented in this chapter. Regarding *role-figure mobility*, we implemented moving role-figures, *MobilityManager*, and mobility strategies. *Terminal mobility* has been handled by *MobilityManager* and *MobilityAgent* role-figures. This implementation used the following design rules for role-figure mobility: *RD1*, *RD4*, and *RD7*, in addition the options (*q*), (*d*), (*f*), (*i*), (*m*), (*o*), (*p*), and (*q*) from the design rule *RD9* have also been implemented. Regarding terminal mobility management we have applied the following design rules: *TD2*, *TD3*, and *TD5*. This platform has been implemented and demonstrated in both fixed and wireless environments with a number of mobile applications. These applications have been specified in plays each with a set of role-figure specifications. Several test cases have been demonstrated. This platform is subject to continuous improvements, and more applications are being developed based on its functionality. The main characteristics of this platform have been presented in [Shi03]. Several implementation versions of this platform have been developed and experimented, see [Luh03], [Hen04] and [Smi04] for the implementation details.

CHAPTER **4**

The Role-figure Model

4.1 Introduction

IN THIS CHAPTER, an abstract model for the implemented role-figure functionality is presented. This model is called the *role-figure model*. This model will be used to reason about the structure and the behaviour of role-figures with regard to the role-figure mobility. In Sec. 4.2 five modelling techniques relevant to our modelling effort are presented. These are general techniques that have been used in the description and specification of service systems and communicating systems.

The role-figure model is discussed in four parts. In Sec. 4.3 we give an operational model that shows how role-figures can be implemented. In the operational model we informally discuss the structure and the implementation of role-figures. In Sec. 4.4 we present the role-figure model semantics. These semantics give structural and behavioural semi-formal semantic rules used to outline the structure and the behaviour of role-figures. Based on these semantics we further elaborate on the role-figure model. In Sec. 4.5 we handle the dynamics of role-figures, and in Sec. 4.6 we discuss certain properties of role-figures. These four levels will be called the *operational model*, the *semantics*, the *dynamics* and the *properties* of the role-figure model. We conclude this chapter by giving an example of our semantics in Sec 4.7.

4.1.1 Motivation

The mobility management architecture, have been specified using various UML diagrams that have served as the basis for the specification of the prototype implementation (some of these diagrams have been presented in chapter 3). The implementation and demonstration of the architecture helped gaining knowledge of the applicability and validity of the proposed solutions for mobility management. This is a typical research process that involves certain extent of prototype implementation. On the other hand, a model as a basis for formal understanding and reasoning was also needed. There was both a need for a non-ambiguous compact specification of the generic parts of the computing architecture, as well as a model that could be used as a basis for reasoning about various issues related to mobility. Formal approaches can add value to the limited analytic power of architectural notations based on non-formal notations such as UML.

Regarding the construction of the role-figure model, two main challenges have been encountered:

The first challenge The role-figure model needs to be programming language and implementation platform independent. At the same time it should be abstract enough, service oriented, and capable of capturing the features and properties of the role-figure mobility.

The second challenge The formalism that we base our role-figure model on needs to be simple to understand, provide various levels of abstraction of the system specifications, has been used in similar activities, and has a good language and tool support.

We handle these challenges in this chapter. We construct a role-figure model that is platform independent, abstract, and service oriented. We use a simple and compact formalism that is capable of handling role-figure mobility at different levels of detail. The role-figure model also will be the basis for the formal specification, analysis, and validation of role-figures, which will be presented in Chapter 5.

4.1.2 Modelling aspects

Thus far in the thesis, a role-figure has been defined as an actor with a specific behaviour that realizes the service-components. As such it is the behavioural element of the service system. In the terminology framework we sketched a general structure of the role-figure and only outlined its parts. Also, the role-figure mobility has been handled based on simplifications regarding the role-figure's behaviour and role-session definitions. Both the general structure and the simplified definitions were necessary to avoid handling details, and therefore generating exhaustive diagrams, e.g. message sequence diagrams.

In this chapter we present a concrete description of the role-figure structure and behaviour. In the role-figure model we will consider the following aspects of role-figures:

(i) *Role-figures* are realized by *actors*.
(ii) *Role-figures* can be dynamically created.
(iii) *Role-figures* instantiation and execution are dependent on the availability of the system *capabilities*.
(iv) *Role-figures* comprise *behaviour*, which is an extended finite state machine.
(v) *Role-figures* interact with each other via *role-sessions*, and *role-figures* are connected to each other via *interfaces*.
(vi) *Role-figures* interact with each other via asynchronous messages. These messages are the only means to interact with a *role-figure* from the outside.
(vii) *Role-figures* comprise *methods* (well-defined sets of tasks that correspond to the actor basic support procedures), which are used for managing and controlling the actor object of the role-figure.
(viii) 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	changes role-figure locations

4.2 Related work

In this section five different modelling approaches will be discussed. These approaches reflect four totally different viewpoints. Each one has different concepts and goals. These choices are: the *Java class semantics*, the *SDL agent semantics*, the *ODP semantics*, the *Actor language model*, and the *Rewriting Logic*. These choices give a broad picture of modelling in terms of design, specification, implementation and realization. We conclude this section with a discussion on our choice for the modelling approach of the role-figure.

4.2.1 The Java class semantics

Role-figures are mapped to software components as means for implementing the services. The Java programming language has been used in the implementation for role-figures. Figure 4-1 shows a class diagram for the java implementation of a role-figure, which is part of the prototype implementation of the TAPAS Computing Architecture [AHJ01]. It includes the classes *Actor* (with two interfaces: *ActorInterface* and *ControlInterface*), *CapabilitySet*, and *ContextInformation*. A role-figure is implemented as a hierarchy of classes with methods, which can execute in a java platform to realize the functionality of a given service.

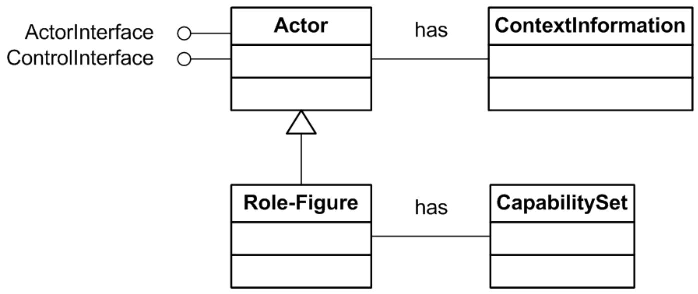


Figure 4-1 Class diagram of the role-figure implementation in Java

As a preliminary assumption, we considered building our role-figure model based on a framework of semantics for Java. Based on the study conducted by Wallace [Wal97], operational semantics of the Java class will be presented. Java is a class-based, object-oriented high-level programming language. In a java program, actions have the form of operations performed on objects, which are instances of user-defined classes with named state variables (fields) and procedures (methods) [Wal97]. The Java class semantics has a rather lengthy specification; we will focus on two main issues that are relevant to role-figures: the state specification and the method specification.

The semantics of a java class can be specified using the Abstract State Machine (ASM) [Gur95]. A state **S** of an ASM **M** with vocabulary γ consists of a nonempty set **X**, called the super-universe of **S**, and an interpretation of every function name in γ over **X**. These two parts define the state location (*l*): $l = (f, x)$, where *f* is an *r*-ary function name and *x* is an *r*-tuple of elements of **S**. An ASM changes states by a state update, which is a change in the interpretation of a single function name for a given tuple of arguments. The update itself is expressed within the definition of **S**. An update of **S** is a pair (l, y), where *l* is a location and *y* is an element of **S**. This update is equivalent to the pair (current state, next state). A transition according to this terminology is a set of terms that change variable values. A term can be a variable name, function name, etc. Each change in a state is an apparent change in a variable. An ASM performs controlled state updates through transition rules. For a given state **S**, a transition rule can be an update using assignment or conditional statements for instance.

The java method declaration is the other issue we focus on. A method declaration consists of a method header and a method body. In general, a method has an identifier, a list of parameters, a return type, a list of thrown exceptions, and may have certain attributes, e.g. access status and static attributes. A method is invoked by an invocation request that contains: the invoker, the invoked method identifier, and an argument list. When such an invocation is encountered, the execution of the invoker or the current method is suspended, while the execution of the invokee (invoked method) starts. Before this takes place, information about the invoker is recorded: the target object, argument values, and the point to which control returns after termination of the invokee. As methods may be invoked recursively, a stack of method invocations is maintained, whose top element corresponds to the current invocation.

Using the Java class semantics to model role-figure aspects may be advantageous as these semantics give a clear sighting of the role-figure java implementation, particularly the programs that will execute them. Also, java method semantics can be used to model methods in the role-figures. We can use the java method requests (with a list of parameters) and return type semantics to model the role-figure method request invocation and return. We can also apply the java semantics for the stack of invocation to model the recursive method invocation of role-figures.

However, the java class semantics may not be the most appropriate choice to model other aspects of the role-figure. The Java class semantics is too detailed to be used for this purpose. For instance the behaviour part of the role-figures, which is an extended finite state machine specification, may be modelled using other semantics. It is also emphasized that modelling a role-figure is not intended to be a programming language dependent. Our conclusion is that stand-alone java semantics does not seem to give a comprehensive solution for the semantics of our role-figure model.

4.2.2 The SDL agent semantics

Specification and Description Language (SDL) is a standard language for specifying and describing systems. It has been developed by ITU in several standards. Its major versions that are in use nowadays are SDL-92

[ITU93] and SDL-2000 [ITU99]. SDL was envisaged for use in all kinds of real-time systems, especially telecommunications. The most important property defined by an SDL specification is the behaviour of the system. It shows how a system reacts to events in the environment, and communicates with it via signals. Pederson stated in [Ped00] that SDL, as opposed to UML, has a concrete and complete semantics focusing on the object and state machine views of the system. It is this particular view that we will focus on. An SDL system consists of agents that are connected by channels and gates. Agents can communicate via signals, and request other agents to perform procedures. An agent may have both a state machine and composite agents. An agent, accordingly, may be a process or a block. In [Ped00] a UML conceptual model of the SDL was presented (see Figure 4-2.)

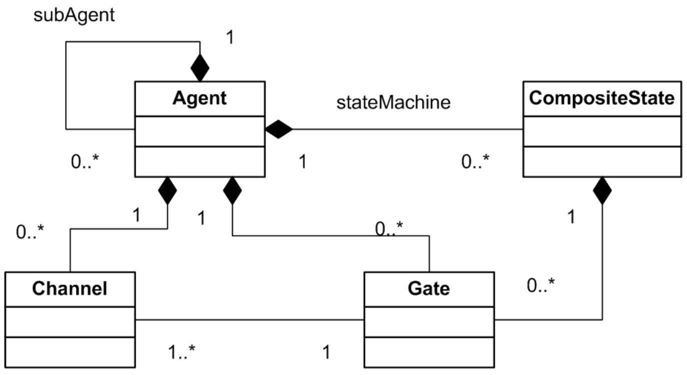


Figure 4-2 Class diagram of the SDL concepts [Ped00]

As SDL is an object-oriented language, an agent type can be the specialization of another agent type. All sorts of virtuality constraints, redefinition, finalization, and property adding may be applied in SDL to provide a powerful object orientation methodology. The state machine of an agent, as in Figure 4-2, is a composite state. This means that it has states and transitions between these states. The states themselves may be composite states as well. State machines in SDL have first-in-first-out (FIFO) buffers called the input queue. Signals are stored in these buffers and consumed by the state machines in their order of arrival. These signals may trigger state transitions. In a transition various actions may happen: variables are assigned values, agents are created, signals are sent, procedures are called, etc. A state machine is either waiting for a stimulus event (input, signal, exception handling, spontaneous event, etc.), or is in a transition at any given moment.

Procedures in SDL are meant as patterns of behaviour specification that can be used by agents as part of their transitions. Procedures have signatures defining their calling mechanisms.

In SDL, agents may have a number of required and implemented interfaces. Interfaces define signals (the signals and data an agent may send), variables, and remote procedures (procedure signatures an agent may use to request remote procedures from other agents), as in Figure 4-3.

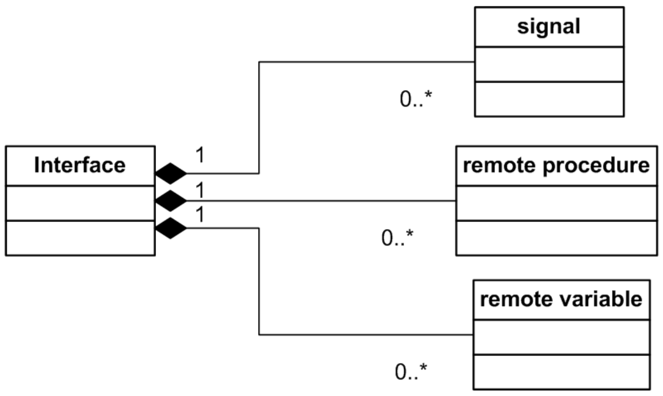


Figure 4-3 The concept of Interface in SDL [Ped00]

Bræk in [Bræ00] discussed the methodology of using the SDL and UML languages in the development process. In [San00] a wide range of SDL implementation issues were studied. Among these issues is the ideal perception of the SDL of the world surrounding it. Examples of such idealism are that state transitions take insignificant time, input queues are infinite, and that each state machine is executing in an independent processor. Proposals to implement SDL state machines as software processes on CPUs was also given.

It has been very clear at the beginning of the thesis work that SDL-2000 is a major candidate for modelling role-figures. In particular, the behaviour specification and the role-sessions of role-figures may be modelled using the SDL-2000 terminology. The extensions to SDL-2000 in Floch's thesis [Flo03] may be of a very particular usage in our modelling efforts also. Actually, as we will see in the role-figure semantics, the behaviour specification of the individual role-figures will rely on an extended finite state machine specification, which can adopt the SDL agent semantics. However, by the time when the modelling and formalization work of this thesis started a tool support for SDL-2000 lacked. Decision needed to be taken on either using a tool that only supports SDL-96, or search for a formal tool based on other formalism. The latter was the choice we took.

4.2.3 The ODP semantics

The formal semantics of the computational model of ODP was worked out by Najm and Stefani, and presented in [NS95]. Further elaborations on this model have been studied by Dutszadeh and Najm in [DN96] and [DN97]. Objects in the ODP computational model 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. Objects can have multiple interfaces, so that it is possible to have separate views of the object by its cooperating objects. Access to the object is only possible through invocations of its advertised operations on a designated interface. Interfaces in ODP are strongly typed, which yield interface types and subtypes. Essentially, an interface type is the specification of the operation signatures available on that interface. These signatures define the name of the operations, the number and type of their argument parameters, and a set of termination signatures to specify the outcome of the operations. Operations are called and executed asynchronously.

A formal interpretation of the ODP computational interface, according to [NS95] is shown in **Table 4-1**.

Table 4-1 Syntax of ODP interface definition

$$\begin{array}{lll}
\alpha & ::= & \langle m_1 : Opsig, \cdots m_n : Opsig \rangle \\
Opsig & ::= & Int \mid Ann \\
Int & ::= & Nil \to Term \mid Arg \to Term \\
Ann & ::= & Nil \to Nil \mid Arg \to Nil \\
Term & ::= & [\overline{m_1} : Arg, \ldots, \overline{m_q} : Arg] \\
Arg & ::= & t_1 \times \ldots \times t_p
\end{array}$$

Where m_1, .., m_n denote operation names with operation signatures, *Opsig*. Operations in ODP as thus are units of interactions between objects. There can be two kinds of operations: Interrogation or *Int* (a two-way interaction between two objects, invoked by a call request and returns a result in the form of termination, or *Term*), and Announcement or *Ann* (a one-way interaction, which does not return any result to the caller). According to this kind of interface type definition, an interface is a record of operations that can be invoked on instances of that type. The type associated with an operation name is a function type whose domain is the Cartesian product of argument types, and its range is either a *Nil* for announcement or a union of all possible terminations for an interrogation operations. To better understand the interface type definition we highlight the following example found in [NS95]:

$$\alpha =_{def} \langle op : t \rightarrow [ok : Nil, nok : Nil], factory : Nil \rightarrow [ok : f] \rangle$$

This interface type defines two operations *op* and *factory*. The operation *op* takes an argument that is a reference to an interface of type *t*, and returns either *ok* or *nok*. The operation *factory* takes no argument and returns a reference to an interface of type *f*.

The formal operational semantics of the ODP were built based on the conditional rewriting logic [Mes92]. Several customization efforts have been applied to fit the needs of the ODP terminology, e.g. the interface type theory mentioned above.

Generally speaking, the ODP computational model has several features that can be considered when modelling role-figures. Among these features are the dynamic creation of object interfaces, the definition of interfaces as procedural abstractions, and the distributed operation invocation. Also, we can apply the definition for the ODP computational terms in our role-figure model. The considered computational terms in ODP are either distributed computations or computational elements. These computational elements are ODP computational objects, invocation requests, or responses. These features of the ODP computational model are similar to the main aspects of the role-figure, in Sec. 4.1.2. Role-figures can be modelled as ODP computational objects and their interactions can be modelled as requests and responses. These similarities have convinced us to use these ODP semantics as the basis for our model.

4.2.4 The Actor language model

Another approach that was studied was the actor language model, by Agha, Hewitt and Talcott in [AH88], [Agh90] and [Tal96]. The name actor here is coming from the nature of the object as an active entity, which takes actions[1]. Actors have intensions, resources, messages and a scheduler. Actors are considered to be independent computational agents that interact solely via message passing. An actor can create other actors, send messages, and modify its local state. An actor can only affect the

[1] This actor term should not be confused with the concept of *actor* in TAPAS.

local state of other actors by sending messages, and it can only send messages to its acquaintances – addresses of actors it was given upon creation, received in a message, or actors it created itself. The semantic framework for the actor language model was defined in an actor rewriting theory, Rt_A, [Tal96]. The rewriting logic was chosen to give the behaviour of the actors in the form of rewriting rules. The paper [Tal96] formalized the actor language model using rewriting logic, while [DG94] formalized the actor language model using linear logic. The equations abstracting the states of the actors gave also computational paths, and allowed the treatment of the actor computations at many levels of detail.

An actor rewriting theory is basically a rewriting theory, $Rt = (Eth, Rules)$, that consists of an equational theory, $Eth = (Signature, Equations)$ over a given set of variables, *Var*, together with a labelled set of rewriting rules, *Rules*. An actor system is a collection of actors interacting with each other and with their environment via asynchronous messages. The behaviour of the individual actors is given by the following Abstract Actor Structure (AAS)[2]:

Sorts: **A**, **V**, **S**, and **oF**, with $\mathbf{A} \subseteq \mathbf{V}$.	**sort definition**
Relations: $En_d \subseteq \mathbf{A} \times \mathbf{S} \times \mathbf{V}$, $En_{ex} \subseteq \mathbf{A} \times \mathbf{S}$	**predicates**
Operations: (some of the operations)	**operations**
$\_ \triangleleft \_ : \mathbf{A} \times \mathbf{V} \rightarrow \mathbf{oF}$	**message**
$Deliv : En_d \rightarrow \mathbf{oF}$	**message delivery**
$Ex : En_{ex} \rightarrow \mathbf{A} \rightarrow \mathbf{oF}$	**single execution**
$\#new : En_{ex} \rightarrow \mathbf{Nat}$	**created actors**

A is a countable set of actor addresses, **V** is a set of values that can be communicated between actors, and **S** is a set of actor states. **oF** is the multisets of actors and messages, in which no two actors have the same address. Let *a* range over **A**, *v* range over **V**, and *s* range over **S.** $En_d\ (a, s, v)$ is a predicate that holds if actor *a* at state *s* (noted $(s)_a$) is enabled for delivery of message *v* (delivery here stands for the consumption of the message). $En_{ex}\ (a, s)$ is a predicate that holds if actor *a* at state *s* (noted

[2] This is a shortened specification of the Abstract Actor Structure that is relevant to the role-figure model. The detailed specification in [Tal96] handles an actor system component to be an encapsulation of actors and messages with only certain actors (the receptionists of the component) to receive messages from outside the component.

$(s)_a$) is enabled for execution (execution stands for the evolution from a given state to another state). $a \triangleleft v$ is a message with target a and content v. If En_d (a, s, v) then $Deliv$ (a, s, v) gives an actor with address a in a state resulting from delivering the message. Ex $((s)_a)$ specifies the result of a single execution step of actor $(s)_a$. $\#new$ (s) is the number of new actors that will be created by actor $(s)_a$.

A simple example of an actor system, an actor always at state idle enabled for execution but not delivery:

$$\#new(\text{idle}) = 0 \text{ and } Ex((\text{idle})_a)[] = (\text{idle})_a$$

The actor rewriting theory extends the AAS and defines two layers of computations: internal computations, and interaction of an actor system with its environment.

This actor language model has been intensively studied and used by many researchers as a framework to model concurrent objects as communication interactions between messages and objects, using mainly mathematical structures and focusing on the computational semantics.

We have closely studied this model, together with its formalization using the rewriting logic theory. We have also looked at examples of actor system applications and experiences, [AH88], [Agh90], [DG94] and [Tal96]. This kind of formalism has shown us the applicability and advantages of formal methods in actor-based systems. The rewriting rules applied in this model have inspired us to use a similar approach in the role-figure model. The principles of the actor language model are similar to the role-figure concept. However, handling adaptability aspects, such as role-figure mobility, using this model does not seem a trivial task. To handle these aspects we need to apply certain modifications and extensions to this model.

4.2.5 Rewriting Logic

The ODP semantics and the actor language theories used rewriting logic as a formalization framework for their concepts. In this subsection, we briefly present the main elements of the rewriting logic theory, which was first introduced by Meseguer in [Mes92]. A rewriting theory $\mathcal{R}$, see Appendix II, as was introduced and elaborated in [MM93], is a pair $\mathcal{R}$ $=((\Omega, \Gamma), R)$ where (Ω, Γ) is an equational specification with signature of operations Ω and set of equational axioms Γ, and *R* is a set of labelled *rewriting rules*. The equational specification describes the static structure of the system, while the dynamic parts of the system are described by rules in *R*.

According to [MM93], an object, in our case a role-figure, in a given state is represented as a term $<O : C \mid a_1 : v_1 , a_2 : v_2 , \ldots a_n : v_n >$, where *O* is the object's name, *C* is its class, the a_i's are the object's attribute identifiers, and the v_i's are the corresponding values. Each *rewriting rule* will have the form:

$$\begin{aligned} r(\tilde{x}): \quad & M_1 \cdots M_n \left\langle O_1 : F_1 \middle| atts_1 \right\rangle \ldots \left\langle O_m : F_m \middle| atts_m \right\rangle \\ & \rightarrow \left\langle O_{i_i} : F'_{i_1} \middle| atts'_{i_1} \right\rangle \ldots \left\langle O_{i_k} : F'_{i_k} \middle| atts'_{i_k} \right\rangle \\ & \left\langle O_1 : D_1 \middle| atts''_1 \right\rangle \ldots \left\langle O_p : D_p \middle| atts''_p \right\rangle \\ & M'_1 \ldots M'_q \\ & if \quad C \end{aligned}$$

where *r* is the rule's label, $\tilde{x}$ is a list of variables occurring in the rule, the *M*'s are message expressions, $i_1,\ldots, i_k$ are different numbers and between $1,\ldots, m$, and *C* is the rule's condition. This form suggests that a number of objects and messages can participate in a transition in which certain objects can change state (those indexed by $i_1, \ldots, i_k$) new objects may be created, those indexed by ($1, \ldots, p$), and some new messages may be created (the M's on the right hand side of the rule). Some objects may also be removed (those indexed by $1,\ldots, m$ and do not appear in $i_1, \ldots, i_k$). If exactly one object and at most one message appear on its left hand side then these rules are considered as asynchronous, otherwise they are synchronous.

Rewriting logic is a logic of concurrent change that can naturally deal with state and highly nondeterministic computations. It supports very well concurrent object oriented computations. Rewriting logic has fundamental properties as a logical and semantic framework to specify systems. It does so by providing a great deal of expressiveness while maintaining and securing a fully abstract and generic description of concurrent systems. In this context, the so-called concurrent rewriting logic provides a general model of concurrency from which many other models can be obtained by specialization. Another advantage of this kind of formalism is that it is reflective, i.e. system verification and formal analysis may be expressed using the same formal language. This is achieved by regarding the system specification as a first-class entity to be used by the verification process at meta-level. The rewriting logic research program has shown a good deal of progress and solid results [Mau04]. Using rewriting logic many systems were specified and verified formally. In this direction we may list the following formalizations efforts: actor languages [Tal96], active networks [DMT00], the PLAN algorithms [WMG00] and meta-objects and composable distributed services [DMT04].

In the rewriting logic, specifying the behaviour of the individual role-figures as well as the configurations of role-figures can be described using rewriting rules. A role-figure in the rewriting logic is considered as a first-class entity. This level of abstractness and detail is just adequate for us.

At the same time equations, in the rewriting logic, will abstract the states of the role-figures and the states of the role-figure configurations. The abstraction of the computational actions, using equations also, will allow treating role-figure actions at many levels of detail depending on the particular need.

4.2.6 Others

Besides these five different modelling approaches we also looked at other formal methods as a choice for semantic framework. In the literature one can find many formal notations and specification languages of different orientation and background. There are also guidelines for selecting and comparing formal methods, e.g. [BH95-1] and [BH95-2]. We looked at one family of formal methods – process algebras. Process Algebras, such as CSP [Hoa85], CCS [Mil89], and π-calculus [MPW92], view any system as abstract communicating processes. A system specification that is based on process algebra is a representation of the communication primitives, the input/output ports, and the communicating pairs of processes. The main focus of these algebras is on the sequence of inputs and outputs. Process algebras are very powerful and have gained a lot of popularity. However, using them to formalize the computing architecture and the role-figure model was not our choice. The knowledge of many systems (e.g. systems discussed in subsections 4.2.3 and 4.2.4) that are similar to TAPAS and have been formalized using rewriting logic [Mes92] encouraged us to choose this kind of formalism over process algebras. We were also encouraged by the success and popularity of the rewriting logic in the recent years, as well as the several research efforts and dedicated conferences on its topics.

4.3 The Operational Model

Part of the modelling of role-figures is the modelling of their structural arrangement. This shows how role-figures can be logically mapped into the implementation domain. In this section we give an operational model that shows the structural arrangement of role-figures and their implementation. The operational model provides an informal discussion on the following: *behaviour and methods*, *role-sessions and interfaces*, and *implementation model for role-figures*.

4.3.1 Behaviour and methods

The behaviour of a role-figure is given by the specification of its role as part of an overall service functionality. Role-figures also rely on the functionality of the computing architecture. This functionality provides support procedures for communication, control, and management. In Sec. 1.4.2.2 we denoted these procedures as actor basic support procedures. The functionality of the computing architecture is implemented as methods in the actor objects that will execute the role-figures. These methods can be invoked and executed.

To give an operational model for role-figures we need to handle both the behaviour of role-figures and the methods of the computing architecture. The behaviour is considered the unit of *functionality*, so that role-figures can change their functionality by changing their behaviour. Methods are considered the unit of *operation*, so that the actor objects of the role-figures can execute operational and housekeeping tasks needed for the management of the service system.

The role-figure behaviour may be considered as a special method. It defines the functionality of the role-figure as a whole. It is active throughout the lifespan of the role-figure. It is an extended finite state machine based specification. The behaviour specification includes signature and state transition rules. The signature of the behaviour specification contains the type definitions, the axioms on these types, and the variable definitions. State transitions rules define how the behaviour of the role-figure will evolve.

Methods can handle a variety of tasks. A typical method can return the current state of the role-figure. Other methods can achieve initialization, synchronization, resetting, connecting, suspending, and destroying role-figures. For these example methods the following names are usually used: *init*(), *synchronize*(), *reset*(), *bind*(), *pause*(), and *halt*(), respectively. The parentheses show that these methods may have formal parameters, i.e. values to be passed upon invocation. The send and receive tasks can also be realized by methods, e.g. *send*() and *receive*(). Upon a method invocation, a method is activated and executed. Return result may be sent back to the invoker to indicate how the method execution terminates.

The execution of methods requires higher priority than the execution of the behaviour of a role-figure, in terms of queued requests and scheduled tasks. This priority however will not be discussed in this thesis.

4.3.2 Role-sessions and interfaces

Role-figures interact with each other via role-sessions, and are connected to each other via interfaces. Ideally the specification of a role-figure, and its current state, is the specification of all its role-sessions and interfaces. A role-figure based service system can be described at any time as a set of connected role-figures that interact with each other using messages. The system evolves when the role-figures consume these messages. In TAPAS the interface concept and the role-session concept are related and dependent. In Sec. 3.4.2.1 we discussed this dependency, and concluded that without appropriate concepts for role-sessions we can only handle the creation and recreation of interfaces. In this chapter we focus on the specification aspects of interfaces.

Regarding interfaces, we have seen the concept of ODP interface used in [NS95]. We adopt this concept in our role-figure model to constitute our interface-based semantics. Using this concept imply that the role-figures only access the correct methods on the given interfaces. For instance, interfaces connecting the director and other role-figures in a domain would include all the actor basic support procedures for control and management. On the other hand, an interface connecting two role-figures in two domains may prohibit the plug out or the behaviour change functionality between these two role-figures.

Another interesting interface concept was presented by Carrez, Fanatechi, and Najm in [CFN03], where they introduced an interface type theory. This interface type theory has been used as the basis for behavioural contracts of an assembly of components (configuration of components). The interface compatibility rules, defined as part of the contracts, guarantee certain safety and liveness properties of an assembly of components. Some of the concepts of this interface type theory may be used in our role-figure model semantics to guarantee the compatibility between interfaces, in particular the interfaces of a moving role-figure.

The incorporation of such interface type theory into our semantics will not be handled in this thesis.

4.3.3 Implementation model for role-figures

We need a general implementation model for role-figures to logically map them into the implementation domain. Implementation models can vary a lot based on the used computing architecture, communication infrastructure, etc. We focus on giving an implementation model that is general enough to be used as the basis for our role-figure semantics and dynamics. This implementation model should handle at least the following issues: the interpretation of the role-figure actions, the communication between role-figures, and the role-figure operations that involve the operating memory. The operating memory is used to instantiate role-figures, and to provide the data space for the role-figure state information. Figure 4-4 sketches an implementation model for role-figures. An interpreter process activates role-figures and passes on messages to them. It is also responsible for scheduling their actions. A role-figure executes based on its behaviour definition. The methods provide the means for control, management, and maintenance by the architecture. The data space is used to store all the data required in the operation of the role-figure, e.g. variables, capability information, and interface information. The role-figure behaviour, methods, and data space are considered to exist in the operating memory. A role-figure interacts with the rest of the architecture via asynchronous messages *received from* and *sent to* the message delivery system, which is an abstraction of the underlying communication system.

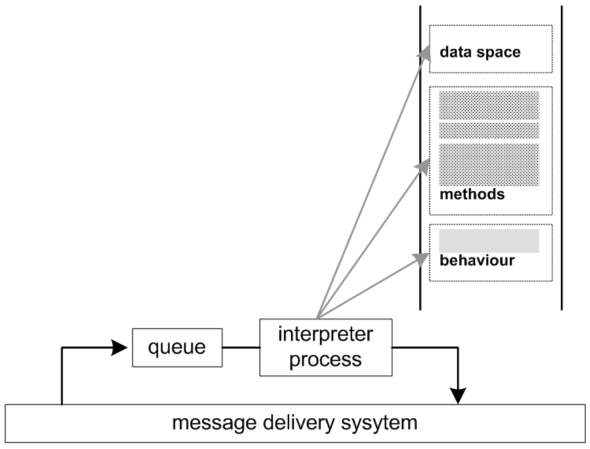


Figure 4-4 Implementation model for role-figures

To illustrate the operational principles of this implementation model we show how role-figures execute in a TAPAS domain. Figure 4-5 shows a system that executes two role-figures, *a* and *b*, in a domain of one node, *node1*. The system as a whole is administered by a director *dir*. There are also two software modules execute in the node. The Node Execution Support (NES) facilitates the communication between nodes, while the Actor Environment Execution Module (AEEM) can be regarded as the interpreter process of role-figures. An execution run of role-figure *a* is shown as a chain of actions running from top to bottom. These actions can be receiving a message, accessing the data space, invoking a method, behaviour interaction, or sending a message.

This role-figure gets messages from a queue, where incoming messages are queued. It sends messages to a message delivery system, where outgoing messages can be sent. The queue is associated with AEEM, while the message delivery system is associated with NES. In Figure 4-5 interfaces are shown as thick lines around role-figures, e.g. interface between role-figure *a* and *dir*. Interfaces connecting two role-figures are connected by a line.

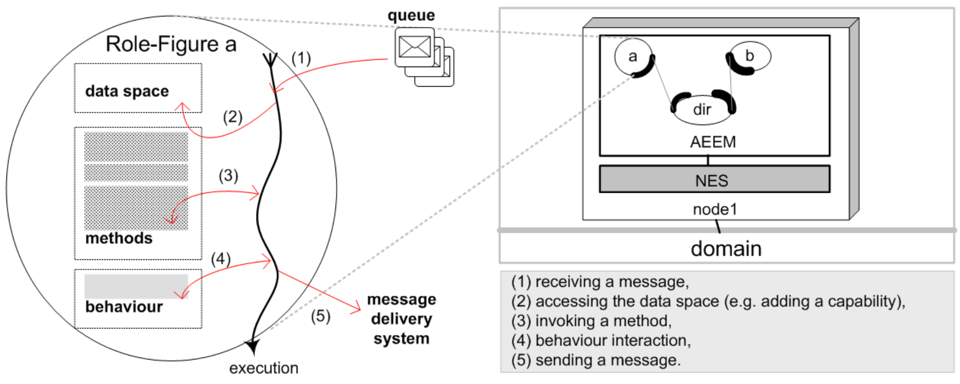


Figure 4-5 The implementation of a role-figure system

The operational model will provide the basis for the semantics and dynamics of the role-figure model.

4.4 The Role-figure Model Semantics

This section presents the semantics of the role-figure model, which are semi-formal semantic rules outlining the structure and the behaviour of role-figures. These semantic rules are inspired by the semantics of the ODP computational model presented in [NS95]. The semantic rules outlining the behaviour of role-figures are rewriting rules that are based on the rewriting logic theory presented in [Mes92] and [MM93].

The semantics will be divided into four subsections: *role-figure components*, *interface definition*, *behaviour definition*, and *behaviour evolution*. These subsections will use the following notations:

▪ ***a***, ***b***, ***f***, ***g***, ***h***	denote role-figure names;
▪ *RoleFigures*	denotes the set of all existing role-figures in a given system;
▪ $\mathcal{A}$,$\mathcal{A}'$,… $\mathcal{B}$,$\mathcal{B}'$,…	denote role-figures ***a*** and ***b*** as they evolve, respectively;
▪ *i*, *j*	denote interface names;
▪ α, α°	denote interface types;
▪ $\langle w_1=\nu_1,w_2=\nu_2\rangle$	denotes the record containing two fields named w_1 and w_2 and having the values ν_1 and ν_2, respectively;
▪ $r.w_1$	stands for the value of the w_1 field in record r;
▪ $=_{\mathrm{def}}$	stands for equality by definition;
▪ $\|\|$	denotes the asynchronous parallel operator;
▪ $\lhd$	denotes an infix insert operator;
▪ $\overset{n}{\underset{i=1}{\lhd}}$	denotes applying the operator $\lhd$ n times
▪ $\rhd$	denotes an infix remove operator.

The operators ||, $\lhd$ and $\rhd$ are commutative, associative, and have $\varnothing$ as a neutral element. The insert operator "a$\lhd$b" only executes if its left-hand side argument, a, is not in its right-hand side argument, b. Otherwise it does nothing. The remove operator "a$\rhd$b" only executes if its left-hand side argument, a, is in its right-hand side argument, b. Otherwise it does nothing.

4.4.1 Role-figure components

To develop the semantics for the role-figure model we need to define the syntax to denote the various elements of a set of cooperating role-figures, or a *Role-Figure Configuration (RFC).* Inspired by the semantics of the

ODP computational terms, we define an *RFC* as a set of asynchronously interacting role-figures. *RFC* is either an Empty computation ∅, *Role-Figure Configuration Element (RFCE),* or parallel *RFC-s*, as in Table 4-2.

Table 4-2 Main syntax components of the role-figure model.

RFC	::=	$\emptyset \quad \mid \quad RFCE \quad \mid \; RFC \parallel RFC$
$RFCE$	::=	$RF \mid MSG$
RF	::=	$\langle Int = \gamma, Beh = \beta, Cap = \pi, Que = \omega, Met = \mu \rangle$
MSG	::=	$Req \mid Sig \mid Ret$

Where:

▪ *RFC*	denotes a role-figure configuration that may be empty, *RFCE*, or parallel *RFC*-s.
▪ *RFCE*	denotes an RFC Element that may be a role-figure, *RF*, or a message, *MSG*.
▪ *RF*	denotes a role-figure, and is defined by: *Int* (set of interfaces), *Beh* (behaviour), *Cap* (set of capabilities), *Que* (queue of *messages*), and *Met* (set of executing methods).
▪ *MSG*	denotes a message that may exist in the configuration, which can be a method invocation request *Req*, communicating signal between role-figures *Sig*, or a method return result *Ret*.

A role-figure ***a*** is defined by the instantaneous state and structure of its parts, i.e. *Int*, *Beh*, *Cap*, *Que*, and *Met*. These parts may evolve and change as the role-figure consumes messages from the role-figure configuration. We use the following notation to represent the role-figure evolution of role-figure ***a***:

$$\mathcal{A} = < a : \text{RoleFigure} \mid Int = \gamma, Beh = \beta, Cap = \pi, Que = \omega, Met = \mu >$$

where $\mathcal{A}$.*Int*, $\mathcal{A}$.*Beh*, $\mathcal{A}$.*Cap*, $\mathcal{A}$.*Que*, and $\mathcal{A}$.*Met* are its interfaces, behaviour, capabilities, queue of messages, and executing methods, respectively. This notation is used by rewriting logic to represent objects. The class definition is given by (*a*:RoleFigure), while the attributes are defined by *Int*, *Beh*, *Cap*, *Que*, and *Met* (see Sec. 4.2.5). We use role-figure names to distinguish different role-figures, e.g. role-figure ***a*** evolves through $\mathcal{A}, \mathcal{A}', \ldots$ while role-figure ***b*** evolves through $\mathcal{B}, \mathcal{B}', \ldots$ As a simplification, we will omit the class definition in the rest of the chapter and assume $\mathcal{A},\mathcal{A}',\ldots$ always stand for role-figure ***a*** (similarly, $\mathcal{B},\mathcal{B}',\ldots$ always stand for role-figure ***b***, etc.).

Met in the role-figure definition denote the executing methods, which execute in the operating memory. In our semantics we only consider the executing methods that have invoked other methods and are waiting for return values from other role-figures. The methods that are included in *Met* constitute the active tasks of a moving role-figure that we discussed throughout chapter 3.

In Table 4-3 the definitions of the role-figure are given.

Table 4-3 The definitions of the role-figure.

Interface	γ	::=	$\varnothing \mid \gamma \lhd [j:\alpha] \mid \gamma \rhd [j:\alpha]$
Behaviour	β		(defined in Sec. 4.4.3)
Capabilities	π	::=	$\varnothing \mid \pi \lhd [c:cn] \mid \pi \rhd [c:cn]$
Queue	ω	::=	$\varnothing \mid \omega \lhd [q] \mid \omega \rhd [q]$
Method	μ	::=	$\varnothing \mid \mu \lhd [m:mn] \mid \mu \rhd [m:mn]$
Invocation request	Req	::=	$\left\langle \begin{array}{l} tar = j:\alpha, src = a, met = m:mn, \\ ref = n:mn, ret = r, arg = \tilde{p} \end{array} \right\rangle$
Signal	Sig	::=	$\langle tar = j:\alpha, src = a, name = sig, arg = \tilde{p} \rangle$
Return	Ret	::=	$\langle tar = j:\alpha, src = a, ref = n:mn, arg = \tilde{p} \rangle$
Argument list	$\tilde{p}$	::=	$(p_1 : t_1, \cdots, p_n : t_n)$

The intuitive interpretation of these constructs is as follows:

- γ denotes the interface part definition, which takes the form of a list of interface references tagged by their types (Sec. 4.4.2)

  An interface *j* of type α, denoted [*j*: α], may:

  – be added to this list by the "$\lhd$" operator: $\gamma \lhd [j:\alpha]$

  – be removed form this list by the "$\rhd$" operator: $\gamma \rhd [j:\alpha]$
- β denotes the behaviour part definition. This definition has a very distinctive structure, which will be handled in detail later.
- π denotes the capability part definition, which takes the form of a list of capability identifiers, denoted *c*, tagged by corresponding capability name, denoted *cn* (capability name

stands for the class or type of a capability, while the capability identifier is an instance or value of that class). Capabilities may be inserted to or removed from π using the insert operator "$\triangleleft$" and the remove operator "$\triangleright$", respectively.

- ω denotes the queue part definition, which takes the form of a list of messages. Messages can be request messages, signal messages, or return messages, and are denoted *q*. Messages may be inserted to or removed from ω using the insert operator "$\triangleleft$" and the remove operator "$\triangleright$", respectively.
- μ denotes the method part. This part specifies the executing methods, and takes the form of a list of method identifiers, *m*, tagged by the name of the corresponding method name, e.g. *mn* (Methods have names, as well as method identifiers. These identifiers are used to identify the running instances of these methods. Several instances of the same method may be executed simultaneously and might be waiting for return results). Methods may be inserted to or removed from μ using the insert operator "$\triangleleft$" and the remove operator "$\triangleright$", respectively.
- *Req* denotes a method invocation request that is a record containing the following fields: target interface *tar* (an interface to the receiving role-figure), source role-figure *src* (the name of the sending role-figure), invoked method *met*, invoking method *ref*, return type *ret* (the return type of the invoking method), and argument list *arg* (may be used to pass the parameters to the invoked method).
- *Sig* denotes a signal that is a record containing the following fields: target interface *tar*, source role-figure *src*, signal name *name*, and argument list *arg*.
- *Ret* denotes a return that contains the following fields: target interface *tar*, source role-figure *src*, invoking method *ref*, and argument list *arg*.
- $\tilde{p}$ denotes an argument list of parameters $p_1, \ldots, p_n$ with types $t_1, \ldots, t_n$, respectively.

In the syntax and the definitions of the role-figure model we have applied the following changes and extensions to the ODP semantics [NS95]:

- The ODP interface definition has been extended with signals, which can be considered as announcement methods (described in Sec. 4.4.2).
- We have added capabilities to the definition of the role-figure object.
- We use names to identify capability, message, and method in a role-figure definition.

4.4.2 Interface definition

The interface definition is based on the type language and typing rules used in [NS95], especially the subtyping relation $\preceq$. [NS95] gives a list of inference rules to define a type theory that has a subtyping relation. These inference rules define type equality rules and subtyping rules. This type definition and type relations will be adopted in our role-figure model with minor modifications.

An interface type in the role-figure model will include signal definition and method signature definition, i.e. a record of several fields defining method signatures and signals accessible at this interface. In the role-figure model we will use the notation: ($\alpha \preceq \beta$), which means α is a subtype of β, and implies that the method and signal definitions of α are included in β.

An interface connects a role-figure to another role-figure. There are two sets of methods in an interface definition: the offered methods and the required methods. To understand these sets of methods assume α is an interface in role-figure ***a***, and that α connects ***a*** to role-figure ***b***. The offered methods are the methods offered by ***a*** and can be invoked by ***b***. The required methods are the methods required by ***a*** that ***b*** can perform. In our semantics, all role-figures offer the same set of methods, as these methods are the implementation of the support procedures that must exist in every actor object. Accordingly the two sets of methods in every interface type are similar.

Similarly, there are two sets of signals in an interface definition: the offered signals (signals that may be received) and the required signals (signals that may be sent). These sets of signals however can be different, as role-figures can send and receive different sets of signals with regard to different role-figures. In our semantics, as well as in the formal analysis we will not handle in detail behaviour and signal interactions. The example service functionality that we will experiment with will use few role-figures and few signals. For simplicity, we assume the two sets of signals in an interface definition are similar, i.e. a role-figure via this interface can send and receive the same set signals.

The implications of this simplification will not affect the correctness of our semantics and analysis regarding the mobility management. However, if the semantics will be aimed for analysing and validating the service functionality this simplification must be reconsidered. We believe that considering a more accurate subtype will only involve minor changes in the handling of role-figure communications and properties of role-figure configurations, which will be handled later in this chapter. **Table 4-4** gives the syntax for interface definition, or α, which can be regarded as another interpretation of **Table 4-1**.

Table 4-4. Syntax of Interface definition

$$\begin{aligned}
\alpha \quad &::= \quad \langle m_1 : methsig, \cdots m_n : methsig, sig_1, \cdots, sig_k \rangle \\
methsig \quad &::= \quad argument \rightarrow return \\
argument \quad &::= \quad Nil \mid \tilde{p} \\
\tilde{p} \quad &::= \quad (p_1 : t_1, \cdots, p_n : t_n)
\end{aligned}$$

Where:

▪ $m_1, .., m_n$	denote method names;
▪ *methsig*	denotes method signature, which specifies an argument list *argument*, which might be a list $\tilde{p}$ (of parameters $p_1,\ldots,p_n$ of types $t_1,\ldots,t_n$) or empty, and a return type *return*;
▪ $sig_1, .., sig_k$	denote signals with possible arguments, similar to those defined in the signal definition *Sig*, in Sec. 4.4.1, excluding the source and the target fields.

4.4.3 Behaviour definition

We have seen in Sec. 4.2.2 how the SDL semantics handle the behaviour definition in the SDL agent semantics. Also, in the TAPAS architecture an XML-based Extended Finite State Machine (EFSM) model has been both specified and implemented in [JA03]. In [SJS04] this EFSM has been used in the Service Management Architecture. In the SDL and the TAPAS approaches the behaviour definition is based on the operational semantics of the state machine model. During the execution of such state machine model the following information are required: current state information, state transition rules, triggering events at every state, tasks performed during state transitions (tasks are performed on the defined variable by the specification), signals sent during state transitions, and the corresponding next states in every state transition. In our role-figure model we handle the behaviour definition in a rather abstract way, and allow for different ways to specify the role-figure behaviour.

A behaviour definition β, as outlined in **Table 4-5**, has the following structure: a reference to a behaviour specification *B* (that contains the state transition rules including triggering events, tasks performed, signals sent, and next states), current state *St*, set of input signals *Sg* (trigger events for state transition at the current state), set of successor states *Sc* (the next states after the firing of the input signals), and set of stable states *Ss* (states where behaviour change is permitted). The behaviour specification *B* can be specified in an external module that is imported into the role-figure semantics. This importation of behaviour specification is the key point to handle behaviour change and role-figure mobility by our semantics. As a role-figure behaviour evolves and transits from one state to another all the *St, Sg, Sc,* and *Ss* change and reflect the status of the role-figure behaviour. For example, if the current state of a role-figure behaviour changes then both the set of input signals *Sg* and the set of successor states *Sc* change. Also, if a new behaviour specification *B* is imported then the current state is set to the initial state of this specification, as well as changing the set of stable states to the one specified in this specification. This level of detail is adequate to reason about the behaviour change and role-figure mobility management.

Table 4-5. Syntax of Behaviour definition

β	$::=$	$\langle B = b : behaviour, St = st : state, Sg = sg : \tilde{g}, Sc = sc : \tilde{s}, Ss = ss : \tilde{s} \rangle$
$\tilde{s}$	$::=$	$(state_1 \cdots, state_f)$
$\tilde{g}$	$::=$	$(sig_1, \cdots, sig_f)$

Where:

▪ *B*	EFSM behaviour specification
▪ $\tilde{s}$	Set of state names $state_1,\ldots,state_f$
▪ $\tilde{g}$	Set of signal names $sig_1,\ldots,sig_f$

4.4.4 Behaviour evolution

The behaviour of a role-figure is specified by a set of rewriting rules, each of which specifies a state transition. A general rewriting rule, indicated by

♣, will be used as a basis for this set of rewriting rules. This general rewriting rule specifies the state transitions of role-figure ***a***:

$$♣ \qquad \text{l: } \mathcal{A} \,||\, \mathcal{T} \,||\, \mathcal{Q} \,||\, \mathcal{M} \rightarrow \mathcal{A}' \,||\, \Sigma \,||\, \mathcal{T}' \,||\, \mathcal{Q}' \,||\, \mathcal{M}' \quad \text{if C}$$

In this rule *l* is used as a label. $\mathcal{A}$ and $\mathcal{A}'$ stand for role-figure ***a*** that evolves from $\mathcal{A}$ to $\mathcal{A}'$. Σ is the role-figures created in this rewriting rule, e.g. Σ can be $\mathcal{B}$ meaning that role-figure ***b*** was created. $\mathcal{T}$ and $\mathcal{T}'$ are *return* sets, $\mathcal{Q}$ and $\mathcal{Q}'$ are *signal* sets, $\mathcal{M}$ and $\mathcal{M}'$ are *request* sets, and C is a condition.

This general rewriting rule could be 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.

In this general rewriting rule, $\mathcal{A}$, $\mathcal{T}$, $\mathcal{Q}$, $\mathcal{M}$, $\mathcal{A}'$, Σ, $\mathcal{T}'$, $\mathcal{Q}'$, and $\mathcal{M}'$ should satisfy the following set of conditions (we will call these conditions the $♣|_{\text{conditions}}$):

(i) $\mathcal{T} \cap \mathcal{T}' = \mathcal{Q} \cap \mathcal{Q}' = \mathcal{M} \cap \mathcal{M}' = \varnothing$

returns, *signals*, and *requests* on the left hand side of the rewriting rules are all consumed in a transition.

(ii) $\mathcal{A} \notin \Sigma$

Role-figures are unique throughout a rewriting rule, as well as globally if explicitly defined in an additional rule.

(iii) Assume: $s = \langle tar = i : \alpha, src = a, met = m : mn, ref = n : mn, ret = r, arg = \tilde{p} \rangle$ is a *request* sent by role-figure ***a*** (to apply ♣ assume: ***a*** evolves from $\mathcal{A}$ to $\mathcal{A}'$ after the sending, and $s \in \mathcal{M}'$). Then the following hold:

- $i \in \mathcal{A}.Int$
- $\exists$***b***, ***b***$\in$ *RoleFigures* AND *i* is an interface to ***b***

- The signature of the invoked method m is included in the definition of α.
- If the invoked method m has the following method signature ($methsig_m ::= argument_m \rightarrow return_m$)
  THEN ($r =_{\text{def}} return_m$) AND ($\tilde{p} =_{\text{def}} argument_m$)
- $n \in \mathcal{A}'.Met$

(Sending a method invocation request means to suspend the invoking method until a return is received at the sending role-figure)

A method invocation request must be sent to an existing role-figure. Moreover, it must refer to an interface *i* that has the invoked method as part of its interface definition. This requires a matching in the argument list and return type. The matching is performed at the invoker's interface. The request will also carry a reference to the invoker (*src* and *ref* stand for source role-figure and method, respectively).

Assume now that role-figure ***b*** receives this *request* in another rewriting rule (to apply ♣ assume: ***b*** evolves from $\mathcal{B}$ to $\mathcal{B}'$ after the receiving, and $s \in \mathcal{M}$) then the following hold:

- $\exists\ [j : \alpha^\circ], j \in \mathcal{B}.Int$ and $\alpha \preceq \alpha^\circ$
- $s \in \mathcal{B}'.Que$
- If the invoked method m has the following method signature ($methsig_m ::= argument_m \rightarrow return_m$)
  THEN ($r =_{\text{def}} return_m$) AND ($\tilde{p} =_{\text{def}} argument_m$)

Receiving a method invocation request means to put this request into the queue of the role-figure (using the operation $\mathcal{B}.Que \lhd s$). Moreover, it should refer to an interface *j* that has the invoked method as part of its interface definition. This requires a matching in the argument list and return type at the invokee's interface.

(iv) Assume:

$s = \langle tar = i : \alpha, src = a, name = sig, arg = \tilde{p} \rangle$ is a *signal* sent by role-figure ***a*** (to apply ♣ assume: ***a*** evolves from $\mathcal{A}$ to $\mathcal{A}'$ after the sending, and $s \in \mathcal{Q}'$). Then the following hold:

- $i \in \mathcal{A}.Int$
- $\exists$***b***, ***b***$\in$ *RoleFigures* AND *i* is an interface to ***b***
- The signal name and its arguments are included in the definition of α.

A signal must be sent to an existing role-figure. Moreover, it must refer to an interface *i* that have the signal name and its arguments as part of its interface definition. The matching is performed at the invoker's interface. Also, when sending a signal we match the interfaces of the role-figures.

Assume now that role-figure ***b*** receives this *signal* in another rewriting rule (to apply ♣ assume: ***b*** evolves from $\mathcal{B}$ to $\mathcal{B}'$ after the receiving, and $s \in \mathcal{Q}$) then the following hold:

- $\exists\ [j : \alpha°], j \in \mathcal{B}.Int$ and $\alpha° \preceq \alpha$
- $s \in \mathcal{B}'.Que$

When receiving a signal we match the interfaces of the role-figures. Receiving a signal means putting it in the role-figure queue (using the operation $\mathcal{B}.Que \lhd s$).

(v) Assume:

$s = \langle tar = i : \alpha, src = a, ref = n : mn, arg = \tilde{p} \rangle$ is a *return* sent by role-figure ***a*** as a response to an invocation request received from another role-figure (to apply ♣ assume: ***a*** evolves from $\mathcal{A}$ to $\mathcal{A}'$ after the sending, and $s \in \mathcal{T}'$). Then the following hold:

- $i \in \mathcal{A}.Int$
- $\exists$***b***, ***b***$\in$ *RoleFigures* AND *i* is an interface to ***b***

When sending a return we only match the interfaces of the role-figures. The arguments of the return have already been matched by the method invocation request semantics in condition (iii).

Assume now that role-figure ***b*** receives this *return* in another rewriting rule (to apply ♣ assume: ***b*** evolves from $\mathcal{B}$ to $\mathcal{B}'$ after the receiving, and $s \in \mathcal{T}$) then the following hold:

- $\exists\,[j : \alpha^\circ]$, $j \in \mathcal{B}.Int$ and $\alpha^\circ \preceq \alpha$
- $n \in \mathcal{B}.Met$
- $s \in \mathcal{B}'.Que$

When receiving a return we match the interfaces of the role-figures. Also, when receiving a return it must be checked against the list of executing methods that have been suspended and are waiting for a return. Receiving a return means putting it in the role-figure queue (using the operation $\mathcal{B}.Que \lhd s$).

(vi) A *request*, *signal*, or *return* may be consumed from the queue of a role-figure:

$$\mathcal{A}.Que \rhd s$$

Consuming a *request*, *signal*, or *return* means to remove them from the queue of the role-figure.

(vii) In the rewriting rule ♣:

- Either ($\mathcal{A}.Int \subseteq \mathcal{A}'.Int$), ($\mathcal{A}.Int = \mathcal{A}'.Int$), or ($\mathcal{A}'.Int \subseteq \mathcal{A}.Int$) hold in one rewriting transition.
- Either ($\mathcal{A}.Met \subseteq \mathcal{A}'.Met$), ($\mathcal{A}.Met = \mathcal{A}'.Met$), or ($\mathcal{A}'.Met \subseteq \mathcal{A}.Met$) hold in one rewriting transition.
- Either ($\mathcal{A}.Que \subseteq \mathcal{A}'.Que$), ($\mathcal{A}.Que = \mathcal{A}'.Que$), or ($\mathcal{A}'.Que \subseteq \mathcal{A}.Que$) hold in one rewriting transition.
- Either ($\mathcal{A}.Cap \subseteq \mathcal{A}'.Cap$), ($\mathcal{A}.Cap = \mathcal{A}'.Cap$), or ($\mathcal{A}'.Cap \subseteq \mathcal{A}.Cap$) hold in one rewriting transition.

A role-figure's set of interfaces may increase (when the role-figure creates another role-figure, or creates an interface to existing role-figure), remain the same (e.g. when the role-figure sends signal or request), or decrease (when the role-figure destroys another role-figure). A role-figure's list of running methods may increase (when the role-figure sends an invocation request), remain the same, or decrease (when the role-figure receives a return result). A role-figure's queue may increase (when the role-figure receives a message), remain the same, or decrease (when it consumes a message). Similarly, a role-figure's capabilities may change (by adding or removing capabilities) or remain the same.

(viii) In the rewriting rule ♣, if *BehaviourChange* (discussed later) is not applied then the following hold:

- Either $(\mathcal{A}.Beh.St = \mathcal{A}'.Beh.St)$ or $(\mathcal{A}'.Beh.St \in \mathcal{A}.Beh.Sc)$

A role-figure may remain at the same state or carry out a state transition, in which its state will be one of the successor states.

(ix) If $\aleph = RoleFigures \cup \sum$ and $\Im = \bigcup_{A \in \aleph} A.Int$

then

- $\forall \boldsymbol{i}{:}\alpha,\ \boldsymbol{i} \in \Im$ and $\forall \boldsymbol{a},\ \boldsymbol{a} \in \aleph$
- $\forall r{:}Req,\ r.tar \in \Im$ and $r.src \in \aleph$
- $\forall s{:}Sig,\ s.tar \in \Im$ and $s.src \in \aleph$
- $\forall r{:}Ret,\ r.tar \in \Im$ and $r.src \in \aleph$

At any instance of our rewriting rules, a reference to a role-figure (e.g. ***a***) or a reference to an interface (e.g. ***i***) in any *Req*, *Sig* or *Ret* must be to an existing role-figure and to a defined interface.

(x) $\forall \mathcal{B} \in \sum$, then $\mathcal{B}.Met = \emptyset$ and $\mathcal{B}.Que = \emptyset$

Newly created role-figures will have neither suspended methods, nor messages in their queues. Behaviour, interfaces, and capabilities may be specified as part of the creation process, or the plug in phase.

The conditions (i) – (x) will constitute the semantic basis for the dynamics in the following section.

4.5 The Role-figure Model Dynamics

This section provides a discussion on various scenarios of the proposed role-figure model. The rewriting rules and their conditions throughout subsections 4.5.1 - 4.5.3 will be denoted: $\mathfrak{Dynamics}|_{\text{rules, conditions}}$.

These rewriting rules will handle the following: behaviour evolution (state transitions), communications (sending, receiving, and consuming of messages), and adaptability functionality (plug in, plug out, create interface, behaviour change, capability change, and role-figure move). These rewriting rules will apply conditions (i)-(x) from the previous section, and will further strengthen these conditions – references to these conditions will be explicitly included in the rules.

4.5.1 Behaviour evolution

A role-figure may perform a spontaneous transition. In our model we consider two such transitions, i.e. one that occurs due to an internal action, and another that takes place as a result of an interaction with other role-figures. In this section we consider the first one, while the latter will be dealt with in sections 4.5.2. A spontaneous transition, in this regard, may be expressed by the following instance of our general rewriting rule:

$$\mathcal{A} \rightarrow \mathcal{A}'$$

Applying the conditions (i)-(x) gives:

- (vii) $\mathcal{A}.Cap \subseteq \mathcal{A}'.Cap$
- (vii) $\mathcal{A}.Int = \mathcal{A}'.Int$, $\mathcal{A}.Que = \mathcal{A}'.Que$, $\mathcal{A}.Met = \mathcal{A}'.Met$
- (viii) $\exists s{:}state$, $s \in \{\mathcal{A}.Beh.Sc \cup \mathcal{A}.Beh.St\}$ so that $\mathcal{A}'.Beh.St = s$

These conditions suggest that at a spontaneous transition, a role-figure may change its capability definition as well as perform a state transition. However, interface, queue, and method definitions remain unchanged.

4.5.2 Communications

In this subsection we handle the communications of role-figures, i.e. sending, receiving, and consuming of messages. The conditions (iii), (iv), (v), and (vi) we presented earlier gave the semantics of sending and receiving requests, sending and receiving signals, sending and receiving returns, and consuming messages, respectively. Receiving meant to put the message (request, signal or response) into the role-figure's queue (e.g. $\mathcal{A}.Que \lhd s$). Consuming a message meant to remove it from the queue (e.g. $\mathcal{A}.Que \rhd s$). The distinction between receiving and consuming messages is crucial in the handling of the content of the queue of a moving role-figure. In our role-figure mobility management in chapter 3 we did not handle the queue content. As a simplification we will implicitly consider the consuming of a message as a part of receiving it in our dynamics, meaning that the message will be put into the queue and consumed immediately. To extend our dynamics and handle the queue content in the role-figure mobility this simplification must be reconsidered.

Also, in the remaining part of the dynamics the equality of the role-figure definition parts on the two sides of the rewriting rule will be assumed if not otherwise mentioned, i.e. the parts of $\mathcal{A}$ and $\mathcal{A}'$ are unchanged in a rewriting rule if not explicitly changed.

Sending and receiving requests
A role-figure (invoker) may invoke a method in another role-figure through sending a method invocation request via the appropriate interface. The invoking method will be suspended and a reference is added to the role-figure's method definition. Requests, denoted *req*, may be sent and received by the following rewriting rules:

Sending: $\mathcal{A} \rightarrow \mathcal{A}' \parallel req$

Receiving: $\mathcal{A} \parallel req \rightarrow \mathcal{A}'$

Applying the conditions (i)-(x) gives:

- (iii) Sending: $\mathcal{A}'.Met = \mathcal{A}.Met \lhd req.ref$
- (vii) Sending:

$\mathcal{A}.Int=\mathcal{A}'.Int, \mathcal{A}.Beh=\mathcal{A}'.Beh, \mathcal{A}.Cap=\mathcal{A}'.Cap, \mathcal{A}.Que=\mathcal{A}'.Que$

- (vi) Receiving: $\mathcal{A}'.Que = \mathcal{A}.Que \lhd req$
- (vii) Receiving:
  $\mathcal{A}.Int=\mathcal{A}'.Int, \mathcal{A}.Beh=\mathcal{A}'.Beh, \mathcal{A}.Cap=\mathcal{A}'.Cap, \mathcal{A}.Met=\mathcal{A}'.Met$

Sending and receiving signals
Similar to the sending and receiving of requests, signals, denoted *sig*, may be sent and received by the following rewriting rules:

$$\text{Sending:}\quad \mathcal{A} \rightarrow \mathcal{A}' \parallel sig$$
$$\text{Receiving:}\quad \mathcal{A} \parallel sig \rightarrow \mathcal{A}'$$

Applying the conditions (i)-(x) gives:

- (vi) Receiving: $\mathcal{A}'.Que = \mathcal{A}.Que \lhd sig$

When a signal is received and ($sig \in \mathcal{A}.Beh.Sg$) it triggers a state transition:

- (viii) Receiving: $\mathcal{A}'.Beh.St \neq \mathcal{A}.Beh.St$ AND $\mathcal{A}'.Beh.St \in \mathcal{A}.Beh.Sc$

Sending and receiving returns

Returns, denoted *ret*, may be sent and received by the following rewriting rules:

$$\text{Sending:}\quad \mathcal{A} \rightarrow \mathcal{A}' \parallel ret$$
$$\text{Receiving:}\quad \mathcal{A} \parallel ret \rightarrow \mathcal{A}'$$

- (vi) Receiving: $\mathcal{A}'.Que = \mathcal{A}.Que \lhd ret$
- (v) Receiving: if $ret.ref \in \mathcal{A}.Met$ and $ret.ref \in R = \{ \bigcup_{r \in \mathcal{A}.Que} r.ref \mid r{:}Ret \}$

  then $\mathcal{A}'.Met = \mathcal{A}.Met \rhd ret.ref$

The second condition implies that if a return reference at a queue meets the terms of a suspended method return then this method will be removed from *Met* definition part.

4.5.3 Adaptability functionality

In **Table 4-6** we define method invocation requests for 6 particular methods for the support functionality.

Table 4-6. Syntax of method invocation requests

$$
\begin{aligned}
mr &\ :\ Req \\
mr &::= pi \mid po \mid ci \mid bc \mid cc \mid mo \\
pi.arg &::= (name, loc, beh : \beta, cap : \pi) \\
po.arg &::= (name) \\
ci.arg &::= (j_1 : \alpha_1, \cdots, j_n : \alpha_n) \\
bc.arg &::= (beh : \beta, cSt : state) \\
cc.arg &::= (p_1 : c_1, \cdots, p_n : c_n) \\
mo.arg &::= (loc)
\end{aligned}
$$

Where:

▪ *mr*	management request
▪ *pi*	*PlugInActor* request with role-figure name *name*, location *loc*, behaviour *beh*, and capability set *cap*
▪ *po*	*PlugOutActor* request to a role-figure *name*
▪ *ci*	*CreateInterface* request with interfaces $j_1,\ldots,j_n$
▪ *bc*	*BehaviourChange* request with new behaviour *beh*, and current state *cSt*
▪ *cc*	*CapabilityChange* request with capabilities $p_1,\ldots,p_n$
▪ *mo*	*RoleFigureMove* request to new location *loc*

We will refer to these arguments by using the record syntax, e.g. the name of the role-figure in the plug in request is: *pi.arg.name*.

Role-figure Plug in

This method plugs in a role-figure and gives its location, behaviour, and capabilities. It is requested by a role-figure to plug in another role-figure. To simplify this request we will hide the complex process of director play management, capability allocation, configuration management, etc. and describe it by a single rewriting rule. In this rule, we have a *pi* request sent by role-figure ***a*** at $\mathcal{A}$, which will evolve to $\mathcal{A}'$. On the right hand side, we create a role-figure ***b*** to become $\mathcal{B}$:

$$\mathcal{A} \parallel pi \rightarrow \mathcal{A}' \parallel \mathcal{B}$$

Later in the formal specification in chapter 5 we will show the director can control this request, as well as the capability management. Applying the conditions (i)-(x), as well as using the definitions of the invocation requests in **Table 4-6**, gives:

- (ii) $\mathcal{B} \in \Sigma$
- (vii) $\mathcal{A}.Int \subseteq \mathcal{A}'.Int$
  (add an interface in ***a*** that is connected to ***b***)
- $\mathcal{B}.Beh = pi.arg.beh$
- $pi.arg.name =_{\text{def}} \boldsymbol{b}$
- $pi.arg.loc =_{\text{def}}$ location **(*b*)**
  (assume ***b***'s location is given by location **(*b*)**)
- $pi.arg.cap \subseteq \mathcal{B}.Cap$
- (x) $\mathcal{B}.Met = \emptyset$ and $\mathcal{B}.Que = \emptyset$

These conditions illustrate that a role-figure that creates another one will add an interface to its interface definition. The created role-figure will get its behaviour definition from the *beh* argument in the *pi* request. The created role-figure's name and location are given in the *pi* request arguments *name* and *loc*, respectively. It also adds the *cap* argument into its capability definition. Condition (x) holds for *Que* and *Met* parts of the newly created role-figure. Based on the arguments of the *pi* request many different conditions could be applied, e.g. a PlugIn request without interface arguments leaves the interface definition unchanged.

Role-figure Plug out

A role-figure may initiate a request to plug out another role-figure by the following rewriting rule. The following rewriting rule implies that the *po* request has been arrived at a role-figure ***a*** *that* is at $\mathcal{A}$:

$$\mathcal{A} \parallel po \rightarrow \emptyset$$

A role-figure, which receives and consumes this request, disappears and its instance is terminated so all its parts become $\emptyset$. Removing interfaces to the terminated role-figure should be handled at the connected role-figures. The following rewriting rule, on its left hand side, implies that a connected role-figure ***b*** will evolve from $\mathcal{B}$ to $\mathcal{B}'$:

$$\mathcal{B} \rightarrow \mathcal{B}'$$

Such that by applying the conditions (i)-(x) it gives:

- (vii) $\mathcal{B}'.Int \subseteq \mathcal{B}.Int$

  (remove an interface in ***b*** that is connected to ***a***)

Create Interface

This is a request to ask a role-figure to create interface(s) defined in the *ci* argument(s). The following rewriting rule implies that the *ci* request will be consumed by a role-figure ***a** that* is at $\mathcal{A}$ and will evolve to $\mathcal{A}'$:

$$\mathcal{A} \parallel ci \rightarrow \mathcal{A}'$$

- $\mathcal{A}'.Int = \mathcal{A}.Int \overset{n}{\underset{i=1}{\lhd}} ci.j_i$

This adds all the interfaces in the *ci* request 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. The semantics of the interface type theory mentioned previously could be used here to further reason about interface creation in the role-figure model semantics.

Behaviour Change

A behaviour change implies that a role-figure will be assigned a different behaviour than its original one, and a different current state than its current state. The following rewriting rule implies that the *bc* request will be consumed by a role-figure ***a** that* is at $\mathcal{A}$ and will evolve to $\mathcal{A}'$:

$$\mathcal{A} \parallel bc \rightarrow \mathcal{A}'$$

- If $En_{bc}(\mathcal{A}, \mathcal{A}.Beh.St)$ then $\mathcal{A}'.Beh.B = bc.beh$

  where $En_{bc}(\mathcal{A}, \mathcal{A}.Beh.St) \subseteq Actors \times States$

  and $En_{bc}(\mathcal{A}, \mathcal{A}.Beh.St)$ = TRUE if $\mathcal{A}.Beh.St \in \mathcal{A}.Beh.Ss$

  else FALSE

- If $En_{bc}(\mathcal{A}, \mathcal{A}.Beh.St)$ then $\mathcal{A}'.Beh.St = bc.cSt$

The predicate $En_{bc}(\mathcal{A}, \mathcal{A}.Beh.St)$ provides a Boolean expression for enabling a behaviour change in the stable states (from Sec. 4.4.3: states where behaviour change is permitted). According to this rewriting rule, the actor behaviour part definition will be replaced by that one received in *bc*, and the current state $\mathcal{A}'.Beh.St$ will be set to *bc.cSt.*

Capability Change

Capabilities in the role-figure model are meant for providing an overall basis for resource management. Allocating capabilities is a function of availability at the location of the role-figure where they are requested. Capability definition of an instantiated role-figure may be changed by a *CapabilityChange* method. The following rewriting rule implies that the *cc* request will be consumed by a role-figure ***a*** *that* is at $\mathcal{A}$ and will evolve to $\mathcal{A}'$.

$$\mathcal{A} \parallel cc \rightarrow \mathcal{A}'$$

- $\mathcal{A}'.Cap = \mathcal{A}.Cap \overset{n}{\underset{i=1}{\lhd}} cc.p_i$

This adds all the capabilities in the *cc* request to the role-figure's capability definition, *Cap.*

Role-figure Move

Role-figure mobility is the movement of instantiated role-figures. A move method is equivalent to a sequence of *pi*, *cc*, *ci*, *bc* and *po* methods. *mo* request supplies the new location to plug in, while its interface, behaviour, capability, queue, and method definition are all derived from the original role-figure instance.

The following rewriting rule specifies a role-figure move. We imply that the *mo* request will be consumed by a role-figure ***a*** *that* is at $\mathcal{A}$ and will evolve to $\mathcal{A}'$:

$$\mathcal{A} \parallel mo \rightarrow \mathcal{A}'$$

Such that by applying the conditions (i)-(x) it gives:

- (vii) $\mathcal{A}'.Int \subseteq \mathcal{A}.Int$, $\mathcal{A}'.Cap \subseteq \mathcal{A}.Cap$
- (viii) $\mathcal{A}'.Beh = \mathcal{A}.Beh$
- (x) $\mathcal{A}'.Que = \emptyset$, $\mathcal{A}'.Met = \emptyset$

These conditions assume a *pi* method at the new location using the original role-figure's behaviour obtained by the *bc* request. Capability and interface definitions can be updated by applying the *cc* and *ci* methods. The role-figure instance at the original location is terminated by a *po* method. On the other hand, queue and method part definitions of the moved role-figure will be empty.

4.6 The Role-figure Model Properties

The role-figure model is aimed to reason about the structure and the behaviour of role-figures (or the service systems realized by cooperating role-figures). In the semantics of the role-figure model, a role-figure configuration was defined as a set of asynchronously interacting role-figures. In such configurations role-figures may be dynamically created, interact with each other by sending and receiving messages, change their definitions parts (interfaces, behaviour, capabilities, etc.), be terminated, or move to new locations.

In a role-figure configuration there are certain requirements that need to be verified to ensure the correct operation of the service system as a whole. This verification process takes place at the service system design phase. It can identify design errors, and thus can improve the service system at early design phases.

In distributed systems, there are two main correctness requirements:

- *Safety* properties: means bad things don’t happen during the execution of a program.
- *Liveness* properties: means good things that capture the requirements of the system do happen during the execution of a program.

Safety and liveness properties were first introduced by Lamport in [Lam77]. A formal definition for safety and liveness properties was proposed by Alpern and Schneider in [AS84]. [AS84] states that a property (a set of sequences of program states) holds for a program if the set of state sequences defined by the program (program executions) is contained in the property. Examples of safety properties include mutual exclusion, deadlock freedom, and partial correction. The bad thing in these properties is: two processes executing in critical sections at the same time, deadlock, and terminating in a state not satisfying the postcondition after having been started in a state that satisfies the precondition, respectively [AS84]. Examples of liveness properties include termination and guaranteed service. The good thing in these properties is: completion of the final instruction, and receiving and satisfying service requests, respectively [AS84].

There are also state and path properties defined by Holzmann in [Hol04]. A state property proves that a state with property *P* is reachable or not. A path property proves that a certain sequence of states is executable or not. Every property, e.g. path or state, is the intersection of a safety property and a liveness property [AS84].

We list three main properties that are distinctive in our role-figure model. First we elaborate on the role-figure configuration definition presented in the semantics of the role-figure model in Sec. 4.4.1.

Based on the definitions in Table 4-2 and Table 4-3, a role-figure configuration evolves as an assembly of role-figures and messages, such that each role-figure is at certain state and that messages are sent and consumed by role-figures. Also, every interface in any of the role-figures is connected to another interface in another role-figure obeying the same interface type definition. Therefore, an elaborated definition of a role-figure configuration is as follows (in the definition of the role-figure configuration and the definitions of its properties we will use the letters M, N, O, P to denote the numbers of role-figures and messages, and will use the letters *i*, *j*, *k*, *l* as indices to range over these numbers. The letter *Q* will be used to denote the number of transitions a configuration goes through, while the index *q* will range over *Q*):

$rfc = \{\mathbf{a}_1,\ldots,\mathbf{a}_\mathrm{M}, \mathbf{g}_1\ldots,\mathbf{g}_\mathrm{N}\}$

Such that

$$\begin{cases} rfc \xrightarrow{\mathbf{g}_{rfc}} rfc^1 \xrightarrow{\mathbf{g}_{rfc^1}} \cdots rfc^q \xrightarrow{\mathbf{g}_{rfc^q}} \cdots, & \mathbf{g}_{rfc} \in \{\mathbf{g}_1 \cdots \mathbf{g}_\mathrm{N}\} \\ \forall \mathbf{a}_i : \quad \mathcal{A}_i \to \mathcal{A}_i^1 \to \cdots \mathcal{A}_i^q \to \cdots, & 1 < i < \mathrm{M} \\ \forall \mathcal{A}_i : \quad \mathcal{A}_i = \langle Int_i, Beh_i, Cap_i, Que_i, Met_i \rangle, & 1 < i < \mathrm{M} \\ \forall [\mathbf{k} : \alpha_\mathbf{k}] \in \mathcal{A}_i.Int_i : \quad \exists [\mathbf{l} : \alpha_\mathbf{l}] \in \mathcal{A}_j.Int_j \wedge \alpha_\mathbf{l} \preceq \alpha_\mathbf{k}, & 1 < i < \mathrm{M} \end{cases}$$

rfc is well-formed iff $\clubsuit|_{\text{conditions}} \wedge \mathfrak{Dynamics}|_{\text{rules, conditions}}$

This definition specifies role-figure configuration as an assembly of role-figures, $\mathbf{a}_{1,\ldots,}\mathbf{a}_\mathrm{M}$, and messages, $\mathbf{g}_{1,\ldots,}\mathbf{g}_\mathrm{N}$. The role-figure configuration, *rfc*, evolves through $rfc \to rfc^1 \to \ldots\ rfc^q \to \ldots$, and in every transition it consumes a message that exists in the current configuration, e.g. $\mathbf{g}_{rfc}$ that is one of *rfc*'s messages. New messages can be generated in these transitions, hence no further conditions are applied on $\mathbf{g}_{rfc^1}\ldots\mathbf{g}_{rfc^q}$ as for $\mathbf{g}_{rfc}$, which can only be in $\mathbf{g}_{1,\ldots,}\mathbf{g}_\mathrm{N}$ (*q* gives the number of transitions a configuration has gone through at a particular time).

Every role-figure, $\mathbf{a}_i$, evolves through $\mathcal{A}_i$ (denoted $\mathcal{A}_i \to \mathcal{A}_i^1 \to \mathcal{A}_i^2 \to \ldots$ where *i* is used as an index to range over the number of role-figures, M, assuming that it is not plugged out during any of the transitions of the configuration so far). For every role-figure configuration, the structure of the different role-figure states and the role-figure connections need to be valid (the connection between two role-figures at $\mathcal{A}_i$ and $\mathcal{A}_j$ should obey the subtype relation, $\alpha_{\mathbf{l}} \preceq \alpha_{\mathbf{k}}$, between their connected interfaces, **l** of type $\alpha_{\mathbf{l}}$ and **k** of type $\alpha_{\mathbf{k}}$, [**l**: $\alpha_{\mathbf{l}}$] and [**k**: $\alpha_{\mathbf{k}}$], respectively).

The role-figure configuration is considered ***well-formed*** if and only if it obeys (behaves and evolves according to) the rules and conditions constructed in the role-figure semantics. Well-formedness ensures that every role-figure is evolvable, the role-figure configuration has correct role-figure connections, and each role-figure has correct structure.

Following are the three role-figure configuration properties: *Pluggability*, *Consummability*, and *Playability*.

Pluggability This property proves that a role-figure can be plugged in at certain location. This property demonstrates a safety property, as a role-figure that fails to plug in is considered as a bad thing. The way to claim this property is by ensuring that the consumption of a plug in request has achieved the plug in of a role-figure at the appropriate location. It is also required to prove that the required capabilities and behaviour of the created role-figure satisfy the requirements of the plug in request. This property is defined by:

Pluggability:

$$\boldsymbol{P}_{pluggability} =_{def}$$

$$\forall\ rfc = \{\mathbf{a}_1, \ldots, \mathbf{a}_{\mathrm{M}}, \mathbf{g}_1, \cdots, \mathbf{g}_{\mathrm{N}}, \mathbf{g}_{plugin}\},\ \mathbf{g}_{plugin} = pi(\mathbf{a}_{new}, loc_i, beh_i, capset_i),$$

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

$$\text{such that} \begin{cases} rfc' = \{\mathbf{a}_1, \ldots, \mathbf{a}_{\mathrm{M}}, \mathbf{g}_1, \cdots, \mathbf{g}_{\mathrm{N}}\} \\ \mathcal{A}_{\mathbf{a}_{new}} = < \mathrm{Beh} = < \mathrm{B} = pi.beh_i >, \mathrm{Cap} = pi.capset_i > \\ \mathrm{location}(\mathbf{a}_{new}) = pi.loc_i \end{cases}$$

The plugged in role-figure at location loc_i and $rfc' = \{\boldsymbol{a}_1, \ldots, \boldsymbol{a}_{\mathbf{M}}, \mathbf{g}_1 \ldots, \mathbf{g}_{\mathrm{N}}\}$ constitute together a role-figure configuration after consuming the plug in request (the plug in request is consumed according to the rules defined in Sec. 4.5).

Consummability This property proves that all messages generated by the role-figures during the execution of the rewriting rules will eventually be consumed. This property demonstrates a liveness property, as unconsumed messages don't generate an erroneous situation. This property does not imply that consumed messages are consumed by the role-figures they intended for:

Consumeability:

$$\boldsymbol{P}_{consummability} =_{def}$$

$$\forall\ rfc = \{\mathbf{a}_1, \ldots, \mathbf{a}_{\mathrm{M}}, \mathbf{g}\},$$

$$rfc \xrightarrow{\mathbf{g}} rfc^1 \xrightarrow{\mathbf{g}_{rfc^1}} \cdots rfc^Q \xrightarrow{\mathbf{g}_{rfc^Q}} rfc_{\mathrm{terminal}}$$

$$\text{such that} \begin{cases} \forall rfc^i : & rfc^i = \{\mathbf{a}_1, \ldots, \mathbf{a}_{\mathrm{N}}, \mathbf{g}_1, \ldots, \mathbf{g}_{\mathrm{P}}\}, \quad 1 < i < Q \\ rfc_{\mathrm{terminal}} = \{\mathbf{a}_1, \ldots, \mathbf{a}_{\mathrm{O}}\} & \\ \forall \mathbf{a}_j \in rfc_{\mathrm{terminal}} : & \mathcal{A}_j.Que_j = \phi, \quad 1 < j < \mathrm{O} \end{cases}$$

This is the consummability property with respect to a configuration that has only a single message. The configuration consumes messages and evolves based on the actions that will occur after the consumption; e.g. 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 system terminal states of a configuration and checks if they contain any unconsumed message (system terminal states are those states of the system where no rewriting rule could be applied any further). The consummability property holds if the terminal configuration, $rfc_{terminal}$, has no messages, and that every role-figure in this configuration has no messages in its queue, $\mathcal{A}_j.Que_j = \phi$.

Regarding the applicability of the consummability property, it is only possible to be verified at terminal states. We added this constraint because it is impossible to verify this property without it. This implies that we need to execute the system till it terminates, which is sometimes not possible.

Playability This property proves that a role-figure, after its plug in phase, is playing or performing according to its predefined role. This property also demonstrates a liveness property. This can be achieved by verifying 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 playability we only consider messages that are signals. Requests and returns are not considered since they don't trigger state transitions.

There can be two types of this property. Weak Playability proves that a role-figure has begun performing once it has been plugged in. So at least a single state transition has been performed. Strong Playability requires that a role-figure is proved to be free of non-progress cycles.

In the following definition, the weak playability, $\boldsymbol{P}_{wplayability}$, shows that a consumption of a signal by a plugged in role-figure has been accomplished. This signal, $\mathbf{g}_k$, is one of the input signals of the role-figure, $\mathcal{A}_i'.Beh_i.Sg_i$. Here a signal does not necessarily trigger a transition from the current state to another state. A role-figure may remain in the same state after consuming such a signal.

Weak Playability:

$\boldsymbol{P}_{wplayability} =_{def}$

$\forall\ rfc = \{\mathbf{a}_1, \ldots, \mathbf{a}_i, \ldots, \mathbf{a}_M, \mathbf{g}_1, \ldots, \mathbf{g}_N\},$

$$rfc \longrightarrow \ldots rfc' \xrightarrow{\mathbf{g}_k} rfc''$$

such that

$rfc' = \{\mathbf{a}_1, \ldots, \mathbf{a}_i, \ldots, \mathbf{a}_O, \mathbf{g}_1, \ldots, \mathbf{g}_k, \ldots, \mathbf{g}_P\}, \quad \mathbf{g}_k \in \mathcal{A}_i'.Beh_i.Sg_i,$

$rfc'' = \{\mathbf{a}_1, \ldots, \mathbf{a}_i, \ldots, \mathbf{a}_O, \mathbf{g}_1, \ldots \mathbf{g}_{k-1}, \mathbf{g}_{k+1}, \ldots, \mathbf{g}_P\}$

Based on our definition of pluggability and weak playability, it can be clearly seen that weak playability implies pluggability, since a role-figure that is playing or consuming signals and performing state transitions is considered to be plugged in.

Strong Playability:

The following definition of the strong playability, $\boldsymbol{P}_{splayability}$, adds another requirement to the weak playability. It demands that a progression is achieved in the behaviour evolution of the role-figure. The difficulty of proving this property is two fold. First, it requires knowledge of the role-figure state, which cannot be obtained by an external observation. Second, it requires knowledge of whether a state in the behaviour specification is a progress state or not.

$$\boldsymbol{P}_{splayability} =_{def}$$

$$\forall\ rfc = \{\boldsymbol{a}_1,\ldots,\boldsymbol{a}_i,\ldots,\boldsymbol{a}_{\mathrm{M}},\mathbf{g}_1,\ldots,\mathbf{g}_{\mathrm{N}}\}$$

$$rfc \longrightarrow \cdots rfc^{q} \xrightarrow{\mathbf{g}_{rfc^{q}}} \cdots rfc^{Q-1} \xrightarrow{\mathbf{g}_{rfc^{Q-1}}} rfc^{Q} \qquad 1 < q < Q$$

$$\text{such that} \begin{cases} rfc^{q} = \{\boldsymbol{a}_1,\ldots,\boldsymbol{a}_i,\ldots,\boldsymbol{a}_{\mathrm{O}},g_1,\ldots,g_{\mathrm{P}}\}, & 1 \le q \le Q \\ \mathcal{A}_i \longrightarrow \mathcal{A}_i^{1} \longrightarrow \cdots \mathcal{A}_i^{q} \rightarrow \cdots \mathcal{A}_i^{Q}, & 1 < q < Q \\ \exists st_i \in \{\mathcal{A}_i^{1}.Beh_i\,.St_i\,,\cdots,\mathcal{A}_i^{Q}.Beh_i\,.St_i\,\}, \quad st_i \in \boldsymbol{a}_i\,|_{Beh.\mathrm{Pr}ogress} & \end{cases}$$

The strong playability 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 playability is the denotation, $\boldsymbol{a}_i\,|_{Beh.Progress}$, which stand for the progress states in the role-figure behaviour.

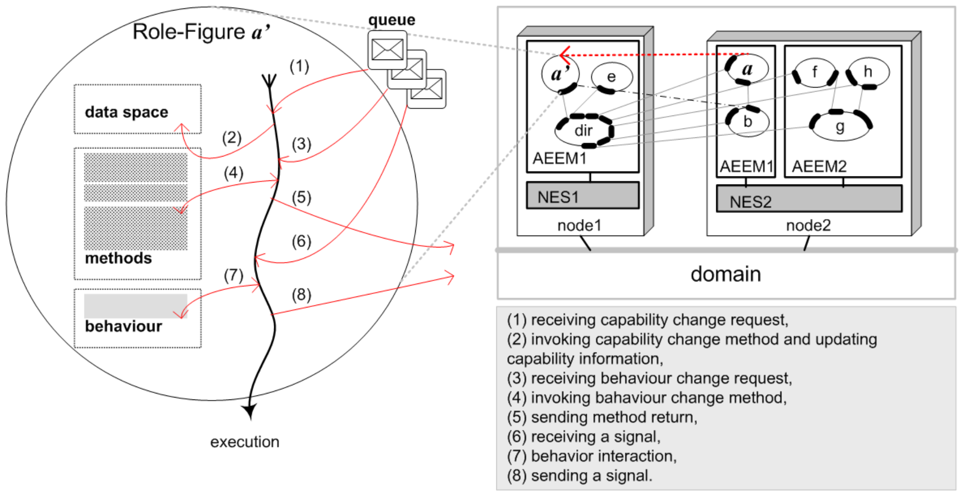
Based on our definition for weak playability and strong playability, it can be clearly seen that strong playability implies weak playability, since a role-figure that evolves through progress states is performing at least one state transition.

The role-figure mobility management mechanisms described in the previous chapter come with their own properties. They present a mechanism for moving role-figures based on a set of requirements maintained in a mobility strategy. The verification of these mechanisms is possible using these three basic properties – the property of one of the role-figure mobility management mechanisms is an intersection of these three properties. We believe that these three basic properties are adequate for the discussion on the role-figure mobility management in our architecture due to the nature of the role-figure itself (i.e. consume messages, plug in roles, and play roles). The proof of this statement is not handled here, and left out as an open issue. This proof is believed to be derivable in a similar manner to the study on “defining liveness” in [AS84].

4.7 Discussion

In this section we give a discussion on a specific mobility management example. This discussion is aimed to show how we can use our semantics. Figure 4-6 shows an example of a system that executes a number of role-figures in a domain of two nodes. It demonstrates a movement of role-figure ***a*** from *node2* to *node1*. The role-figure obtains a different name at *node1*, i.e. ***a'***. This indicates that it is another instance of the same role-figure. Part of the execution of role-figure ***a'*** is explained. As part of the movement process, capabilities and behaviour definition are updated. These updates are conducted using methods accessible in the role-figure. Also, an example interaction with other role-figure is shown by sending and receiving signals. Interfaces, queue, AEEM and NES modules are similar to those used in Figure 4-5.

A message sequence diagram for this system is shown in Figure 4-7.



(1) receiving capability change request,
(2) invoking capability change method and updating capability information,
(3) receiving behaviour change request,
(4) invoking bahaviour change method,
(5) sending method return,
(6) receiving a signal,
(7) behavior interaction,
(8) sending a signal.

Figure 4-6 Example of mobility management

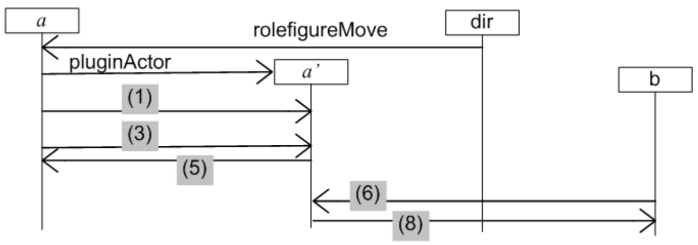


Figure 4-7 A message sequence diagram of the example

The signals used in Figure 4-7, in step (6) and step (8), are exchanged with role-figure ***b*** in the system. In the following we show the application of our semantics to specify part of this example. We assume using a role-figure mobility management mechanism that does not involve a mobility manager, e.g. *RMM3*. We also assume using a mobility strategy that does not have requirements regarding queue content and executing methods, i.e. applying the design rules *RD9* (*a*) and *RD9* (*q*) from Sec. 3.4.4.2. The example semantics includes the director, *dir*, as well as the other role-figures, ***a***, ***b***, ***e***, ***f***, ***g***, and ***h***. Only the details of role-figure ***a*** are shown. The notation j_x means an interface to role-figure *x*.

$dir = <Int = \{ j_e ,j_a j_b ,j_f j_g ,j_h \}>$

$e = <Int = \{ j_{dir} \}>$

$a = <Int = \{ j_{dir} ,j_b \}, Beh = <B = bh1, St = st1, Sg = sg1, Sc = sts1, Ss = sts2>,$

$\quad Cap = \{cap1, cap2\} , Que = \emptyset, Met = \emptyset>$

$b = <Int = \{ j_{dir} ,j_a \}>$

$f = <Int = \{ j_{dir} ,j_g \}>$

$g = <Int = \{ j_{dir} ,j_f ,j_h \}>$

$h = <Int = \{ j_{dir} ,j_g \}>$

To illustrate the movement of role-figure ***a***, we need to show the transition from this configuration of role-figures containing ***a*** into another configuration of role-figures containing ***a'***:

$$mo(a,NES1.AEEM1) \parallel dir \parallel a \parallel e \parallel b \parallel f \parallel g \parallel h \rightarrow dir \parallel a' \parallel e \parallel b \parallel f \parallel g \parallel h$$

On the left hand side there exists a *rolefigureMove* request with the role-figure instances. Only the target and the argument of this request are shown. The other fields of the structure, e.g. the source, are hidden. This move request is assumed to be sent by any of the other role-figures. The structure of the other requests, *pi*, *cc*, *ci*, and *bc*, is also hidden in the transitions of this example. The move request will instruct role-figure ***a*** (in *NES2.AEEM1*) to move to another location of performance (to *NES1.AEEM1*). It will become role-figure ***a'***. This transition consists of several applications of the specified rewriting rules (in Sec. 4.5). We show the transitions applied by role-figure ***a*** in the following path of transitions – we simplify the transitions by defining the main parts of the structure of role-figure ***a*** in the transitions themselves (the structure of ***a*** after the first transition is not changed):

$mo(a, NES1.AEEM1) \parallel a = <Int = \{j_{dir}, j_b\}, Beh = <B = bh1, St = st1>, Cap = \{cap1, cap2\}> \rightarrow$

$\rightarrow a \parallel pi(a', NES1.AEEM1, bh1) \rightarrow$

$\rightarrow a \parallel a' = <Int = \{j_{dir}, j_a\}, Beh = <B = bh1>> \parallel cc(cap1, cap2)) \rightarrow$

$\rightarrow a \parallel a' = <Int = \{j_{dir}, j_a\}, Beh = <B = bh1>, Cap = \{cap1, cap2\} > \parallel ci(j_b) \rightarrow$

$\rightarrow a \parallel a' = <Int = \{j_{dir}, j_a, j_b\}, Beh = <B = bh1>, Cap = \{cap1, cap2\} > \parallel bc(bh1, st1) \rightarrow$

$\rightarrow a \parallel a' = <Int = \{j_{dir}, j_a, j_b\}, Beh = <B = bh1, St = st1>, Cap = \{cap1, cap2\}> \mathrm{po}(a) \rightarrow$

$\rightarrow a' = <Int = \{j_{dir}, j_a, j_b\}, Beh = <B = bh1, St = st1>, Cap = \{cap1, cap2\} >$

CHAPTER **5**

The Formal Analysis

5.1 Introduction

THE ROLE-FIGURE model presented in Chapter 4 provided an abstract model for the implemented functionality of role-figures. The model was a preliminary step to the formal specification, analysis, and validation of role-figures. In this chapter we present the Maude formal specification of the role-figure model. We also use this specification to conduct formal analysis and validation of the proposed solution for the role-figure mobility management.

We used Maude, the formal language and tool supporting rewriting logic, to achieve two major goals. First, it will be used to formally construct the various viewpoints at various levels of detail of the role-figure model. Second, the Maude's reflective features are deployed to model and check the correctness of the proposed solution for the role-figure mobility management.

This Maude specification of the role-figure model is presented in Sec. 5.2. Experiences with analysing and validating the Maude specification are discussed in Sec. 5.3.

5.2 The Maude Specification

Maude is based on the rewriting logic principles. In this section we present our Maude specification of the role-figure model. This section show the main constructs of the Maude language in Sec. 5.2.1. Then, in Sec. 5.2.2, a set of assumptions on the role-figure model are considered. The Maude specification of the role-figure model follows in Sec. 5.2.3. This specification includes the specification of the computing architecture functionality, as well as the semantics and the dynamics of the role-figure model. The service specification in Maude is presented at the end of the section.

5.2.1 Preliminaries on Maude

The following is not intended to be a Maude tutorial. It is just a brief introduction to the main syntax elements, which is used to highlight the role-figure model formalization efforts, both the structure of the formal role-figure model and the conducted formal analysis. The reader is advised to read the Maude manual [CDE03] for a detailed description, and to visit [Mau04] for the tool and a tutorial of the language.

Maude is a high-performance language and system supporting both equational and rewriting logic computations [CEL96]. It has been developed at SRI International [Mau04]. As a programming language, Maude makes it possible to specify programs and routines. In Maude the basic units of specification and programming are called modules. A typical Maude program consists of a hierarchy of modules.

In Maude there are three kinds of modules:

- *Functional modules*: are theories in membership equational logic, which is a sub-logic of the rewriting logic [Mes92]. Maude functional modules consist usually of equations and membership assertions.
- *System modules*: are pairs (***T***, ***R***), where ***T*** is a membership equational theory, and ***R*** is the collection of labelled and possibly conditional rewriting rules. Evolution of the system, or shortly rewriting of the system, happens modulo the equational axioms in **T**. Controlling and effecting the execution of these rewriting rules is achieved by the so-called strategies.
- *Object-oriented modules*: are the Maude specifications for concurrent object-based systems. These modules specify predefined configurations that declare sorts representing objects, messages, and the rules of interactions.

Functional, system, and object-oriented modules use the following syntax as a definition, respectively (Maude keywords are written with boldface):

```
fmod REAL-NUMBERS is ... endfm
mod VENDING-MCHINE is ... endm
omod ROLE-FIGURE is ... endom
```

The modules are defined between the beginning and the end of this declaration. Also modules can be imported by other modules, e.g. using the keyword “protecting”.

The declaration part of a Maude module includes definitions of types, operators, and variables. Types in Maude are called sorts, and defined using the keywords sort and sorts. They are partially ordered via a subsort relation, which is denoted by < (period, . , is used to indicate the end of each declaration):

```
sort A .                *** definition of one sort
sorts B C .             *** definition of more than one sort
subsort A < B < C .     *** subsort relation
```

Operators in Maude are declared with the keyword op and ops (definition of more than one operator). Operators have name, arguments, and return result. An argument may be represented by an underscore, _ , in the syntax. The list of arguments may be empty (defining a constant), while the operator itself may be declared in prefix or mixfix form, e.g.:

```
op null  : -> Nat .                   *** null is defined to be a constant
op add  : Nat Nat -> Nat [assoc] .    *** prefix form addition operator on natural numbers
op _+_ : Nat Nat -> Nat [assoc] .     *** mixfix form addition operator on natural numbers
```

assoc is an example of an attribute declaration standing for associativity. In Maude variables can be defined using the following syntax:

```
vars X Y : Nat .
```

A term in Maude is either: a constant, a variable, or an application of an operator on a list of argument terms, e.g.:

```
add X Y .     *** the add operator declared above used to add two variables, X and Y
X + Y .       *** the + operator declared above used to add two variables, X and Y
```

The Maude is based on the membership equational logic. This may be simply interpreted by the following two definitions: an equation is a declaration of the form (***t*** = ***t’***), where ***t*** and ***t’*** are terms, while membership is a declaration of the form (***t*** : ***s***), where ***t*** is a term and ***s*** is a sort. Equations in Maude can be unconditional or conditional (by which a condition is provided to control the interpretation and execution of the equations). Similarly, memberships can be unconditional or conditional. In both cases the conditions can either be a single equation, single membership, or a conjunction of equations and memberships. The following show some examples of unconditional equation eq, unconditional membership mb, conditional equation ceq, and conditional membership declarations cmb, respectively.

```
eq X + Zero = X .
mb add (X Y) : Nat .
ceq X + Zero = X if X =/= Zero.
cmb add (X Y) : Nat if X =/= Y.
```

The main operating principle of the Maude interpreter regarding the functional modules is to reduce or simplify terms by applying the given set of equations. By other words, a given term *t* is matched to every left hand side of the equations and memberships, and thereafter simplified or rewritten. This process ends when no further rewriting is possible, that's when ground terms are obtained – terms that cannot be simplified.

Rewriting rules in Maude have the following definition,:

```
rl [<label>] :    <term1>  =>  <term2>    [<statement attributes>] .
crl [<label>] :    <term1>  =>  <term2>    [<statement attributes>]
                  if <condition> [<statement attributes>] .
```

rl and crl are used to specify the unconditional and conditional rewriting rules, respectively. Both have a label, which is used to identify and trace the execution of the rules, as well as statements of the form:

```
<term1>  =>  <term2> .
```

The conditional rule is followed by a condition to control the execution of the rule, meaning that a rule is only executed if the condition holds. In Maude an object in a given state is represented as a term of the form:

<object : Class | attribute$_1$: value$_1$, … , attribute$_n$: value$_n$ >

An object in an Object-Oriented Maude module is defined as a class using the keyword Class, with attributes tagged with their values. Messages to communicate between objects are defined as Msg, and declared using the keyword msg, while a configuration in Maude is defined as Configuration, and it is an assembly of objects and messages, e.g.:

```
msg request_ : Oid -> Msg .                  *** Oid sort defines an object identifier
op multimsg__ : Oid Oid -> Configuration . *** a configuration of two objects
```

Finally, the tool support for Maude is built around a rewriting engine interpreter. By using the Maude interpreter's ***rewrite*** or ***rew*** command it is possible to rewrite a given initial conditions based on a set of rewriting rules given in a module.

5.2.2 Assumptions and simplifications

In our Maude specification we apply the following assumptions and simplifications over the semantics and dynamics of the role-figure model:

- The behaviour specification is described in Maude modules. These modules are imported by the role-figure model specification to plug in specific behaviours for role-figures. The current state of a behaviour specification is a variable in this specification.
- We assume a single and generic interface type in TAPAS. The definition of this interface type in a role-figure is abstracted by the name of the connected role-figure. (if different interface types are used between role-figures then this abstraction must be removed and the interface types must be used instead of the connected role-figures)
- We apply an abstract capability definition. There is no specialization over the different classes of the capabilities of the system (i.e. *functions*, *resources*, or *data*). There is a capability type and a capability set type. As such a role-figure may consume any capability and may add it to its set of capabilities.
- We use different plug in requests, e.g. plug in request without arguments, plug in request with initial capability and behaviour definitions, etc. The director controls the plug in of all role-figures. Similarly, the plug out request also is handled by the director.
- The queue of role-figures is not handled by the Maude specification. Therefore, messages in the queue of a moving role-figure will be lost. (To handle the queue an ordered set definition, a FIFO model, must be used to maintain the arriving messages to a role-figure. If the queue of a moving role-figure is handled during the movement, then only signal messages can be forwarded to the moved role-figure)
- The executing methods of a moving role-figure are not handled by the Maude specification. Therefore, these methods are always discarded. (To handle these methods a replica of the moving role-figure must be maintained until all executing methods finish their executions)
- We will use slightly different structure for the method invocation requests as was presented in the semantics of the role-figure model. In some cases the invoking method field will not be used, while the invoked method field will be abstracted by the name of the invocation request. Also in some method invocation requests we will add new argument. These alterations will be explicitly mentioned.
- In the role-figure mobility management we use the role-figure name and its location as the Location-of-Performance of a role-figure.

5.2.3 The Role-figure Model Specification

In this section we only give the main elements of our Maude specification (the complete Maude specification for the role-figure model is given in the module *RoleFigureModel* in Appendix IV. Two extension Maude modules are also included. The module *Capability* handles mainly the issues related to the allocation of capabilities in the system. The module *Mobile* handles issues related to mobility and behaviour change.

The role-figure model specification is an object-oriented module:

omod RoleFigureModel **is** … **endom**.

In this module, a role-figure is specified as a class with attributes. These attributes are the role-figure's location, interfaces, behaviour, capabilities, queue, and methods, denoted by *Location*, *Int*, *Beh*, *Cap*, *Que*, and *Met*, respectively. The supervisory role-figure Director has a simple structure, which only includes an interface definition part:

The role-figure and Director class definition

```
class  RoleFigure | Location : Loc, Int : OidSet, Beh : BehType, Cap : CapSet,
            Que : MesSet, Met : MetSet .
class  Director | Int : OidSet .
```

Role-figure and director use the following sort definitions:

Sort definitions

```
sorts Loc OidSet Content State StateSet Beh BehType Signal SignalSet Cap
            CapSet Mes MesSet Met MetSet .
subsort Oid < OidSet .          *** object definitions
subsort State < StateSet .     *** state definitions
subsort Signal < SignalSet . *** signal definitions
subsort Cap < CapSet .          *** capability definitions
subsort Met < MetSet .          *** method definitions
subsort Mes < MesSet .          *** message definitions
```

Loc defines location type, while Content defines message content type. These sorts are partially ordered via the subsort relation, e.g. a State is a subsort of StateSet.

Role-figure methods

In the Maude specification, the invocation requests of the main methods of the role-figure model are defined using messages. These requests follow the syntax defined in the role-figure model semantics, as well as apply the assumptions mentioned earlier in Sec. 5.2.2. Certain requests however have several message definitions. The reason is to achieve various levels of detail in the specification. The definitions of these messages are the following:

Method invocation request definitions

```
*** plugin request takes target, source, name, and location arguments
msg pi-tar_src_name_location_         : Oid Oid Oid Loc -> Msg .
*** plugin request takes target, source, name, location, behaviour and capability
*** arguments
msg pi-tar_src_name_location_with__ : Oid Oid Oid Loc BehType CapSet -> Msg .
*** plugout request takes target, source, and role-figure name arguments
msg po-tar_src_name_                  : Oid Oid Oid -> Msg .
*** create interface request takes target, source, and interface(s) arguments
msg ci-tar_src_j_                     : Oid Oid OidSet -> Msg .
*** behaviour change request takes target, source, and behaviour arguments
msg bc-tar_src_beh_                   : Oid Oid BehType -> Msg .
*** behaviour change request takes target, source, behaviour, current state, stable
*** states (used to control the behaviour change and move procedures), and
*** progress states (used to check the playability properties) arguments
msg bc-tar_src_beh____                : Oid Oid BehType State StateSet StateSet ->
          Msg .
*** capability change request takes target, source, and capability arguments
msg cc-tar_src_p_                     : Oid Oid CapSet -> Msg .
*** move request takes target, source, role-figure name, location and content
*** arguments
msg mo-tar_src_location__with_        : Oid Oid Oid Loc Content -> Msg .
*** move return takes target, source, role-figure name and location arguments
msg mor-tar_src_to__                  : Oid Oid Oid Loc -> Msg .
```

Beside these method requests, the following message definition is required. The first line defines a general message, which is mainly used as a signal to stimulate the behaviour of the role-figures. multimsg is used as a broadcast message that sends a message to a list of recipients.

Message definitions

```
msg msg_from_to_ : Content Oid Oid -> Msg .
op multimsg_from_to_ : Content Oid OidSet -> Configuration .
ceq multimsg M from A to (B N) =
        (msg M from A to B) (multimsg M from A to (N - B)) if not M == creInt .
```

A and B are Oid variables, N is an OidSet variable, and M is a Content variable. creInt is a constant used in the create interface rules.

Constants, Operations and Variables

The role-figure specification uses the following constants, operations, and variables:

Constants, Operations, and Equation definitions

```
*** constants definition
ops st1 st2 : -> State .                    *** states
ops sts1 sts2 : -> StateSet .               *** state sets
ops bh1 bh2 : -> Beh .                      *** behaviours
ops m remInt creInt moving : -> Content . *** "content" of some messages
ops director a a' b c e : -> Oid .          *** role-figures
ops n n' : -> OidSet .                      *** Oid sets
ops bht1 bht2 : -> BehType .                *** behaviour type
ops sig1 sig2 : -> Signal .                 *** signals
ops sigs1 sigs2 : -> SignalSet .            *** signal sets
ops cap1 cap2 : -> Cap .                    *** capabilities
ops caps1 caps2 : -> CapSet .               *** capability sets
ops met1 met2 : -> Met .                    *** methods
ops mets1 mets2 : -> MetSet .               *** method sets
ops mes1 mes2 : -> Mes .                    *** messages
ops mess1 mess2 : -> MesSet .               *** message sets
ops loc1 loc2 : -> Loc .                     *** locations
*** constants
op nil : -> OidSet .                        *** an empty set
op nilOid : -> Oid .                        *** a nil object
op nilcaps : -> CapSet .                     *** an empty capability set
op nilbht : -> BehType .                    *** an empty behaviour definition
op nilmess : -> MesSet .                     *** an empty message set
op nilmets : -> MetSet .                      *** an empty method set
*** Some set-operations:
op in : Oid OidSet -> Bool .                *** an Oid is in a set
op subseteq : OidSet OidSet -> Bool .       *** subset of a set
op _-_ : OidSet Oid -> OidSet .             *** Oid removed from set
*** Equations on sets:
eq A A = A .                                 *** concatenation of sets
eq in(A, B N) = A == B or in(A,N) .         *** in operation
eq in(A,nil) = false .                       *** in operation
eq subseteq(A N ,N') = in(A,N') and subseteq(N,N') . *** subseteq operation
eq subseteq(nil,N') = true .                *** subseteq operation
eq (A N) - A = N - A .                      *** further operations
ceq N - A = N if not in(A,N) .              *** further operations
```

The following variables are also used.

Variable declarations

```
*** variable declaration
vars A A' B C D : Oid .
vars N N' N'' : OidSet .
vars M : Content .
vars ST1 ST2 : State .
vars STS1 STS2 : StateSet .
vars BH1 BH2 : Beh .
vars BHT1 BHT2 : BehType .
vars SIG1 SIG2 : Signal .
vars SIGS1 SIGS2 : SignalSet .
vars CAP1 CAP2 : Cap .
vars CAPS1 CAPS2 : CapSet .
vars MET1 MET2 : Met .
vars METS1 METS2 : MetSet .
vars MES1 MES2 : Mes .
vars MESS1 MESS2 : MesSet .
vars LOC1 LOC2 : Loc .
```

Role-figures
A role-figure is expressed using the following notation, e.g. for the role-figure a with interfaces to the director and two other role-figures (b and c):

```
< a : RoleFigure | Location : loc1, Int : director b c, Beh : bht1, Cap : caps1, Que :
        mess1, Met : mets1 >
```

Role-figure Configurations
An example of a role-figure configuration (an assembly of role-figures and messages) is presented as follows.

role-figure Configuration example

```
(pi-tar director src a name e location loc1)
< director : Director | Int : a b c >
< a : RoleFigure | Int : director b c > < b : RoleFigure | Int : director > < c : RoleFig-
        ure | Int : director >
```

In this configuration a plug in request message (target, source, name, and location arguments), a director, and three role-figures (a, b, and c) exist. These role-figures have interfaces. Each role-figure has at least one interface with the director. The plug in request has been issued by the role-figure a and sent to the director. It requests the instantiation of a role-figure with the name e at the location loc1. Based on the desired level of detail these definitions may be further specified and extended to handle more complex configurations and behaviour patterns.

Role-figure Plug in
The role-figure plug in method instantiates a role-figure. It may have different utilizations by passing different arguments to the instantiated role-

figure. Plug in method is defined by a number of rewriting rules, e.g. a rewriting rule plug in request with or without initial capability and behaviour definitions.

We present two rewriting rules specifying the role-figure plug in semantics. The first rewriting rule, labelled [PlugInNoContentSemantics], specifies the plug in of a role-figure without capability and behaviour specification, while the second rewriting rule, labelled [PlugInwithCapBehSemantics], specifies the plug in of a role-figure with capability and behaviour specification.

role-figure plug in rewriting rules

```
*** A simple PlugIn by which the director is performing the plugin
crl [PlugInNoContentSemantics] :
  (pi-tar D src A name B location LOC1)
  < director : Director | Int : N > =>
  < director : Director | Int : B N > < B : RoleFigure | Location : LOC1, Int : director
          A, Beh : nilbht, Cap : nilcaps, Que : nilmess, Met : nilmets >
  (ci-tar A src director j B) if not in(B,N) and (D == director) .
```

```
*** this corresponds to PlugIn with cap/beh
crl [PlugInwithCapBehSemantics] :
  (pi-tar D src A name B location LOC1 with BHT2 CAPS2)
  < director : Director | Int : N > =>
  < director : Director | Int : B N > < B : RoleFigure | Location : LOC1, Int : director
          A, Beh : BHT2, Cap : CAPS2, Que : nilmess, Met : nilmets >
  (ci-tar A src director j B)  if (D == director) .
```

In the first rule, A, B and D are role-figure variables. A is the role-figure that issued the plug in request (pi-tar D src A name B location LOC1), while B is the role-figure to be instantiated at location LOC1. If the rule is executed, it results the instantiation of the B role-figure at location LOC1 with empty definitions – empty capability, behaviour, queue and method definitions. Its interface definition includes an interface to role-figure A and a default interface to the director. The director also adds an interface to role-figure B. The rule results also a message to create an interface at role-figure A, (ci-tar A src director j B). This rule is a conditional rule. It is only executed if the conditions on its right hand side are met. There are two conditions: that B does not exist (not in(B,N)), and that the plug in request is sent to the director (D == director). If either of these conditions is not met this rule will not be executed.

The second rule is similar to the first one except for passing the role-figure B capability and behaviour specification (pi-tar D src A name B location

LOC1 with BHT2 CAPS2). This rule is a conditional rule, and has one condition (D == director). In this rule the queue and method definitions of role-figure B will be empty after the execution of this rule.

It is important also to notice that in a rewriting rule, the attributes of a class that don't appear on the right hand side of the rule assumed not changed from their values at the left hand side.

Role-figure Plug out

A role-figure may initiate a request to plug out another role-figure. This means terminating the role-figure instance, and removing all interfaces to the terminated role-figure. Similar to the Plug in case, the director is responsible for this request. To handle role-figure plug out we define several rewriting rules:

role-figure Plug out rewriting rules

```
*** in this rewriting rule the director is sending the plugout request
crl [PlugOuttoDirectorSemantics] :
  (po-tar director src A name B)
  < director : Director | Int : N > =>
  < director : Director | Int : N > (po-tar B src director name B) if in(B,N) .
```

```
*** A simple PlugOut method request by which the director is performing the plug-
        out process
crl [PlugOuttoActorSemantics] :
  (po-tar A src B name C)
  < A : RoleFigure | Location : LOC1, Int : N, Beh : BHT1, Cap : CAPS1, Que :
        MESS1, Met : METS1 > =>
  (multimsg remInt from A to N) if (C == A) and (B == director) and in(B,N) .
```

```
*** these two rules will handle the removal of the existing interfaces
crl [RemoveInterfacesAtActorSemantics] :
  (msg remInt from B to A)
  < A : RoleFigure | Int : N > =>
  < A : RoleFigure | Int : N - B >  if in(B,N) .
crl [RemoveInterfacesAtDirectorSemantics] :
  (msg remInt from B to A)
  < A : Director | Int : N > =>
  < A : Director | Int : N - B >  if in(B,N) .
```

The first rewriting rule, [PlugOuttoDirectorSemantics], specifies the handling of the plug out request, (po-tar director src A name B), at the director. The director sends another plug out request to the role-figure B. The second rewriting rule, [PlugOuttoActorSemantics], specifies the handling of the plug out request at the role-figure that is requested to plug out, (po-tar A src

B name C). If this rule is executed it results in destroying the role-figure and generating multiple messages to remove interfaces to this role-figure.

The last two rules, [RemoveInterfacesAtActorSemantics] and [RemoveInterface-sAtDirectorSemantics], handle the removal of interfaces to role-figures that exist in the interface list. There are two rules for this task because we have two classes, one for the director and one for the role-figure. These semantics, as well as the operations on sets defined earlier, achieve the semantics of the remove operator $\triangleright$ on interfaces, which has been defined in Chapter 4.

Create Interface

Requesting a create interface method instructs role-figures to create interface(s) defined in the argument of that request. There are two rules, [CreateInterfacesAtActorSemantics] and [CreateInterfacesAtDirectorSemantics], to handle the creation of interfaces to role-figures, one for the director and one for the role-figure.

role-figure Create interface rewriting rule
*** Create Interface: **crl** [CreateInterfaceAtActorSemantics] : (ci-tar A src C j B) < A : RoleFigure | Location : LOC1, Int : N, Beh : BHT1, Cap : CAPS1, Que : MESS1, Met : METS1 > => < A : RoleFigure | Int : B N > if in(C,N) or (B == C) . **crl** [CreateInterfaceAtDirectorSemantics] : (ci-tar A src C j B) < A : Director | Int : N > => < A : Director | Int : B N > if in(C,N) or (B == C) .

In these rules a role-figure A updates its interface list with an interface to role-figure B in two cases: if the sender of the request exists in the interface list, or if the sender is requesting to create an interface to itself. These semantics, as well as the operations on sets defined earlier, achieve the semantics of the insert operator $\triangleleft$ on interfaces, which has been defined in Chapter 4.

Capability Change

The capability definition of a role-figure may be changed by a capability change procedure, in the following rule [CapabilityChangeSemantics]:

role-figure Capability Change rewriting rule
*** Capability Change:

```
crl [CapabilityChangeSemantics] :
  (cc-tar A src B p CAPS2)
  < A : RoleFigure | Location : LOC1, Int : N, Beh : BHT1, Cap : CAPS1, Que :
        MESS1, Met : METS1 > =>
  < A : RoleFigure | Location : LOC1, Cap : CAPS2 > if in(B,N) .
```

In this rewriting rule a role-figure A changes its capability definition from CAPS1 to CAPS2, (cc-tar A src B p CAPS2). The rule is only executed if the sender B exists in the interface list.

Behaviour Change

A behaviour change implies that a role-figure will be assigned a behaviour different than what it is executing, as in the following rule [Behaviour-ChangeSemantics]:

role-figure Behaviour Change rewriting rule

```
*** Behaviour Change:
crl [BehaviourChangeSemantics] :
  (bc-tar A src B beh BHT2)
  < A : RoleFigure | Location : LOC1, Int : N, Beh : BHT1, Cap : CAPS1, Que :
        MESS1, Met : METS1 > =>
  < A : RoleFigure | Location : LOC1, Beh : BHT2 > if in(B,N) .
```

In this rewriting rule a role-figure A changes its behaviour definition from BHT1 to BHT2 by consuming the request (bc-tar A src B beh BHT2). The rule is only executed if the sender B exists in the interface list. An extended behaviour change request to handle the playability properties of the role-figure model including the arguments current state, stable states, and progress states is specified in Appendix IV.

Role-figure Move

A role-figure move is handled by a move method. In this method the following sequence of tasks is performed: Plug in a new instance of the role-figure at a new location, updating its capability definition as required by the original role-figure, creating interfaces to the list of connected role-figures, changing its behaviour so both instances perform the same role, and plug out the original role-figure. The following rules specify the move method.

role-figure Move rewriting rules

```
*** role-figure Move:
crl [RoleFigureMoveSemantics] :
  (mo-tar A src B location A' LOC2 with M)
  < A : RoleFigure | Location : LOC1, Int : N, Beh : BHT1, Cap : CAPS1, Que :
          MESS1, Met : METS1 > =>
    < A : RoleFigure | Location : LOC1, Int : N > (pi-tar director src A name A' loca-
          tion LOC2 with M) if in(B,N) .
```

```
  ***  plugin request for moving role-figures
  crl [PlugInMovingSemantics] :
    (pi-tar D src A name A' location LOC1 with M)
    < director : Director | Int : N > =>
    < director : Director | Int : A' N >
    < A' : RoleFigure | Location : LOC1, Int : director A, Beh : nilbht, Cap : nilcaps,
          Que : nilmess, Met : nilmets >
    (mor-tar A src director to A' LOC1)
    if not in(A',N) and (M == moving)  and (D == director) .
```

```
  *** the following rule handles the return of this method
  crl [RoleFigureMoveReturnSemantics] :
    (mor-tar A src D to A' LOC2)
    < A : RoleFigure | Location : LOC1, Int : N, Beh : BHT1, Cap : CAPS1, Que :
          MESS1, Met : METS1 > =>
    < A : RoleFigure | Location : LOC1, Int : A' N >
    (cc-tar A' src A p CAPS1) (multimsg creIntS from A' to N) (bc-tar A' src A beh
          BHT1) (po-tar director src A' name A) if (D == director) .
```

In the first rule, [RoleFigureMoveSemantics], a role-figure move request is consumed and a plug in request is initiated at the new location, role-figure *A'* at LOC2.

In the second rule, [PlugInMovingSemantics], we handle the plug in of the role-figure at the new location. This rule is needed to provide the triggering of the rest of the move procedure by sending a return, (mor-tar A src director to A' LOC1). This rule ensures that the role-figure does not exist at that location using the following condition: not in(A',N) and (M == moving).

In the last rule, [RoleFigureMoveReturnSemantics], the move method return is handled. In this rule the sequence of capability, interface, behaviour, and plug out procedures is executed. These procedures are handled by the rules specified earlier.

5.2.4 Service Specification

To experiment with the concepts of role-figure mobility we present an example of a client/server service that uses role-figures to provide functionality (the complete Maude specification of this example is included in Appendix IV, in the *ClientServer* module). Figure 5-1 illustrates this client/server example. A client role-figure ***a*** running in *Node1* uses the service provided by the server role-figure ***b*** that runs in *Node2*. *Node1* and

Node2 are in domain *domain1*. Role-figure ***a*** will move from *Node1* to *Node2* and become role-figure ***a'***. This movement is represented by an arrow in the figure, while interfaces between role-figures are represented by connecters.

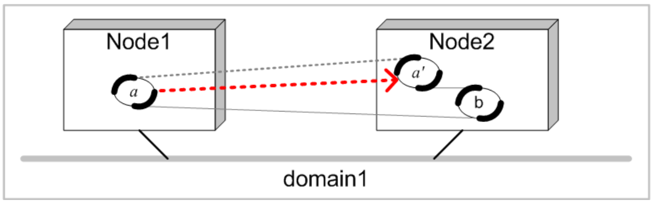


Figure 5-1 An example of a client server configuration

The sequence diagram of Figure 5-2 shows an example execution of the system. In this figure the director and the methods handled by the director have been dropped for clarity. The messages of Figure 5-2 will be explained later in the Maude specification.

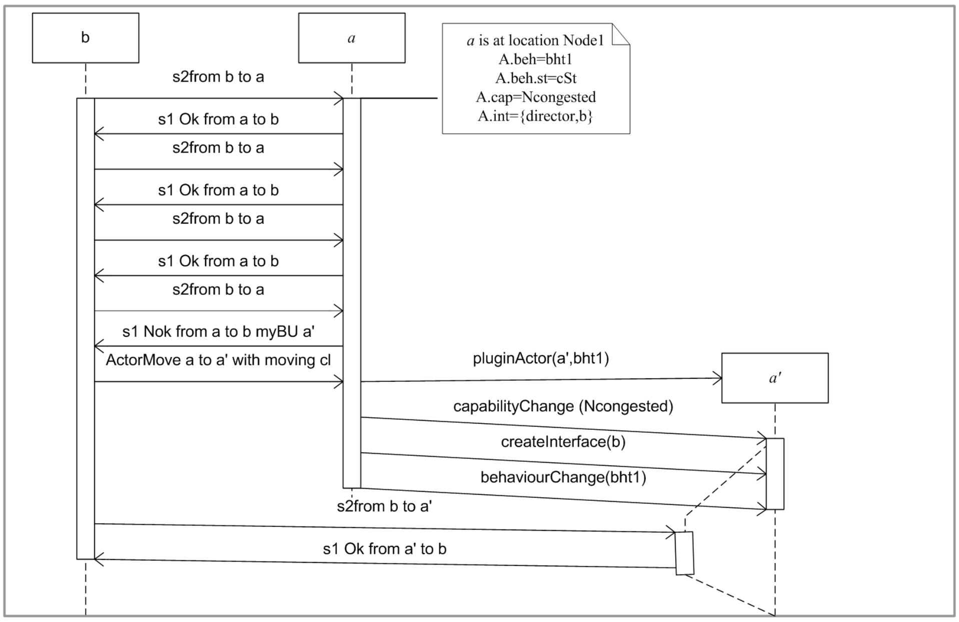


Figure 5-2 An example message diagram of the client/server configuration

The client is considered receiving a data stream from the server with changing transmission bandwidth. At the same time the client relies on a communication channel that could suffer congested conditions. Both the transmission bandwidth and the communication channel are modelled using sorts, constants, and variables in the Maude specification. The server and the client start communicating using initial transmission bandwidth.

The server attempts to increase its transmission bandwidth to the client. Once a congested channel is detected, the client informs the server about it, and supplies a backup location. The server moves the client role-figure to this backup location, and resumes transmitting to it starting from the initial transmission bandwidth.

The Maude module specification of this example uses a subset of the role-figure module presented in the previous subsection. This subset is a simplified set of semantics and rewriting rules. In the following we illustrate the main parts of this module.

In this module we define a class Actor instead of class role-figure. The client role-figure ***a*** is declared of class *ActorClient*, while the server role-figure ***b*** is declared of class *ActorServer*. *ActorClient* and *ActorServer* are inherited from the Actor class using the subclass relation.

Client/Server role-figure definitions

```
***Application actors inheriting from the generic actor class
class  Actor | Int : OidSet, Beh : BehType, Cap : CapSet,
          Que : MesSet, Met : MetSet .
class  ActorClient | Count : MachineInt, Backup : Oid .
class  ActorServer | Trans : MachineInt .
subclass ActorServer < Actor .
subclass ActorClient < Actor .
```

In ActorClient and ActorServer, *Count* and *Trans* keep track of the utilized channel. These values will increase in case the server requires more bandwidth. *Backup* defines the backup location for the client.

The module uses the following capability definitions: *Congested* and *Ncongested* to determine when an *ActorClient* role-figure suffers a congested condition. The module defines the Status of the channel by *Ok* and *Nok*. The sorts of capability and status are *Cong* and *Status*, respectively.

Client/Server capability and status definitions

```
***capability definition: Congestion
sorts Status Client Server Cong congectionValue .
subsort Cong < Cap < CapSet .
*** capability instances, and congestion condition
op Congested : -> Cong .
op Ncongested : -> Cong .
op congestionValue : -> MachineInt .
*** the status of the channel
op Ok : -> Status .
op Nok : -> Status .
```

We also need specific signals that will be used to communicate between the client and the server.

Client/Server signal definitions

```
*** signal definitions
msg s1_from_to_ : Status Oid Oid -> Msg .
msg s1_from_to_myBU_ : Status Oid Oid Oid -> Msg .
msg s2from_to_ : Oid Oid -> Msg .
```

s1 is used to acknowledge the reception of a signal at the client. It may include either an *Ok* or a *Nok* status, in the latter a *myBU* backup location is specified. s2 is sent by the server to the client.

The role-figure configuration, in Figure 5-2, may be represented in Maude with a triggering event, a message (s2from b to a), as in the following definition.

Client/Server example configuration definition

```
eq testCS2 =
  (s2from b to a)
  < a : ActorClient | Int : director b , Beh : bht1, Cap : Ncongested, Que : mess1,
          Met : mets1, Count : 0, Backup : a' >
  < b : ActorServer | Int : director a , Beh : bht2, Cap : caps2, Que : mess2, Met :
          mets2, Trans : transmissionChannel > .
```

A client normally has a counter *Count* to check the utilization of its channel availability, while a server measures its transmission channel by *Trans*, which both are integers for simplicity. Note that the client has an *Ncongested* capability upon start up. A client may adapt to a shortage of its channel by providing a backup instance ***a'*** that might be plugged in at some predefined location, again for simplicity in Node2, and resume receiving from the server.

The following rewriting rule defines a normal operation by the client upon receiving from the server, while the other rule is used for congestion situations:

ActorClient rewriting rules

```
*** a non-congested client receiving a signal from the server
crl [receiveNotcongested] :
  (s2from B to A)
  < A : ActorClient | Int : N, Beh : BHT1, Cap : CONG, Que : MESS1, Met :
          METS1, Count : COUNT, Backup : A' > =>
  < A : ActorClient | Cap : Ncongested, Count : COUNT + 1 > (s1 Ok from A to B)
  if (COUNT <= congestionValue) .
```

```
 *** a congested client receiving a signal from the server
crl [receiveCongested] :
  (s2from B to A)
  < A : ActorClient | Int : N, Beh : BHT1, Cap : CONG, Que : MESS1, Met :
         METS1, Count : COUNT, Backup : A' > =>
  < A : ActorClient | Cap : Congested, Count : 0 >
  (s1 Nok from A to B myBU A')
  if (COUNT > congestionValue) .
```

The first rule specifies how a role-figure would evolve in normal non-congested situations, while the second rule signals to the server that it is experiencing a congested situation.

The following two rewriting rules define the operation of the server and the operation of the client for normal and congestion situations:

ActorServer rewriting rules

```
*** if the sending is to continue normally
crl [adapt2] :
  (s1 stus from B to A)
  < A : ActorServer | Int : N, Beh : BHT1, Cap : CAPS1, Que : MESS1, Met :
         METS1, Trans : TRANS >
  => < A : ActorServer | Trans : TRANS - 1 > (s2from A to B)
  if (stus == Ok) and (TRANS =/= 0) .
```

```
*** if the client suffers a congestion condition, two cases;
*** First case: if there is a  Backup for the congested actor
crl [adapt3] :
  (s1 stus from B to A myBU B')
  < A : ActorServer | Int : N, Beh : BHT1, Cap : CAPS1, Que : MESS1, Met :
         METS1, Trans : TRANS >
  =>  < A : ActorServer | Trans : TRANS - 1 >
  (ActorMove B to B' with moving cl) (s2from A to B')
  if (stus =/= Ok) and (TRANS =/= 0) .

*** Second case: if there is no Backup for the congested actor
 crl [adapt3nobackup] :
  (s1 stus from B to A)
  < A : ActorServer | Int : N, Beh : BHT1, Cap : CAPS1, Que : MESS1, Met :
         METS1, Trans : TRANS >
  => < A : ActorServer | Trans : TRANS - 1 > (ActorPlugOut B to director from A)
         (ActorPlugIn B from A )
  if (stus == Nok) and (TRANS =/= 0) .
```

These rules specify the server actions to the client signals. A client signal *s2* comes with a status *stus*. This is examined, and in case of congestion, a move request is initiated to the backup location *myBU*. A simple scenario to test the move procedure is to steadily increase the sending bandwidth from the server until the congestion situation is reached. The server will detect this situation by receiving a status *Nok*, which shows degradation in some mandatory QoS requirements have happened. We assume the simplest scenario, both the server and the client can still communicate, and the client could move to a backup location.

5.3 Formal analysis and validation

After presenting the formal specification of the role-figure model in Sec. 5.2, we show in this section the results of the formal analysis and validation. The formal analysis and validation is presented in three parts. In Sec. 5.3.1 we present the model execution, which is a direct simulation, debugging, and trace analysis process. The model execution assesses obtaining better versions of the specification. In Sec. 5.3.2 we present the exhaustive execution of the role-figure model. This part examine all the possible executions of a given role-figure configuration. In Sec. 5.3.3 we present our brief validation efforts for the role-figure mobility management. We conclude the formal analysis and validation with some experiences in Sec. 5.3.4.

5.3.1 Model executions

A Maude formal specification has a great advantage. It can be validated immediately by executing configurations. This execution provides quick feedbacks on the specification. We used this feature during the specification of the role-figure mobility management in particular. In this subsection we define a number of configurations of role-figures and messages covering different scenarios of service adaptability and mobility management. We use the default Maude interpreter (using the command **rew** or **rewrite**) to *rewrite* these test cases according to the rewriting rules of the role-figure model.

Each execution of these configurations simulates an arbitrary run of the specification, and results a configuration that cannot be applied to any of the rewriting rules – hence the default Maude interpreter stops executing and returns a result configuration.

Another important facility of the Maude tool is used to discover and detect errors. This built-in facility is the tracing of the executions. We use two types of configurations. Configurations of generic actors are used to check the computing architecture functionality, while configurations of client and server role-figures are used to check the application or service functionality.

Some computing architecture configurations

In the following we present several configurations and clarify how the Maude system interprets them. We show here several configuration examples for the computing architecture: testS1 (tests the plug out rewriting rules), testS13 (tests the plug in and the plug out rewriting rules) and testS6 (tests the role-figure move rewriting rules). To check other configurations see Appendix IV. In this appendix each Maude module contains several configurations used for testing purposes. Each one of these configurations tests a set of the rewriting rules in the given module. By executing one of these configurations the corresponding set of rewriting rules can be tested.

role-figure Configuration examples *test1* and *test6*

```
eq testS1 =
  (po-tar e src director name e)
  < director : Director | Int : a b c e >
  < a : RoleFigure | Location : loc1, Int : director b c e, Beh : bht1, Cap : caps1,
          Que : mess1, Met : mets1 >
  < b : RoleFigure | Location : loc1, Int : director a c e, Beh : bht2, Cap : caps2,
          Que : mess2, Met : mets2 >
  < c : RoleFigure | Location : loc1, Int : director a b e, Beh : bht1, Cap : caps1,
          Que : mess1, Met : mets1 >
  < e : RoleFigure | Location : loc1, Int : director a b c, Beh : bht2, Cap : caps2,
          Que : mess2, Met : mets2 > .
```

```
eq testS13 =
  (po-tar e src director name e)(pi-tar director src a name e location loc1 with m)
  < director : Director | Int : a b c e >
  < a : RoleFigure | Location : loc1, Int : director b c e, Beh : bht1, Cap : caps1,
          Que : mess1, Met : mets1 >
  < b : RoleFigure | Location : loc1, Int : director a c e, Beh : bht2, Cap : caps2,
          Que : mess2, Met : mets2 >
  < c : RoleFigure | Location : loc1, Int : director a b e, Beh : bht1, Cap : caps1,
          Que : mess1, Met : mets1 >
  < e : RoleFigure | Location : loc1, Int : director a b c, Beh : bht2, Cap : caps2,
          Que : mess2, Met : mets2 > .
```

```
eq testS6 =
  (mo-tar a src b location a' loc2 with moving)
  < director : Director | Int : a b c >
  < a : RoleFigure | Location : loc1,  Int : director b c, Beh : bht1, Cap : caps1, Que
          : mess1, Met : mets1 >
  < b : RoleFigure | Location : loc1,  Int : director a c, Beh : bht2, Cap : caps2, Que
```

```
            : mess2, Met : mets2 >
    < c : RoleFigure | Location : loc1,  Int : director b a, Beh : bht1, Cap : caps1, Que
            : mess1, Met : mets1 > .
```

Executing the configuration testS1 as an arbitrary execution of the Maude interpreter gives the following result – this result is obtained as no rewriting rules could be applied any more:

```
Maude> (rew testS1 .)
rewrites: 451 in 13ms cpu (14ms real) (34411 rewrites/second)
rewrite in RoleFigureModel : testS1 .
result Configuration :
< director : Director | Int : a b c >
< a : RoleFigure | Que : mess1 , Int : director b c , Met : mets1 , Cap : caps1, Beh :
        bht1 , Location : loc1 >
< b : RoleFigure | Que : mess2 , Int : director a c , Met : mets2 , Cap : caps2 , Beh :
        bht2 , Location: loc1 >
< c : RoleFigure | Que : mess1 , Int : director a b , Met : mets1 , Cap : caps1 , Beh :
        bht1 , Location : loc1 >
```

The first several lines show how we execute the configuration, how many rewriting rules are executed, and how long it takes to rewrite the testS1 configuration. The result configuration is highlighted. This result configuration shows that all messages that have been produced throughout the execution have been consumed. This means testS1 had a director, four role-figures, and a message before the execution, while after the execution the result configuration has only a director and three role-figures. It also shows that role-figure *e* successfully plugged out, and interfaces to it have been removed from the director and the other role-figures.

Executing the testS13 configuration as an arbitrary execution of the Maude interpreter gives the following result:

```
Maude> (rew testS13 .)
rewrites: 565 in 1ms cpu (3ms real) (509927 rewrites/second)
rewrite in RoleFigureModel : testS13 .
result Configuration :
< e : RoleFigure | Que : nilmess , Int : director a , Met : nilmets , Cap : nilcaps , Beh :
        nilbht , Location : loc1 >
< director : Director | Int : e a b c >
< a : RoleFigure | Que : mess1 , Int : e director b c , Met : mets1 , Cap : caps1 , Beh :
        bht1 ,Location : loc1 >
< b : RoleFigure | Que : mess2 , Int : director a c , Met : mets2 , Cap : caps2 , Beh :
        bht2 , Location : loc1 >
```

```
< c :RoleFigure | Que : mess1 , Int : director a b , Met : mets1 , Cap : caps1 , Beh :
        bht1 , Location : loc1 >
```

The result shows that role-figure *e* successfully plugs out, then plugs in. Note interfaces between *a* and *e* exist in both role-figures, i.e. *a* has an interface to *e* and *e* has an interface to *a*.

testS6 gives the following result when executed:

```
Maude> (rew testS6 .)
rewrites: 705 in 2ms cpu (2ms real) (334440 rewrites/second)
rewrite in RoleFigureModel : testS6 .
result Configuration :
< director : Director | Int : a' b c >
< a' : RoleFigure | Que : nilmess , Int : director b c , Met : nilmets , Cap :caps1 , Beh :
        bht1 , Location : loc2 >
< b : RoleFigure | Que : mess2 , Int : director a' c , Met : mets2 , Cap : caps2 , Beh :
        bht2 ,Location : loc1 >
< c : RoleFigure | Que : mess1 , Int : director a' b , Met : mets1 , Cap : caps1 , Beh :
        bht1 , Location : loc1 >
```

This shows a successful role-figure move, in which the role-figure a' is successfully plugged in and its definition parts have been updated. Note the other role-figures have updated their interface definitions to a'.

Client/Server configurations

Regarding the client/server module, example configurations testCS1 and testCS2 are presented in the following:

Client/Server role-figure Configuration example *testCS1*
eq testCS1 = (s2from b to a) < director : Director | Int : a b > < a : ActorClient | Int : director b , Beh : bht1, Cap : Ncongested, Que : mess1, Met : mets1, Count : 0, Backup : a' > < b : ActorServer | Int : director a , Beh : bht2, Cap : caps2, Que : mess2, Met : mets2, Trans : 1 > .
eq testCS2 = (s2from b to a) < director : Director | Int : a b > < a : ActorClient | Int : director b , Beh : bht1, Cap : Ncongested, Que : mess1, Met : mets1, Count : 0, Backup : a' > < b : ActorServer | Int : director a , Beh : bht2, Cap : caps2, Que : mess2, Met : mets2, Trans : 5 > .

The following are the results of the execution of these two configurations.

```
Maude> (rew testCS1 .)
rewrites: 176 in 8ms cpu (9ms real) (21709 rewrites/second)
rewrite in ClientServer : testCS1 .
result Configuration :
< director : Director | Int : a b >
< a : ActorClient |Que : mess1 , Int : director b , Met : mets1 , Cap : Ncongested , Beh :
        bht1 , Backup : a' , Count : 0 >
< b : ActorServer | Que : mess2 , Int : director a , Met : mets2 , Cap : caps2 , Beh :
        bht2 , Trans : 0 >
```

In the execution of testCS1, the Trans is set to 1, so that the channel utilization between the server and the actor is not increased. This case does not generate a congested situation, and therefore no move request is issued.

```
Maude> (rew testCS2 .)
rewrites: 438 in 0ms cpu (2ms real) (4055555 rewrites/second)
rewrite in ClientServer : testCS2 .
result Configuration :
< director : Director | Int : a' b >
 < a' : ActorClient | Que : nilmess , Int : director b , Met : nilmets , Cap : Ncongested ,
        Beh : bht1 , Backup : nilOid , Count : 0 >
< b : ActorServer | Que : mess2 , Int : director a' , Met : mets2 , Cap : caps2 , Beh :
        bht2 , Trans : 0 >
```

The execution of *testCS2* yields that the client side suffers a shortage in the receiving capability (as its channel is congested), so the server initiates a role-figure move to the backup location specified by the client.

The configurations we presented in this subsection have been experimented in different variations and increments. For instance various numbers of role-figures have been used. Also various numbers of messages, as well as faulty messages have been used. In certain cases non-existing destinations, duplicate role-figure names, and other types of faults have been experimented. In the Maude specification in Appendix IV, only the simplest configurations are included. These configurations can be used to test the different sets of rewriting rules if these rules are changed. Also, these configurations can be easily extended, e.g. by adding more role-figures, by adding more messages, by changing the existing role-figures, or by changing the existing messages.

5.3.2 Exhaustive executions

The executions of the specification of the role-figure model revealed some errors and inconsistencies. However, these executions showed only a single and particular execution of the model with regard to a given configuration.

We need to test other executions of the model, possibly every execution, with regard to these configurations. This is possible by applying state space exploration techniques. This means using techniques to examine all possible executions of the model, until terminal states are reached. These possible executions can be generated up to a certain depth level of the state space.

We perform this analysis to make sure that the results we obtained by executing the role-figure configurations in Sec. 5.3.1 are the only possible results. We achieve this by means of exhaustive executions that we perform manually. This means to enforce the Maude interpreter to examine all the possible rewriting rules after every transition. For instance, we execute only a certain set of the rewriting rules in one execution and discard the others, e.g. only checking the rules for the plug in. Additionally we enforce the execution of our model to be always handled by checking the rewriting rules in an ordered manner, e.g. check the plug in rules and then check the plug out rules.

The result of an exhaustive execution is one or more role-figure configurations, similar to the configurations we obtained in our model executions. The different role-figure configurations mean that there are different possible executions of the model. The resulting configurations and their execution traces can be examined to figure out how these configurations are obtained.

We show here an example role-figure configuration that can give two different result configurations under our exhaustive execution. The exhaustive execution of testS13 presented earlier yields the following result configurations:

```
result Configuration :
< e : RoleFigure | Que : nilmess , Int : director a , Met : nilmets , Cap : nilcaps , Beh :
        nilbht , Location : loc1 >
< director : Director | Int : e a b c >
< a : RoleFigure | Que : mess1 , Int : e director b c , Met : mets1 , Cap : caps1 , Beh :
        bht1 ,Location : loc1 >
```

```
< b : RoleFigure | Que : mess2 , Int : director a c , Met : mets2 , Cap : caps2 , Beh :
         bht2 , Location : loc1 >
< c :RoleFigure | Que : mess1 , Int : director a b , Met : mets1 , Cap : caps1 , Beh :
         bht1 , Location : loc1 >

< e : RoleFigure | Que : nilmess , Int : director a , Met : nilmets , Cap : nilcaps , Beh :
         nilbht , Location : loc1 >
< director : Director | Int : e a b c >
< a : RoleFigure | Que : mess1 , Int : director b c , Met : mets1 , Cap : caps1 , Beh :
         bht1 ,Location : loc1 >
< b : RoleFigure | Que : mess2 , Int : director a c , Met : mets2 , Cap : caps2 , Beh :
         bht2 , Location : loc1 >
< c :RoleFigure | Que : mess1 , Int : director a b , Met : mets1 , Cap : caps1 , Beh :
         bht1 , Location : loc1 >
```

These resulting configurations are only different in the interface definition of the role-figure a. These configurations are obtained by manually controlling the order of the execution of the role-figure model rewriting rules, in particular the plug in and the plug out rules.

The analysis of the trace of this exhaustive execution shows that the plug in and the plug out requests were competing against each other. The first configuration is similar to the result configuration we obtained in Sec. 5.3.1. In the second configuration, the plug out request initiated the removal of the interfaces to the plugged out role-figure, i.e. the interface to e was removed from a. However this removal was executed after the plug in request, sent by a, was consumed.

The reason for such a situation is that role-figure a should not have sent the plug in request in the first place. Role-figure a, as it is defined in configuration test13, has already an interface to role-figure e, i.e. it cannot initiate a plug in request for an already existing role-figure. Exhaustive executions for other configurations have been conducted. Some of these exhaustive executions showed situations of inconsistent semantics similar to the one shown here for test13. The implications of these executions have already been incorporated in the rewriting rules for the role-figure model specification.

5.3.3 Role-figure mobility validation

Another approach to validate the role-figure model is by using the reflective kernel META-LEVEL provided in the Maude language to control the rewriting strategy starting from initial role-figure configurations. The reflection in the rewriting logic theory is briefly explained in Appendix II, furthermore [CDE03] and [Mau04] may be used for a detailed description on the kernel META-LEVEL and the Maude module META-LEVEL. We use this approach to validate role-figure mobility.

The approach is simple and consists of two phases. First, all executions are checked and all possible rewriting paths are examined. This gives all possible result configurations, which could be saved for further inspection. This can be regarded as an automated exhaustive execution of the initial role-figure configurations. Second, the resulting configurations of the first phase are checked against some correctness criteria that validates the mobility feature. If these resulting configurations obey the given requirements for the role-figure mobility then our proposed mobility management solution is said to:

“*give valid and correct configurations*” and
“*handle role-figure mobility according to the used requirements*”

In this regard we exploit the defined properties of the role-figure model in Sec. 4.6: $\boldsymbol{P}_{\text{pluggability}}$, $\boldsymbol{P}_{\text{consumeability}}$, $\boldsymbol{P}_{\text{wplayability}}$, and $\boldsymbol{P}_{\text{splayability}}$. In a general role-figure mobility management, the following items are checked:

(i) A moved role-figure has been successfully plugged in at the new location, with correct and valid interface definitions. This is a direct validation of the property $\boldsymbol{P}_{\text{pluggability}}$.

(ii) The resulting configuration should only contain role-figures and not messages, otherwise unexpected events may have happened and one needs to study the trace generated at this meta-level computations. This is the property $\boldsymbol{P}_{\text{consumeability}}$.

(iii) One, and only one, instance of the moved actor does exist in the resulting configuration – assuming no role-figure replication is applied.

(iv) Interface definition, behaviour definition, and capability definition are handled according to the mobility strategy (assuming that queue and methods are not handled). This is achieved partly through validating the property $\boldsymbol{P}_{\text{wplayability}}$ and $\boldsymbol{P}_{\text{splayability}}$. This means to prove that a moved role-figure has started executing (playing) its role after the move. But also we need further information on the role-figure state-information before the move, as well as the role-figure state-information after the move.

In the first phase, an internal Maude strategy for checking the reducibility of a given term in Maude has been applied. Internal Maude strategies are used to specify at the META-LEVEL how to control the rewriting inference process (more on internal strategies see [CEL96]). This internal Maude strategy was adopted from the strategy used in the formal analysis of the Active Networks in Maude, as presented in [DMT00] and [WMG00]. The internal Maude strategy executes at the meta-level of the language, and results the set of all possible configurations. In the second phase, the resulting configurations were verified against our role-figure mobility assumptions, (i) – (iv). This verification was mainly performed manually. However, early results of using a number of predicates to check the correctness of these assumptions have been achieved, namely assumptions (i) and (ii). This approach validates all non-rewritable states reachable from the initial state, possibly multiple configurations.

As a demonstration of the applied validation approach, we briefly present the validation of the configuration testS6 that contains a simple role-figure mobility management scenario. This configuration, as the mobility example we used in Chapter 4, assumes the application of a mobility method that does not involve mobility manager, e.g. mobility method *RMM3*. Also it assumes that the applied mobility strategy does not have rules regarding the requirements on queue content and executing methods, i.e. applying the design rules *RD9* (*a*) and *RD9* (*q*) from Sec. 3.4.4.2.

```
Maude> (down RoleFigureModel : red terminalConfigurations(RoleFigureModel,
        {'testS6}'Configuration, allAMrules) .)
rewrites: 39790 in 74ms cpu (75ms real) (536998 rewrites/second)
result Configuration :
< director : Director | Int : a' b c >
< a' : Actor | Que : nilmess , Int : director b c , Met : nilmets , Cap : caps1 , Beh : bht1 >
< b : Actor | Que : mess2 , Int : director a' c , Met : mets2 , Cap: caps2 , Beh : bht2 >
< c : Actor | Que : mess1 , Int : director a' b , Met : mets1 , Cap : caps1 , Beh : bht1 >
```

The first line instructs the Maude interpreter to *reduce* (using the command red) the term: terminalConfigurations (RoleFigureModel, {'test6}'Configuration, allAMrules) which calls the operation terminalConfigurations that searches for all possible rewriting paths concerning the configuration testS6. The Maude interpreter matches testS6 against all the rewriting rules of the role-figure model, which are listed in the term allAMrules. When applying the correctness predicates (i) – (iv) defined earlier, the outcome of this execution is the same as the result of the arbitrary execution conducted earlier. Note the dramatic increase in the number of rewrites: 39790 rewrites, compared to 705 in the arbitrary execution in Sec. 5.3.1.

To experiment with this configuration several validation runs have been applied with various sets of rewriting rules, various depth levels, etc. Similar validation runs have been applied to other configurations with different scenarios regarding role-figure mobility.

5.3.4 Experiences

During the formal analysis of the role-figure specification many of the small details governing both the computing architecture and the service functionality have been refined based on the obtained results. Our formalization work has revealed several details regarding the validity of the constructed model. There were several occasions where changes to the role-figure mobility management procedures were needed. After applying these changes, acceptable results have been achieved. Below we list some of these details:

(i) The director, responsible for the play view of the architecture, needs to explicitly monitor the create interface method during the plug in of new role-figures. In the plug in rewriting rules, the director monitors the create interface request to the role-figure, which has requested a plug in of another role-figure. If this role-figure is down, the director plugs out the newly created role-figure.

(ii) Regarding role-figures classes inherited from a base class, new plug in rewriting rules were specified to create these role-figures.

(iii) In a plug out method request, the handling of the removal of the interfaces has been tightly coupled with the handling of the method itself. The director takes a decisive role here, and controls all these removals. The unconnected interfaces turned out to be a real challenge.

(iv) In the move request handling there was a need to propose a solution for backup instances that cannot be plugged in, or those without valid backup.

(v) In the handling of the move request, the plug in of a moving role-figure needed to be handled in a different way than a normal role-figure plug in.

(vi) In the role-figure move rewriting rule that includes the method return, a hidden ambiguity in the handling of the series of method requests was clarified. The initial thought was to request these methods in one rewriting rule. The validation run showed certain situations where the order of these method requests was crucial. One solution to this problem can be to apply returns for these methods, e.g. the plug in request.

(vii) During the while-on-the-move phase of the role-figure move procedure, the role-figure and its replica should not have interfaces to each other, however they must manage to communicate to transfer information related to interface, behaviour, and capability definitions. This issue led to the introduction of a role-figure mobility method that uses locators.

(viii) In certain test configurations, some interface relations could not be updated (because certain connected role-figures themselves were moving), while some capability definitions have not been reclaimed (because they were not available in the new location). These examinations led to a modified approach for the role-figure mobility, as we present in the following section.

A clear benefit of applying the formalization work presented in this chapter was the changes we applied to the handling of the role-figure mobility management. According to the original concept, role-figure mobility was aimed at achieving role-session, state-information, data-space, and capability mobility, similarly to how code mobility paradigm is addressed in other network architectures, e.g. those based on mobile agents.

However, experimenting with the Maude specification and the formal analysis showed some alterations of the original concept. The most important alterations are with regard to the handling of the interfaces and dealing with the while-on-the-move conditions of the moving role-figure. Conducting various mobility scenarios led to the following two conclusions:

(i) Role-figure move procedure should achieve interface mobility alongside handling two replicas of the same role-figure in order to avoid invalid interface definitions. In Figure 5-3 we illustrate a part of the concept with regard to moving interfaces. In this role-figure Mobility scenario role-figure *a* moves to a new location *a'*. The figure focuses on the interface definition, and suggests that interfaces of the moving role-figure should be created to the relevant role-figures in the new location, e.g. *a'* connects to *b* instead of *c*.

(ii) Role-figure mobility should handle the while-on-the-move conditions of the moved role-figure by: considering the relationship between the replica, clarifying the mobility strategy in terms of requirements on the queue contents, and the handling of the suspended methods.

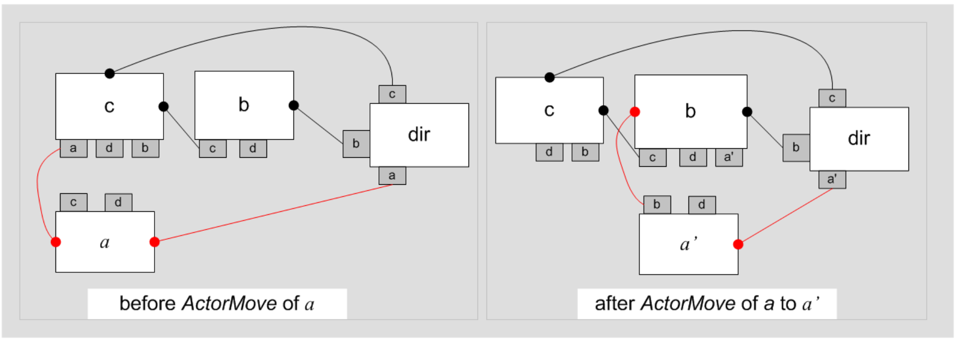


Figure 5-3 Illustration of the revised role-figure mobility

These two situations have not been dealt with in our formalization work. The Maude specification for these two situations can be specified based on our specification and the formal analysis and validation can be similar to our formal analysis and validation efforts. This is a very important topic for further development of our formalization that can be addressed in future work.

CHAPTER **6**

Conclusion

6.1 Results

In this thesis we have considered mobility management in adaptable service systems. Three main mobility types have been considered: *personal*, *role-figure*, and *terminal* mobility. *Personal mobility* is the mobility of the user and its session in domains; *role-figure mobility* is the mobility of instantiated software components, or role-figures; *terminal mobility* is the mobility of the terminals across domains. There are four contributions within the context of the mobility management: the *terminology framework*, the *mobility management architecture*, the *role-figure model*, the *formal model* and *analysis* of the role-figure model. In addition the work with mobility management has contributed to the *TAPAS computing architecture* and *system management architecture*, which was presented in Sec. 1.4.

The *terminology framework* presented in Chapter 2 provided a set of generic concepts and definitions for the three types of mobility. Propositions and requirement rules were also given to show what definitions are required and how they are related to each other. Among the most important ones, *login agent*, *user agent* and *visitor agent* have been defined to handle the personal mobility. *User agent* handles the user interactions at home domain, while *visitor agent* handles the user interactions at visitor domain. *Mobility manager* has also been defined to handle the role-figure and the terminal mobility. Terminal mobility also needed the definition of *mobility agent*.

The *mobility management architecture* was presented in Chapter 3. Concepts, procedures, mechanisms, and design rules to handle the three mobility types presented in the terminology framework were developed.

The mobility management functionality in this chapter has been handled by functional entities that were introduced. *Personal mobility* has been handled by the functional entities: *LoginAgent*, *UserAgent*, *VisitorAgent*, *UserProfileBase*, and *UserSessionBase*, which are role-figures and databases. Several mobility management procedures have been developed for handling the user and user session mobility that constitute the personal mobility. Regarding *role-figure mobility*, *MobilityManager* has been introduced to handle the movement of role-figures. It also handles the mobility strategies, which are sets of domain based rules to control role-figure mobility (i.e. to decide when, how, and where to move). The management functionality for the role-figure mobility was handled by the introduced four different mobility management mechanisms: centralized, proxy, locator, and persistent mechanisms. *Terminal mobility* was handled by two main functional entities: *MobilityManager* and *MobilityAgent*. The proposed solution for *terminal mobility* was only handled with respect to the introduced implications from the other two mobility types, i.e. personal and role-figure mobility. When a user moves with its terminal to a new location or a new domain, the terminal mobility must be handled by the architecture. Also, when a terminal moves the executing role-figures may be required to move to another node or terminal (due to the reduced availability of capabilities in the moving terminal or the limited access rights of a user). Several prototype implementations have been developed to experiment with the different features of this architecture. One prototype implementation was developed for the personal mobility, and another prototype implementation was developed for the terminal and role-figure mobility. The procedures and mechanisms presented in this chapter have been implemented. The development was within the framework of TAPAS (Telematics Architecture for Play-based Adaptable Service Systems).

The role-figure model, developed in Chapter 4, is an abstract model for the implemented role-figure functionality and the role-figure mobility. Concerning this *role-figure model*, we have been investigating several candidate frameworks for our modelling and formalization efforts. Our initial thoughts were to use an existing semantic framework, such as SDL agents, ODP computing objects, or the actor language theories. By using an ODP (Open Distributed Processing) semantic framework and the rewriting logic, the structure of the cooperating role-figures and their behaviour is defined. The model considers the most important parts such as behaviour, interfaces, capabilities, messages, and executing methods. This model also presented three role-figure model properties: *pluggability*, *consummability*, and *playability*.

This role-figure model was used for the formal analysis developed in Chapter 5. This analysis is based on a *Maude formal model*. Maude is a language and tool supporting specification and analysis of rewriting logic theories. Some added assumptions and simplifications were needed to develop and analyze the Maude specification. The formal analysis has been conducted using state space exploration techniques in the Maude tool for the validation of the role-figure mobility management. The formalization and analysis of this role-figure model using the Maude language and tool turned out to be a good choice. Maude was advantageous in developing our specification and conducting the analysis in a relatively fast manner.

The developed mobility management functionality for the three mobility types can be related to similar concepts and related work. With regard to our proposed solution for the *personal mobility*, the main issue was flexibility. The separation between the user interface and the terminal interface was important to address the needs for personal mobility separately from the terminal mobility. This is more flexible than the concepts of user, SIM module, and terminal used in mobile telephony systems. Another important aspect of the proposed solution was the handling of the user capabilities in two main parts: user related and user session related capabilities. Generally, this solution is inline with the handling of personal mobility by VHE and OSA standards. However, our approach to user session as based on role-figures instantiated and controlled by a *user agent* or a *visitor agent* can be more flexible with regard to role-figure mobility. For example, when a user moves, its session also moves. This means its role-figures must be moved as well; however, not every role-figure can move with the user session. Using role-figure mobility it is possible to move these role-figures to an appropriate location, and still achieve user session mobility. In addition to the similarities mentioned above, the concepts of *user agent* and the *visitor agent* can also be compared to the concepts of the home agents and the foreign agents in the Mobile IP.

Also, our handling of the *role-figure mobility* can be compared to code mobility related concepts. Regarding mobile agent systems, a mobile role-figure is similar in principle to a controllable mobile agent with limited autonomous computing capability. The developed concepts for role-figures and role-figure mobility achieve both the *agency* and the *mobility* aspects associated with mobile agents. The *agency* aspect implies that a role-figure as a software component can act on behalf of another component. For example, a *user agent* and *visitor agent* act on behalf of users to offer service personalization to users. Also, a *mobility agent* acts on behalf of a terminal to control its location. The other aspect of mobile agents is the *mobility*. The main objective in moving role-figures is to reduce network traffic, improve capability allocation, increase fault resilience, and allow local access of data. We believe role-figure mobility has the potential to achieve the aspect of *mobility* with better utilization of the system resources due to the limited, but flexible, autonomous characteristics of role-figures. Additionally, we also allow domains and sub-domains to define different mobility strategies to control the role-figure mobility. These strategies are mainly used to control how role-figures are preserved after the move. Mobility strategies also can be used to restrict or prohibit role-figure mobility in a visitor domain.

6.2 Perspectives

The TAPAS *mobility management* architecture and its mobility procedures contain several areas are that left out to be studied and developed in future work. For example the developed role-figure mobility mechanisms can be studied and experimented in large domains of nodes and role-figures. The experimentations can be carried out using varying conditions of network traffic, changing service functionality, and multi mobility managers. Also new role-figure mobility management mechanisms may be developed.

Although our experiences in modelling the role-figure and its properties regarding the role-figure mobility management are quite encouraging, the results of this thesis in this area are just a first step. The role-figure model opens up interesting research areas. The reasoning about the role-figure queue and methods as part of the role-figure mobility management is one example of such research areas. Also the semantics and the dynamics of the role-figure model can be re-constructed based on more elaborated interface type theory than the one we used. The properties of the role-figure model may also be extended to elaborate on the role-figure mobility management mechanisms.

More efforts as well as more case studies are needed to experiment with our Maude formal specification. Such efforts will definitely increase the level of confidence in our results even further. For instance, further validation of the role-figure model can be conducted using the built in Maude model checker. Also, the property specification logic (linear temporal logic) and the decision procedures for it (model checking) can be used to prove other properties of the role-figure model than the three main properties. Besides, extending the role-figure model specification to include an interface type theory, queue semantics handling, and executing methods handling would provide a comprehensive formal analysis for the role-figure model presented in Chapter 4. Conducting a case study that includes the specification and validation of a large service system using our Maude specification is also an interesting future work.

Concerning the *TAPAS architecture*, it has been justified that the TAPAS computing architecture provides solid ground for further studies on adaptable service systems. The *system management* architecture is still under development through PhD work. As the work with these different parts is the subject for academic work, there is a potential for a better integration and harmonization. Seen from the mobility management, the integration of the role of the mobility manager into the service management, configuration management, and capability management architecture components is an interesting topic for further study and development.

REFERENCES

[AAS02] F. A. Aagesen, C. Anutariya, M. M. Shiaa, and B. E. Helvik, “Support Specification and Selection in TAPAS”, *Proc. IFIP WG6.7 Workshop on Adaptable Networks and Teleservices*, Trondheim, Norway, September 2002.

[Agh90] G. Agha, “Concurrent object-oriented programming”, *Communications of the ACM*, 33(9): 125-141, September 1990.

[AH88] G. Agha and C. Hewitt, “Concurrent Programming using Actors”, In A. Yonezawa and M. Tokoro, editors, Object-Oriented Concurrent Programming, pages 37-54, *The MIT press*, 1988.

[AHA03] F. A. Aagesen, B. E. Helvik, C. Anutariya, and M. M. Shiaa, “On Adaptable Networking”, *The 2003 International Conference on Information and Communication Technologies (ICT 2003)*, April 2003.

[AHJ01] F. A. Aagesen, B.E. Helvik, U. Johansen, and H. Meling, “Plug and Play for Telecommunication Functionality: Architecture and Demonstration Issues”, *Proc. Int. Conf. Information Technology for the New Millennium (IConIT)*, Thammasat University, Bangkok, Thailand, May 2001.

[AHW99] F. A. Aagesen, B. E. Helvik, V. Wuwongse, H. Meling, R. Braek and U. Johansen, “Towards A Plug and Play Architecture for Telecommunications”, *Proceedings of IFIP SMARTNET'99*, Bangkok, November 1999.

[ASA05] F. A. Aagesen, P. Supadulchai, C. Anutariya, and M. M. Shiaa, “Configuration Management for an Adaptable Service System”, *IFIP International Conference on Metropolitan Area Networks MAN’05,* Ho Chi Minh city, Viet Nam, April 2005.

[AS84] Bowen Alpern and Fred B. Schneider, “Defining Liveness”, *Technical Report – Number: TR85-650*, Cornell University, Computer Science Department, April 19, 1994.

[BCS00] P. Bellavista, A. Corradi and C. Stefanelli, "A mobile agent infrastructure for terminal, user, and resource mobility", *NOMS 2000 - IEEE/IFIP Network Operations and Management Symposium*, no. 1, pp. 877-890, September 2000.

[BDD99] Berndt H., Darmois E., Dupuy F., Inoue Y., Lapierre M., Minerva R., Minetti R., Mossotto C., Mulder H., Natarajan N., Sevcik M., and Yates M., “The TINA Book”, *Prentice Hall Europe* 1999.

[BH95-1] J.P. Bowen and M.G. Hinchey, “Seven more Myths of Formal Methods”, *IEEE Software*, 12(4):34-41, July 1995.

[BH95-2] J.P. Bowen and M.G. Hinchey, “Ten commandments of formal methods”, *IEEE Computer*, 28(4):56-63, April 1995.

[BPW98] Andrzej Bieszczad, Bernard Pagurek and Tony White, “Mobile Agents for Network Management”, *IEEE Communications Surveys*, volume 1 number 1, 1998.

[Bræ00] R. Bræk, “On Methodology using the ITU-T Languages and UML”, *Telektronikk*, 96 (4), pp. 96-106, 4/2000.

[BS00] Beverly, Schwartz, “Smart Packets: Applying Active Networks to Network Management”, *ACM Transactions on Computer Systems*, vol. 18, no. 1, February 2000. Pages 67-88.

[CFN03] C. Carrez, A. Fanatechi, and E. Najm, “Behavioural contracts for a sound assembly of components”, *The 23rd IFIP International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2003, IFIP TC 6 / WG 6.1), LNCS 2767*, Berlin, Germany, September 2003.

[CDE03] M.G. Clavel, F. Durnan, S. Eker, P. Lincoln, N. Martí-Olie, J. Meseguer, and C. Talcott, Maude 2.0 Manual, June 2003. , [http://maude.cs.uiuc.edu/]

[CEL96] M.G. Clavel, S. Eker, P. Lincoln, and J. Meseguer, “Principles of Maude”, *First Int. workshop on Rewriting logic, Electronic notes in Theoretical Computer Science* (ENTCS) 4, Elsevier, 1996.

[CKV01] A. T. Campbell, M. E. Kounavis and J. Vicente, "Programmable Networks", *Reinhard Wilhelm (ed.), Informatics, 10 Years Back, 10 Years Ahead, Springer-Verlag, Lecture Notes in Computer Science 2000*, pp. 34-49, 2001.

[CMK99] A. T. Campbell, H. G. De Meer, M. E. Kounavis, K. Miki, J. Vicente and D. A. Villela, "A Survey of Programmable Networks", *ACM Computer Communications Review*, Vol. 29, No. 2, pp. 7-24, April 1999.

[CORBA98] Object Management Group, “The Common Object Request Broker Architecture (CORBA): Architecture and Specification version 2.2”, February 1998.

[DG94] John Darlington and Yi-ke Guo, “Formalising Actors in Linear Logic” *In Proceedings of the International Conference on Object-Oriented Information Systems (OOIS'94)*, 1994.

[DH95] S. Deering and R. Hinden, “RFC 1883: Internet Protocol, version 6 (IPv6) specification”, *Internet Engineering Task Force (IETF)*, December 1995.

[DMT00] Grit Denker, Jose Meseguer and C. Talcott, “Formal Specification and Analysis of Active networks and communication protocols”, *DISCEX’2000, IEEE computer society press*, January 2000.

[DMT04] G. Denker, J. Meseguer and C. Talcott, ”Rewriting semantics of Meta-Objects and Composable Distributed services”, *Proceedings of CONCUR'99: Concurrency Theory*, Vol. 1664, pages 415-430, 1999.

[DN96] Joubine Dutszadeh and Elie Najm, “Formal Support for ODP and Teleservices”, *Proceedings of the IFIP/ICCC conference on Information Network and Data Communication*, June 1996.

[DN97] Joubine Dustzadeh, Elie Najm “Consistent Semantics for ODP Information and Computational Models”. *Proceedings of FORTE/PSTV'97*, Osaka, Japan, Chapman & Hall, November 97.

[DO05] Dennis O’Neil, Human Biological Adaptability, [http://anthro.palomar.edu/adapt/default.htm]

[DoD05] The Active Network program, “Smart Packets”, Advanced Technology Office, *U.S. Department of Defense*, [http://www.darpa.mil/ato/programs/activenetworks/actnet.htm].

[Flo03] Jacqueline Floch, “Towards Plug-and-Play Services: Design and Validation using roles”, *Doctoral Dissertation*, Department of Telematics – NTNU, February 2003.

[Hoa85] Hoare, C.A.R, “Communicating Sequential Processes”, *Prentice-Hall*, 1985.

[Hol04] Gerard Holzmann, “The Spin Model Checker”, *Addison-Wesley*, 2004.

[HP93] Christine R. Hofmeister and James M. Purtilo, “Dynamic reconfiguration in distributed systems: Adapting software modules for replacement”, *In Proceedings of the 13th International Conference on Distributed Computing Systems, IEEE Computer Society Press,* May 1993.

[GEN05] The Genesis project homepage, *Department of Electrical Engineering, Columbia University*, [http://www.comet.columbia.edu/genesis/]

[GGK02] K. Gottschalk, S. Graham, H. Kreger, and J. Snell, “Introduction to Web Services architecture”, *IBM Systems Journal*, Vol. 41, No. 2, 2002.

[GSM92] M. Mouly and M. Pautet, “The GSM System for Mobile Communications”, *Mouly & Pautet*, 49, rue Louise Bruneau, F-91120 PALAISEAU – FRANCW, 1992.

[GUP03] 3G TS 22.240 v6.1.0, “Service requirement for the 3GPP Generic User Profile (Release 6)”, *3GPP technical specification*, September 2003.

[Gur95] Y. Gurevich, “Evolving Algebras 1993: Lipari Guide”, *E. Borger, Editor, Specification and Validation Methods*, Oxford University Press, 1995, pages 9-36.

[Hen04] Fred Inge Henden, “Developing Role Figure Model based on UML Specification”, *Master thesis*, Department of Telematics, NTNU, 2004.

[IBM05] The autonomic computing project, *IBM*, [http://www.research.ibm.com/autonomic/]

[IN92] *ITU-T recommendation I.312*, “Principles of Intelligent Network architecture (IN)”, October 1992.

[ITU93] ITU-T Recommendation, “Specification and Description Language SDL-92”, *ITU-T recommendation Z.100*, (03/1993).

[ITU99] ITU-T Recommendation, “Specification and Description Language SDL-2000”, *ITU-T recommendation Z.100*, (11/1999).

[JA03] S. Jiang and F. A. Aagesen, “XML-based Dynamic Service Behaviour Representation”. *NIK03 workshop*, Oslo, Norway, November 2003.

[Lam77] Leslie Lamport, “Proving the Correctness of Multiprocess Program”, *IEEE Trans. On Software Engineering SE-3*, 2 (March 1977), 125-143.

[Lil03] Lars Erik Liljeback, “user and user session mobility in a plug-and-play architecture”, *Master thesis*, Department of Telematics, 2002.

[Luh03] Eirik Luhr, “Mobility Support for Wireless devices – within the TAPAS platform”, *Master thesis*, Department of Telematics, 2004.

[LYB02] A. Liotta, A. Yew, C. Bohoris, G. Pavlou, "Delivering Service Adaptation with 3G Technology", *Proceedings of the 13th IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM 2002), Montreal, Canada*, *Lecture Notes in Computer Science*, vol. 2506, pp.108-120, Springer, October 2002

[Mar04] Lab for Advanced Information Technology, *University of Maryland, Baltimore County,* KQML Web, [http://www.cs.umbc.edu/kqml/]

[Mau04] Maude homepage, *SRI International*, [http://maude.cs.uiuc.edu/]

[MEE02] 3G TS 22.057 v5.4.0, “Mobile Execution Environment (Release 5)”, *3GPP technical specification*, June 2002.

[Mes92] Jose Meseguer, “Conditional rewriting logic as a unified model of concurrency”, *Theoretical Computer Science*, 96:73-155, 1992.

[Mil89] Robin Milner, “Communication and Concurrency”, *Prentice-Hall*, 1989.

[MIT05] *MIT Media Laboratory*, “Challenger: A Multi-Agent System for Distributed Resource Allocation”, [http://challenger.www.media.mit.edu/projects/challenger/]

[MM93] Narciso Marti-Oliet and Jose Meseguer, “Rewriting logic as a logical and semantic framework”, *Technical Report, SRI International, Computer Science Laboratory*, August 1993.

[MPW92] Robin Milner, Joachim Parrow, David Walker, “A calculus of mobile processes (parts I and II)”, *Information and Computation*, 100:1-77, 1992.

[NS95] E. Najm, J.B. Stefani, “A formal semantics for the ODP formal model”, *Computer Networks and ISDN systems 27*, pp.1305-1329, 1995.

[ODP95-1] ISO/IEC 10746-1, ITU-T X.901, “Basic Reference Model of Open Distributed Processing – Part1: Overview”, *International Organization for Standardization*, 1995.

[ODP95-2] ISO/IEC 10746-1, ITU-T X.901, “Basic Reference Model of Open Distributed Processing – Part2: Foundations”, *International Organization for Standardization*, 1995.

[ODP95-3] ISO/IEC 10746-1, ITU-T X.901, “Basic Reference Model of Open Distributed Processing – Part3: Architecture”, *International Organization for Standardization*, 1995.

[OSA03] 3G TS 21.905 v6.7.0, “Vocabulary for 3GPP Specifications (Release 6)”, *3GPP technical specification*, June 2003.

[Par04] “The Parlay specifications”, *The Parlay Group*, homepage at [http://www.parlay.org]

[Ped00] B. Møller-Pedersen, “SDL combined with UML”, *Telektronikk*, 96 (4), pp. 36-53, 4/2000.

[PEN05] “The SwitchWare Project”, *University of Pennsylvania, Department of Computer and Information Science and Bellcore*, [http://www.cis.upenn.edu/~switchware/]

[PER02] C. Perkins, “RFC 3344: IP mobility support for IPv4”, *Internet Engineering Task Force (IETF)*, August 2002.

[PER96] C. Perkins, “RFC 2002: IP mobility support”, *Internet Engineering Task Force (IETF)*, October 1996.

[PER97] C. Perkins, “Mobile IP”, *IEEE Communication Magazine*, pp. 84-99, May, 1997.

[PME02] 3G TS 22.022 v5.0.0: *3GPP technical specification* “Personalization of Mobile Equipment (Release 5)”. September 2002.

[POS93] J. Postel, “RFC0791: Internet Protocol”, *Internet Engineering Task Force (IETF)*, Augustus 1993.

[RB99] S. K. Raza and A. Bieszczad, “Network Configuration with Plug and Play Components”, *The Sixth IFIP/IEEE International Symposium on Integrated Network Management*, May, 1999.

[RPM00] G. C. Roman, G. P. Picco, A. L. Murphy, “Software Engineering for Mobility: A Roadmap”, In *The Future of Software Engineering*, Anthony Finkelstein (Ed.), pp. 243-258, ACM Press 2000.

[SA102] M. M. Shiaa and F.A. Aagesen, “Mobility management in a Plug and Play Architecture”, *Proc. IFIP 7th Int’l Conf. Intelligence in Networks (SmartNet’2002)*, Saariselka, Finland, April 2002.

[SA202] M. M. Shiaa and F.A. Aagesen, “Architectural Considerations for Personal Mobility in the Wireless Internet”, Proc. *IFIP TC/6 Personal Wireless Communications (PWC’2002)*, Singapore, October 2002.

[San00] Richard Sanders, “Implementing from SDL”, *Telektronikk*, 96 (4), pp. 120-129, 4/2000.

[SG90] James W. Stamos and David K. Gifford, “Remote EValuation”, *ACM Transcations on Programming Languages and Systems*, Vol. 12, No. 4, October 1990, Pages 537-565.

[Shi03] M. M. Shiaa, “Mobility Support Framework in Adaptable Service Architecture”. *Network Control and Engineering for QoS, Security and Mobility 2003 IFIP/IEEE Conference (NetCon'2003)*, Muscat-Oman, October 2003.

[Sim04] Audun Simonsen, “Developing mobile applications - The development of a fleet steering application with mobility support”, *Project report*, Department of Telematics, NTNU, 2004.

[SJS04] M. M. Shiaa, S. Jiang, P. Supadulchai, and J. J. Vila-Armenegol, “An XML based Framework for Dynamic Service Management”. *IFIP International Conference, INTELLCOMM 2004*, Bangkok-Thailand, November 2004.

[SL02] M. M. Shiaa and L.E. Liljeback, “User and Session Mobility in a Plug-and-Play Network Architecture”, *Proc. IFIP WG6.7 Workshop on Adaptable Networks and Teleservices*, Trondheim, Norway, September 2002.

[Smi04] M. Smirnov, “Autonomic Communication Roadmap”, *Presentations at Autonomic Communication Networking Session (N60) at the IST 2004 Event*, The Hague, Netherlands, November 2004, [http://www.autonomic-communication.org/]

[SRO04] 3G TS 22.127 v6.3.0, “Service Requirement for the Open Service Access (Release 6)”, *3GPP technical specification*, June 2004.

[SSC03] 3G TS 22.105 v6.2.0, “Service and Service Capabilities (Release 6)”, *3GPP technical specification*, June 2003.

[Str04] John Strassner, “Autonomic Networking – Theory and Practice”, *IEEE/IFIP Network Operations and Management Symposium, NOMS’2004*, April 19-23, 2004.

[Tal96] Carolyn L. Talcott, “An Actor Rewriting Theory”, *First Int. workshop on Rewriting logic, Electronic notes in Theoretical Computer Science* (ENTCS) *4*, Elsevier, 1996.

[TH91] M. Theimer, B. Hayes, “Heterogeneous Process Migration by Recompilation”, *Proceedings of the 11th International Conference on Distributed Computing*, pp. 18-25, 1991

[Tha97] Do Van Thanh, “Mobility as an Open Distributed Processing Transparency”, *Doctoral Dissertation*, Department of Telematics – NTNU, April 1997

[TINA95] “TINA-C deliverable: Overall Concepts and Principles of TINA V1.0”, *TINA Consortium*, February 1995. [http://www.tinac.com]

[TINA97] “TINA Service Architecture, version 5.0”, *TINA Consortium*, June 1997. [http://www.tinac.com]

[TINA98] Project608, TINA concepts for 3rd Generation Mobile Systems, “D3: Convergence of TINA and UMTS concepts”, *Eurescom publications*, June, 1998, [http://www.eurescom.de/]

[Tir98] T. Tiropanis; "Offering role mobility in a TINA environment"; Proceedings of the fifth international conference on intelligence in services and networks, IS&N '98, Antwerp, Belgium, May 1998.

[TLL99] Tzifa, Louta, Liossis, Kaltabani, Polydorou, Demestichas, and Anagnostou, “Open Service Architecture with Personal Mobility Support and Accounting and Charging Capabilities”, *European Multimedia, Embedded Systems and Electronic Commerce Conference EMMSEC99*, Stockholm, Sweden, 21-23 June 1999.

[TSS97] David L. Tennenhouse, Jonathan M. Smith, David Sincoskie, David J. Wetherall and Gary J. Minden, “A Survey of Active Network Research”, *IEEE Communications Magazine*, Volume 35 no 1, 1997, pages 80-86.

[VHE02] 3G TR 22.121 v5.3.1: *3GPP technical specification*, “The Virtual Home Environment (Release 5)”, June 2002.

[VS04] Vivek Sawant, “A Survey of Active Networks Programming Interfaces” *Computer Science Department, University of North Carolina*, Chapel Hill. [http://www.cs.unc.edu/~vivek/research/391-Fall00/survey-anpi.html]

[Wal97] Charles Wallace, “The semantics of the Java programming language: Preliminary version.” *University of Michigan Technical Report CSE-TR-355-97* [http://www.eecs.umich.edu/gasm/papers/javawall.html]

[WMG00] B. Wang, Jose Meseguer and C. Gunter, “Specification and Formal Analysis of PLAN algorithm in Maude”, *Proceedings of Int. workshop on Distributed system validation and verification*, April 2000.

[WPB99] White T., Pagurek B., and Bieszczad A. (1999), “*Network Modeling For Management Applications Using Intelligent Mobile Agents*”, Journal of Network and Systems Management, Special Issue on Mobile Agent-based Network and Service Management, September 1999.

[YSF99] Y. Yemini, S. da Silva, D. Florissi, and H. Huang, “The Network Flow Language: A Mark-based Approach to Active Networks”, *Technical Report, Columbia University Computer Science Department*, July 1999.

APPENDIX I TAPAS Architecture

Appendix I.1 TAPAS – The Service Management Architecture

The framework for Service Management Architecture is the extension of TAPAS that handles the deployment and management of services. This appendix provides a brief introduction to this architecture, for more details see [SJS04]. The framework is illustrated in Figure i. It is centred at the role of a supervisory Role-Figure, the Service Manager.

Services are instantiated and later modified via sending requests to this manager. A network node executes a State Machine Interpreter and has a set of capabilities. The interpreter can run Role-Figure specifications, which are state-machine-based. These specifications define Actions (functions and tasks to be performed by the Role-Figure during a specific state,) and their Action Groups (classification of actions, e.g. actions such as terminate, exit, error handling, etc. can be classified into the Action Group "*Control Functions*".) This technique is used to tackle the problem of platform and implementation independence, as well as achieving a better flexibility and reusability in service and application design. In the framework we also use the classification of capabilities into capability categories, to indicate the operating circumstances, e.g. Powerful-PDA, Basic-PDA, Smart-Phone. The components of the framework will be described shortly.

- *Play Repository* is a database that contains the service definitions and includes: 1) *Role-Figure Specifications* 2) *Mapping Rules* (specify the mapping between the capabilities and their categories.)
- *Capability and Status Repository (CSRep)* is a database that provides a snapshot of the resources of the system. It maintains information on all capability and status data in all system nodes.
- *Action Library* is a database that contains the executable routines for the actions. These routines will be executed by the role-figures. They are implemented according to the capability information they require.

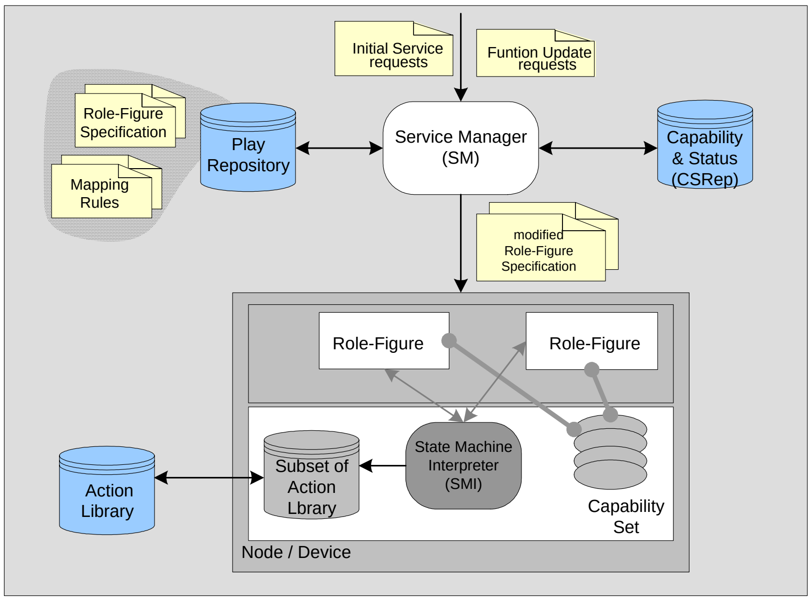


Figure i Dynamic Service Management Framework.

- *Service Manager (SM)* is responsible for the handling of *Initial Service requests*, and *Function Update requests*. It, first, calculates the offered Capability Category according to the *Mapping Rules*. Second, it modifies the *Role Figure Specification* by adding the corresponding Capability Category information. This modified *Role Figure Specification* is sent to the *State Machine Interpreter* for execution.
- *Requests* there are two service requests that may be handled by the SM: 1) *Initial Service request* indicates a role to be executed in a node, 2) *Function Update request* is issued to update a functionality due to a capability change.
- *State Machine Interpreter (SMI)* is a state machine execution support. This is the primary entity in the framework responsible for the execution of the Role-Figures according to the *Role-Figure Specification*.

Appendix I.1 TAPAS – The Dynamic Configuration Architecture

A Dynamic Configuration Architecture as an extension to TAPAS has been developed. This appendix provides a brief introduction to this architecture, for more details see [AAS02] and [ASA05]. This framework is depicted in Figure ii. The framework comprises the following main entities:

- *Capability & Status Repository (CSRep)* this has the same function as in the Service Management Architecture.
- *Play Repository* comprises a set of *Role-Figure Specifications*, and two sets of rules. *Play configuration rules* describe service system configuration constraints, such as specification of the maximum number of roles allowed to install at a specific node. *Reconfiguration rules* define application-specific reconfiguration policies for handling substantial reconfiguration-related events. Examples of such events include a service component failure, a decrease in system QoS and resource unavailability. With application-specific reconfiguration rules, the system can perform appropriate actions to alleviate a problem in a running system.
- *Capability, Status & Event Monitor (CSEMon)* monitors the system capabilities/status and also maintains the CSRep. Moreover, it listens to certain events indicating changes to the system and its environment, which would prevent the system from getting the desired level of services. In response to such events, it notifies the Configuration Manager for further proper reactions.
- *Requests* There are three kinds of requests; *Trouble report* (is an event triggered by the deterioration in the resource availability utilized by services), *Service Request* (issued to install and execute a particular service system, which has not yet been installed), and *Service Component Request* (is used for the instantiation of a particular service component as part of a running service system.)

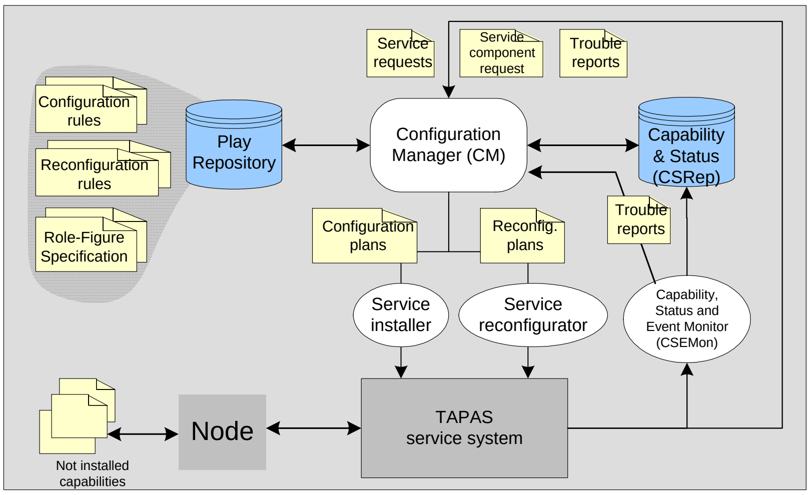


Figure ii Architectural framework for dynamic configuration.

o *Configuration Manager (CM)* is responsible for generating the appropriate configurations and reconfigurations for a system. When there arises a request for installing a new service (i.e., a *service request*), the CM fetches a corresponding play definition and retrieves the system capabilities and status from the *PlayRep* and the *CSRep*, respectively. Valid configurations for such a service are then generated and analyzed. Such a configuration that defines which nodes in the system should execute actors constituting certain Role-Figures will be forwarded to and executed by the *Service Installer*. The determination of a location for executing a particular role (i.e., in the case of a *service component request*) the CM dynamically determines the best location (node) for its installation, based on the current system configuration, available capabilities and status as well as the component's requirements. It then notifies the *Service Installer* to load a corresponding manuscript from the *PlayRep* and instantiates it on the suggested node. Thirdly, upon the receipt of *a trouble report* indicating a problem in a running system, the CM analyzes the problem, fetches related information from the *CSRep* and the *PlayRep*, and produces a service reconfiguration plan to be executed by the *Service Reconfigurator*. Possible plans include role-figure relocation, re-initialization, load balance and distribution. Selection of an appropriate plan depends on the defined reconfiguration rules as well as the nature of a problem.

o *Service Installer* is responsible for the installation of a service into the system by creating corresponding actors for execution of certain roles, according to an obtained play configuration generated by the CM. Allocation of capabilities as well as instantiation of a manuscript for each role are also performed by this entity.

- *Service Reconfigurator* initiates and performs reconfiguration of a service system based on an obtained reconfiguration plan.

APPENDIX II Rewriting Logic

A rewriting theory is a 4-tuple $\mathcal{R} = (\sum, E, L, R)$, where $\sum$ is a ranked alphabet of operator symbol, E is a set of $\sum$-equations, L is a set of labels, and R is a set of labelled conditional rewriting rules. Sentences in rewriting logic are sequent of the form $[t]_E \rightarrow [t']_E$ (read: [t] becomes [t′]), where t, t′ are $\sum$-terms, possibly involving some variables from a countable infinite set of variables $X =_{\text{def}} \{x_1, x_2, \ldots\}$. A theory $\mathcal{R}$ in this logic consists of a set of rules of the form:

$$r : [t] \rightarrow [t'] \text{ if } C$$

where r is a label from the label set L, C is a condition of the form: $[u_1] \rightarrow [v_1] \wedge \ldots \wedge [u_n] \rightarrow [v_n]$, and $[u_i],[v_i]$ are $\sum$-terms, possibly with variables in X.

Given a rewriting theory $\mathcal{R}$, we say that $\mathcal{R}$ entails a sequent [t]→[t'] and write $\mathcal{R} \vdash [t] \rightarrow [t']$ if and only if [t]→[t′] can be obtained by finite application of the following *rules of deduction:*

- **Reflexivity.**

For each $[t] \in T_{\sum,E}(X)$, $\dfrac{}{[t] \rightarrow [t]}$

- **Congruence.**

For each $f \in \sum$, $n \in$ IN, $\dfrac{[t_1] \rightarrow [t_1'] \cdots [t_n] \rightarrow [t_n']}{[f(t_1 \cdots t_n)] \rightarrow [f(t_1' \cdots t_n')]}$

- **Replacement.**

For each rewriting rule r: $[t(x_1 \cdots x_n)] \rightarrow [t'(x_1 \cdots x_n)]$

$$\frac{[\omega_1] \rightarrow [\omega_1'] \cdots [\omega_n] \rightarrow [\omega_n']}{[t(\overline{\omega}/\overline{x})] \rightarrow [t'(\overline{\omega'}/\overline{x})]}$$

- **Transitivity.** $\dfrac{[t_1] \rightarrow [t_2] \quad [t_2] \rightarrow [t_3]}{[t_1] \rightarrow [t_3]}$

- **Symmetry.** $\dfrac{[t_1] \rightarrow [t_2]}{[t_2] \rightarrow [t_1]}$

Rewriting Logic, as other logics, should be understood as a method of correct reasoning about some class of entities, which are, in this case, concurrent systems having states, and evolving by means of transitions.

The following set of analogies can be used as roadmap for using rewriting logic as a semantic framework for concurrent systems:

State	$\leftrightarrow$	*Term*
Transition	$\leftrightarrow$	*Rewriting*
Distributed Structure	$\leftrightarrow$	*Algebraic Structure*

In this thesis we used the following specialization for Rewriting Logic in our Role-figure Model:

RWL$\rightarrow$RWL$_{\mathrm{ACI}}$$\rightarrow$ConcOOP (Rewriting Logic $\rightarrow$ rewriting modulo Associativity, Commutativity and Identity $\rightarrow$ Concurrent Object Oriented Programming).

Rewriting logic is reflective, which means that there is a finitely presented rewrite theory $\mathcal{U}$ that is *universal* in the sense that we can represent any finitely presented rewrite theory $\mathcal{R}$ (including $\mathcal{U}$ itself) and any term t, t′ in $\mathcal{R}$ as terms $\overline{\mathcal{R}}$ and $\bar{t}$, $\bar{t}'$ in $\overline{\mathcal{U}}$, and that we then have the following equivalence:

$$\mathcal{R} \vdash t \rightarrow t' \iff \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \bar{t} \rangle \rightarrow \langle \overline{\mathcal{R}}, \bar{t}' \rangle$$

The theory $\mathcal{U}$ can be seen as a universal interpreter for rewriting logic which can simulate the rewrites pf any given rewrite theory $\mathcal{R}$.

APPENDIX III The full Role-figure Model semantics

In this Appendix the full semantics of the Role-Figure Model is presented. The following notations are used in the semantics:

▪ ***a***, ***b***, ***f***, ***g***, ***h***	denote Role-Figure names;
▪ *RoleFigures*	is the set of all existing Role-Figures in a given system;
▪ $\mathcal{A},\mathcal{A}',\ldots$ $\mathcal{B},\mathcal{B}',\ldots$	denote Role-Figures ***a*** and ***b*** as they evolve, respectively;
▪ *i*, *j*	denote interface names;
▪ α, α°	denote interface types;
▪ $\langle w_1 = \nu_1, w_2 = \nu_2 \rangle$	denotes the record containing two fields named w_1 and w_2 and having the values ν_1 and ν_2, respectively;
▪ $r.w_1$	stands for the value of the w_1 field in record r;
▪ $=_{\text{def}}$	stands for equality by definition;
▪ $\|\|$	denotes the asynchronous parallel operator;
▪ $\lhd$	denotes an infix insert operator;
▪ $\overset{n}{\underset{i=1}{\lhd}}$	denotes applying the operator $\lhd$ n times
▪ $\rhd$	denotes an infix remove operator.

The operators $||$, $\lhd$ and $\rhd$ are commutative, associative, and have $\varnothing$ as a neutral element. The insert operator "a$\lhd$b" only executes if its left-hand side argument, a, is not in its right-hand side argument, b. Otherwise it does nothing. The remove operator "a$\rhd$b" only executes if its left-hand side argument, a, is in its right-hand side argument, b. Otherwise it does nothing.

$$RFC \; ::= \; \varnothing \; | \; RFCE \; | \; RFC \,||\, RFC$$
$$RFCE \; ::= \; RF \,|\, MSG$$
$$RF \; ::= \; \langle Int = \gamma, Beh = \beta, Cap = \pi, Que = \omega, Met = \mu \rangle$$
$$MSG \; ::= \; Req \,|\, Sig \,|\, Ret$$

$$\gamma \quad ::= \quad \varnothing \mid \gamma \lhd [j:\alpha] \mid \gamma \rhd [j:\alpha]$$
$$\pi \quad ::= \quad \varnothing \mid \pi \lhd [c:cn] \mid \pi \rhd [c:cn]$$
$$\omega \quad ::= \quad \varnothing \mid \omega \lhd [q] \mid \omega \rhd [q]$$
$$\mu \quad ::= \quad \varnothing \mid \mu \lhd [m:mn] \mid \mu \rhd [m:mn]$$
$$Req \quad ::= \quad \left\langle \begin{array}{l} tar = j:\alpha, src = a, met = m:mn, \\ ref = n:mn, ret = r, arg = \tilde{p} \end{array} \right\rangle$$
$$Sig \quad ::= \quad \langle tar = j:\alpha, src = a, name = sig, arg = \tilde{p} \rangle$$
$$Ret \quad ::= \quad \langle tar = j:\alpha, src = a, ref = n:mn, arg = \tilde{p} \rangle$$
$$\tilde{p} \quad ::= \quad (p_1 : t_1, \cdots, p_n : t_n)$$
$$\alpha \quad ::= \quad \langle m_1 : methsig, \cdots m_n : methsig, sig_1, \cdots, sig_k \rangle$$
$$methsig \quad ::= \quad argument \to return$$
$$argument \quad ::= \quad Nil \mid \tilde{p}$$
$$\tilde{p} \quad ::= \quad (p_1 : t_1, \cdots, p_n : t_n)$$
$$\beta \quad ::= \quad \langle B = b : behaviour, St = st : state, Sg = sg : \tilde{g}, Sc = sc : \tilde{s}, Ss = ss : \tilde{s} \rangle$$
$$\tilde{s} \quad ::= \quad (state_1 \cdots, state_f)$$
$$\tilde{g} \quad ::= \quad (sig_1, \cdots, sig_f)$$

♣ 1: $\mathcal{A} \parallel \mathcal{T} \parallel \mathcal{Q} \parallel \mathcal{M} \to \mathcal{A}' \parallel \Sigma \parallel \mathcal{T}' \parallel \mathcal{Q}' \parallel \mathcal{M}'$ if C

The ♣|$_{\text{conditions}}$:

(i) $\mathcal{T} \cap \mathcal{T}' = \mathcal{Q} \cap \mathcal{Q}' = \mathcal{M} \cap \mathcal{M}' = \varnothing$

(ii) $\mathcal{A} \notin \Sigma$

(iii) Assume:

$s = \langle tar = i:\alpha, src = a, met = m:mn, ref = n:mn, ret = r, arg = \tilde{p} \rangle$ is a *request* sent by Role-Figure ***a*** (to apply ♣ assume: ***a*** evolves from $\mathcal{A}$ to $\mathcal{A}'$ after the sending, and $s \in \mathcal{M}'$). Then the following hold:

- $i \in \mathcal{A}.Int$
- $\exists$***b***, ***b***$\in$ *RoleFigures* AND *i* is an interface to ***b***
- The signature of the invoked method m is included in the definition of α.
- If the invoked method m has the following method signature ($methsig_m ::= argument_m \to return_m$)
  THEN ($r =_{\text{def}} return_m$) AND ($\tilde{p} =_{\text{def}} argument_m$)

- $n \in \mathcal{A}'.Met$

Assume now that Role-Figure ***b*** receives this *request* in another rewriting rule (to apply ♣ assume: ***b*** evolves from $\mathcal{B}$ to $\mathcal{B}'$ after the receiving, and $s \in \mathcal{M}$) then the following hold:

- $\exists\, [j : \alpha^\circ], j \in \mathcal{B}.Int$ and $\alpha^\circ \preceq \alpha$ (this means α° is a subtype of α)
- $s \in \mathcal{B}'.Que$
- If the invoked method m has the following method signature ($methsig_m$ ::= $argument_m \rightarrow return_m$)
- THEN ($r =_{\mathrm{def}} return_m$) AND ($\tilde{p} =_{\mathrm{def}} argument_m$)

(iv) Assume:

$s = \langle tar = i : \alpha, src = a, name = sig, arg = \tilde{p} \rangle$ is a *signal* sent by Role-Figure ***a*** (to apply ♣ assume: ***a*** evolves from $\mathcal{A}$ to $\mathcal{A}'$ after the sending, and $s \in \mathcal{Q}'$). Then the following hold:

- $i \in \mathcal{A}.Int$
- $\exists$***b***, ***b***$\in$ *RoleFigures* AND *i* is an interface to ***b***
- The signal name and its arguments are included in the definition of α.

Assume now that Role-Figure ***b*** receives this *signal* in another rewriting rule (to apply ♣ assume: ***b*** evolves from $\mathcal{B}$ to $\mathcal{B}'$ after the receiving, and $s \in \mathcal{Q}$) then the following hold:

- $\exists\, [j : \alpha^\circ], j \in \mathcal{B}.Int$ and $\alpha^\circ \preceq \alpha$
- $s \in \mathcal{B}'.Que$

(v) Assume:

$s = \langle tar = i : \alpha, src = a, ref = n : mn, arg = \tilde{p} \rangle$ is a *return* sent by Role-Figure ***a*** as a response to an invocation request received from another Role-Figure (to apply ♣ assume: ***a*** evolves from $\mathcal{A}$ to $\mathcal{A}'$ after the sending, and $s \in \mathcal{T}'$). Then the following hold:

- $i \in \mathcal{A}.Int$
- $\exists$***b***, ***b***$\in$ *RoleFigures* AND *i* is an interface to ***b***

Assume now that Role-Figure ***b*** receives this *return* in another rewriting rule (to apply ♣ assume: ***b*** evolves from $\mathcal{B}$ to $\mathcal{B}'$ after the receiving, and $s \in \mathcal{T}$) then the following hold:

- $\exists\, [j : \alpha^{\circ}]$, $j \in \mathcal{B}.Int$ and $\alpha^{\circ} \preceq \alpha$
- $n \in \mathcal{B}.Met$
- $s \in \mathcal{B}'.Que$

(vi) A *request*, *signal*, or *return* may be consumed from the queue of a Role-Figure:

$$\mathcal{A}.Que \triangleright s$$

(vii) In the rewriting rule ♣:

- Either ($\mathcal{A}.Int \subseteq \mathcal{A}'.Int$), ($\mathcal{A}.Int = \mathcal{A}'.Int$), or ($\mathcal{A}'.Int \subseteq \mathcal{A}.Int$) hold in one rewriting transition.
- Either ($\mathcal{A}.Met \subseteq \mathcal{A}'.Met$), ($\mathcal{A}.Met = \mathcal{A}'.Met$), or ($\mathcal{A}'.Met \subseteq \mathcal{A}.Met$) hold in one rewriting transition.
- Either ($\mathcal{A}.Que \subseteq \mathcal{A}'.Que$), ($\mathcal{A}.Que = \mathcal{A}'.Que$), or ($\mathcal{A}'.Que \subseteq \mathcal{A}.Que$) hold in one rewriting transition.
- Either ($\mathcal{A}.Cap \subseteq \mathcal{A}'.Cap$), ($\mathcal{A}.Cap = \mathcal{A}'.Cap$), or ($\mathcal{A}'.Cap \subseteq \mathcal{A}.Cap$) hold in one rewriting transition.

(viii) In the rewriting rule ♣, if *BehaviourChange* is not applied then the following hold:

- Either ($\mathcal{A}.Beh.St = \mathcal{A}'.Beh.St$) or ($\mathcal{A}'.Beh.St \in \mathcal{A}.Beh.Sc$)

(ix) If $\aleph = RoleFigures \cup \sum$ and $\Im = \bigcup_{A \in \aleph} A.Int$

then

- $\forall \boldsymbol{i}{:}\alpha,\ \boldsymbol{i} \in \Im$ and $\forall \boldsymbol{a},\ \boldsymbol{a} \in \aleph$
- $\forall r{:}Req,\ r.tar \in \Im$ and $r.src \in \aleph$
- $\forall s{:}Sig,\ s.tar \in \Im$ and $s.src \in \aleph$
- $\forall r{:}Ret,\ r.tar \in \Im$ and $r.src \in \aleph$

(x) $\forall \mathcal{B} \in \sum$, then $\mathcal{B}.Met = \varnothing$ and $\mathcal{B}.Que = \varnothing$

APPENDIX IV The Maude specification

In this Appendix we present the Maude specification of the Role-Figure Model.

```
**********************************************************************************************************
*********                          The Role-Figure Maude module                                  ********
**********************************************************************************************************
 (omod RoleFigureModel is

*** sort definitions
sort Type .
sorts Loc OidSet Content State StateSet Beh BehType Signal SignalSet Cap CapSet Mes MesSet
Met MetSet .
subsort Oid < OidSet .
subsort State < StateSet .
subsort Signal < SignalSet .
subsort Cap < CapSet .
subsort Met < MetSet .
subsort Mes < MesSet .

*** the Role-Figure and the Director classes
class  Actor | Int : OidSet, Beh : BehType, Cap : CapSet,
           Que : MesSet, Met : MetSet .
class  Director | Int : OidSet .
class  RoleFigure | Location : Loc, Int : OidSet, Beh : BehType, Cap : CapSet,
           Que : MesSet, Met : MetSet .

*** TAPAS messages:
*** the plug in requests
msg pi-tar_src_name_location_ : Oid Oid Oid Loc -> Msg .
msg pi-tar_src_name_location_with_ : Oid Oid Oid Loc Content -> Msg .
msg pi-tar_src_name_location_with_ : Oid Oid Oid Loc Type -> Msg .
msg pi-tar_src_name_location_with__ : Oid Oid Oid Loc Content Type -> Msg .
msg pi-tar_src_name_location_with__ : Oid Oid Oid Loc BehType CapSet -> Msg .
msg pi-tar_src_name_location_with___ : Oid Oid Oid Loc BehType CapSet Type -> Msg .
*** the plug out requests
msg po-tar_src_name_ : Oid Oid Oid -> Msg .
msg po-tar_src_name__ : Oid Oid Oid Type -> Msg .
*** the create interface request
msg ci-tar_src_j_ : Oid Oid OidSet -> Msg .
*** the behaviour change requests
msg bc-tar_src_beh_ : Oid Oid BehType -> Msg .
msg bc-tar_src_beh____ : Oid Oid BehType State StateSet StateSet -> Msg .
*** the capability change requests
msg cc-tar_src_p_ : Oid Oid CapSet -> Msg .
msg cc-tar_src_p__ : Oid Oid CapSet Type -> Msg .
*** the move requests
msg mo-tar_src_location__with_ : Oid Oid Oid Loc Content -> Msg .
msg mo-tar_src_location__with__ : Oid Oid Oid Loc Content Type -> Msg .
*** the move returns
msg mor-tar_src_to__ : Oid Oid Oid Loc -> Msg .
msg mor-tar_src_to__with_ : Oid Oid Oid Loc Type -> Msg .

*** general message
msg msg_from_to_ : Content Oid Oid -> Msg .

*** multimessage-declaration
op multimsg_from_to_ : Content Oid OidSet -> Configuration .

*** operator declarations
*** constants
ops st1 st2 : -> State .                          *** states
ops sts1 sts2 : -> StateSet .                     *** state sets
ops bh1 bh2 : -> Beh .                            *** behaviours
ops m remInt creInt moving : -> Content .         *** message content
ops director a a' b c e : -> Oid .                *** Role-Figures
ops n n' : -> OidSet .                            *** Role-Figure sets
ops bht1 bht2 : -> BehType .                      *** behaviour type
ops sig1 sig2 : -> Signal .                       *** signals
ops sigs1 sigs2 : -> SignalSet .                  *** signal sets
ops cap1 cap2 : -> Cap .                          *** capabilities
ops caps1 caps2 : -> CapSet .                     *** capability sets
ops met1 met2 : -> Met .                          *** methods
ops mets1 mets2 : -> MetSet .                     *** method sets
ops mes1 mes2 : -> Mes .                          *** messages
ops mess1 mess2 : -> MesSet .                     *** message sets
ops loc1 loc2 : -> Loc .                          *** locations

*** [multi]set constructors:
op nil : -> OidSet .
op nilOid : -> Oid .
op nilcaps : -> CapSet .
op nilbht : -> BehType .
op nilmess : -> MesSet .
op nilmets : -> MetSet .
op __ : OidSet OidSet -> OidSet [assoc comm id: nil prec 15] .

*** set-operations:
op in : Oid OidSet -> Bool .
op subseteq : OidSet OidSet -> Bool .
op _-_ : OidSet Oid -> OidSet .       *** Set minus one Oid!

*** example configurations for testing:
ops testS1 testS11 testS12 testS13 testS2 testS21 testS3 testS4 testS5 testS6: -> Configuration .

*** variable declaration
vars A A' B C D : Oid .                           *** Role-Figures
vars N N' N'' : OidSet .                          *** Role-Figure sets
vars M : Content .                                *** message content
vars ST1 ST2 : State .                            *** state sets
vars STS1 STS2 : StateSet .                       *** state sets
vars BH1 BH2 : Beh .                              *** behaviours
vars BHT1 BHT2 : BehType .                        *** behaviours types
vars SIG1 SIG2 : Signal .                         *** signals
vars SIGS1 SIGS2 : SignalSet .                    *** signal sets
vars CAP1 CAP2 : Cap .                               *** capabilities
vars CAPS1 CAPS2 : CapSet .                          *** capability sets
vars MET1 MET2 : Met .                               *** methods
vars METS1 METS2 : MetSet .                          *** method sets
vars MES1 MES2 : Mes .                               *** messages
vars MESS1 MESS2 : MesSet .                          *** message sets
vars LOC1 LOC2 : Loc .                               *** locations

*** equations on sets:
eq A A = A .
eq in(A, B N) = A == B or in(A,N) .
eq in(A,nil) = false .
eq subseteq(A N ,N') = in(A,N') and subseteq(N,N') .
eq subseteq(nil,N') = true .
eq (A N) - A = N - A .
ceq N - A = N if not in(A,N) .

*** multimessage definition:
ceq multimsg M from A to (B N) =
        (msg M from A to B) (multimsg M from A to (N - B)) if not M == creInt .

eq multimsg creInt from A to (B N) =
        (ci-tar B src A j A) (ci-tar A src B j B) (multimsg creInt from A to (N - B)) .

eq multimsg M from A to nil = none .


*** example configurations for testing
eq testS1 =
  (po-tar e src director name e)
  < director : Director | Int : a b c e >
  < a : RoleFigure | Location : loc1, Int : director b c e, Beh : bht1, Cap : caps1, Que : mess1, Met :
mets1 >
  < b : RoleFigure | Location : loc1, Int : director a c e, Beh : bht2, Cap : caps2, Que : mess2, Met :
mets2 >
  < c : RoleFigure | Location : loc1, Int : director a b e, Beh : bht1, Cap : caps1, Que : mess1, Met :
mets1 >
  < e : RoleFigure | Location : loc1, Int : director a b c, Beh : bht2, Cap : caps2, Que : mess2, Met :
mets2 > .

eq testS11 =
  (po-tar director src a name e)
  < director : Director | Int : a b c e >
  < a : RoleFigure | Location : loc1, Int : director b c e, Beh : bht1, Cap : caps1, Que : mess1, Met :
mets1 >
  < b : RoleFigure | Location : loc1, Int : director a c,   Beh : bht2, Cap : caps2, Que : mess2, Met :
mets2 >
  < c : RoleFigure | Location : loc1, Int : director a b e, Beh : bht1, Cap : caps1, Que : mess1, Met :
mets1 >
  < e : RoleFigure | Location : loc1, Int : director a c,   Beh : bht2, Cap : caps2, Que : mess2, Met :
mets2 > .

eq testS12 =
  (msg remInt from e to director)
  < director : Director | Int : a b c e > .
  < a : Actor | Int : director b c e, Beh : bht1, Cap : caps1, Que : mess1, Met : mets1 > .

eq testS13 =
  (po-tar e src director name e)(pi-tar director src a name e location loc1 with m)
  < director : Director | Int : a b c e >
  < a : RoleFigure | Location : loc1, Int : director b c e, Beh : bht1, Cap : caps1, Que : mess1, Met :
mets1 >
  < b : RoleFigure | Location : loc1, Int : director a c e, Beh : bht2, Cap : caps2, Que : mess2, Met :
mets2 >
  < c : RoleFigure | Location : loc1, Int : director a b e, Beh : bht1, Cap : caps1, Que : mess1, Met :
mets1 >
  < e : RoleFigure | Location : loc1, Int : director a b c, Beh : bht2, Cap : caps2, Que : mess2, Met :
mets2 > .

eq testS2 =
  (pi-tar director src a name e location loc1 with m)
  < director : Director | Int : a b c >
  < a : RoleFigure | Location : loc1, Int : director b c, Beh : bht1, Cap : caps1, Que : mess1, Met :
mets1 >
  < b : RoleFigure | Location : loc1, Int : director a c, Beh : bht2, Cap : caps2, Que : mess2, Met :
mets2 >
  < c : RoleFigure | Location : loc1, Int : director b a, Beh : bht1, Cap : caps1, Que : mess1, Met :
mets1 > .

eq testS21 =
  (pi-tar director src a name e location loc1 with bht1 caps1)
  < director : Director | Int : a b c >
  < a : RoleFigure | Location : loc1, Int : director b c, Beh : bht1, Cap : caps1, Que : mess1, Met :
mets1 >
  < b : RoleFigure | Location : loc1, Int : director a c, Beh : bht2, Cap : caps2, Que : mess2, Met :
mets2 >
  < c : RoleFigure | Location : loc1, Int : director b a, Beh : bht1, Cap : caps1, Que : mess1, Met :
mets1 > .

eq testS3 =
  (ci-tar e src a j c) (ci-tar c src a j e)
  < director : Director | Int : a b c e >
  < a : RoleFigure | Location : loc1,  Int : director b c e, Beh : bht1, Cap : caps1, Que : mess1, Met
: mets1 >
  < b : RoleFigure | Location : loc1,  Int : director a c, Beh : bht2, Cap : caps2, Que : mess2, Met :
mets2 >
  < c : RoleFigure | Location : loc1,  Int : director b a, Beh : bht1, Cap : caps1, Que : mess1, Met :
mets1 >
  < e : RoleFigure | Location : loc1,  Int : director a, Beh : bht2, Cap : caps2, Que : mess2, Met :
mets2 > .

eq testS4 =
  (bc-tar a src b beh bht2)
  < director : Director | Int : a >
  < a : RoleFigure | Location : loc1,  Int : director b n, Beh : bht1, Cap : caps1, Que : mess1, Met :
mets1 > .

eq testS5 =
  (cc-tar a src b p caps2)
  < director : Director | Int : a >
  < a : RoleFigure | Location : loc1,  Int : director b n, Beh : bht1, Cap : caps1, Que : mess1, Met :
mets1 > .

eq testS6 =
  (mo-tar a src b location a' loc2 with moving)
  < director : Director | Int : a b c >
  < a : RoleFigure | Location : loc1,  Int : director b c, Beh : bht1, Cap : caps1, Que : mess1, Met :
mets1 >
  < b : RoleFigure | Location : loc1,  Int : director a c, Beh : bht2, Cap : caps2, Que : mess2, Met :
mets2 >
  < c : RoleFigure | Location : loc1,  Int : director b a, Beh : bht1, Cap : caps1, Que : mess1, Met :
mets1 > .

*** The Role-Figure Model rewriting rules:

*** Create Interface:
crl [CreateInterfaceAtActorSemantics] :
  (ci-tar A src C j B)
  < A : RoleFigure | Location : LOC1, Int : N, Beh : BHT1, Cap : CAPS1, Que : MESS1, Met :
METS1 > =>
  < A : RoleFigure | Int : B N > if in(C,N) or (B == C) .

crl [CreateInterfaceAtDirectorSemantics] :
  (ci-tar A src C j B)
  < A : Director | Int : N > =>
  < A : Director | Int : B N > if in(C,N) or (B == C) .

*** Plug out method
crl [PlugOuttoActorSemantics] :
  (po-tar A src B name C)
  < A : RoleFigure | Location : LOC1, Int : N, Beh : BHT1, Cap : CAPS1, Que : MESS1, Met :
METS1 > =>
  (multimsg remInt from A to N) if (C == A) and (B == director) and in(B,N) .

crl [PlugOuttoDirectorSemantics] :
  (po-tar director src A name B)
  < director : Director | Int : N > =>
  < director : Director | Int : N > (po-tar B src director name B) if in(B,N) .

*** remove interface
crl [RemoveInterfacesAtActorSemantics] :
  (msg remInt from B to A)
  < A : RoleFigure | Int : N > =>
  < A : RoleFigure | Int : N - B >  if in(B,N) .

crl [RemoveInterfacesAtDirectorSemantics] :
  (msg remInt from B to A)
  < A : Director | Int : N > =>
  < A : Director | Int : N - B >  if in(B,N) .

*** Plug in without passing behaviour and capability
crl [PlugInNoContentSemantics] :
   (pi-tar D src A name B location LOC1)
   < director : Director | Int : N > =>
   < director : Director | Int : B N > < B : RoleFigure | Location : LOC1, Int : director A, Beh : nilbht,
Cap : nilcaps, Que : nilmess, Met : nilmets >
   (ci-tar A src director j B) if not in(B,N) and (D == director) .

***  Plug in for non-moving Role-Figure
crl [PlugInNotMovingSemantics] :
   (pi-tar D src A name B location LOC1 with M)
   < director : Director | Int : N > =>
   < director : Director | Int : B N > < B : RoleFigure | Location : LOC1, Int : director A, Beh : nilbht,
Cap : nilcaps, Que : nilmess, Met : nilmets >
   (ci-tar A src director j B) if not in(B,N) and (M =/= moving)  and (D == director) .

***  Plug in for moving Role-Figure
crl [PlugInMovingSemantics] :
   (pi-tar D src A name A' location LOC1 with M)
   < director : Director | Int : N > =>
   < director : Director | Int : A' N >
   < A' : RoleFigure | Location : LOC1, Int : director A, Beh : nilbht, Cap : nilcaps, Que : nilmess,
Met : nilmets >
   (mor-tar A src director to A' LOC1)
   if not in(A',N) and (M == moving)  and (D == director) .

*** Plug in with passing behaviour and capability, possibly to an existing Role-Figure
crl [PlugInwithCapBehSemantics] :
   (pi-tar D src A name B location LOC1 with BHT2 CAPS2)
   < director : Director | Int : N > =>
   < director : Director | Int : B N > < B : RoleFigure | Location : LOC1, Int : director A, Beh : BHT2,
Cap : CAPS2, Que : nilmess, Met : nilmets >
   (ci-tar A src director j B)  if (D == director) .

*** Behaviour Change:
crl [BehaviourChangeSemantics] :
   (bc-tar A src B beh BHT2)
   < A : RoleFigure | Location : LOC1, Int : N, Beh : BHT1, Cap : CAPS1, Que : MESS1, Met :
METS1 > =>
   < A : RoleFigure | Location : LOC1, Beh : BHT2 > if in(B,N) .

*** Capability Change:
crl [CapabilityChangeSemantics] :
   (cc-tar A src B p CAPS2)
   < A : RoleFigure | Location : LOC1, Int : N, Beh : BHT1, Cap : CAPS1, Que : MESS1, Met :
METS1 > =>
   < A : RoleFigure | Location : LOC1, Cap : CAPS2 > if in(B,N) .

*** Role-Figure Move:
crl [RoleFigureMoveSemantics] :
   (mo-tar A src B location A' LOC2 with M)
   < A : RoleFigure | Location : LOC1, Int : N, Beh : BHT1, Cap : CAPS1, Que : MESS1, Met :
METS1 > =>
   < A : RoleFigure | Location : LOC1, Int : N > (pi-tar director src A name A' location LOC2 with M)
if in(B,N) .
```

```
*** Role-Figure move return
crl [RoleFigureMoveReturnSemantics] :
  (mor-tar A src D to A' LOC2)
  < A : RoleFigure | Location : LOC1, Int : N, Beh : BHT1, Cap : CAPS1, Que : MESS1, Met :
METS1 > =>
  < A : RoleFigure | Location : LOC1, Int : A' N >
  (cc-tar A' src A p CAPS1) (multimsg creInt from A' to N) (bc-tar A' src A beh BHT1) (po-tar direc-
tor src A' name A) if (D == director) .

endom)

*** system runs to test the Role-Figure configurations

(rew testS1 .)
(rew testS11 .)
(rew testS12 .)
(rew testS13 .)
(rew testS2 .)
(rew testS21 .)
(rew testS3 .)
(rew testS4 .)
(rew testS5 .)
(rew testS6 .)

*******************************************************************************************************************
*******************************************************************************************************************
*******************************************************************************************************************
*******************************************************************************************************************
```

```
**************************************************************************************************************
*********                          The Capability Maude module                                      ********
**************************************************************************************************************
*** A capability manager is considered in this module to manage capabilities
*** at different locations ... it imports the RoleFigureModel module

(omod  Capability is
protecting RoleFigureModel .
protecting MACHINE-INT .

*** Capability Manager with available and used capability sets
class  ConfigManager | aCap : CapSet, uCap : CapSet .

*** capability management messages:
*** request to allocate capabilities
msg capAllocation-src__ : Oid CapSet -> Msg .
*** return of the allocation
msg capAllreturn-tar__ : Oid CapSet -> Msg .
*** request to release capabilities
msg capRelease-src__ : Oid CapSet -> Msg .

*** capabilities:

ops cap1-a cap1-b cap1-c cap1-d cap1-e cap1-f : -> Cap .
ops cap2-a cap2-b cap2-c cap2-d cap2-e cap2-f : -> Cap .

ops caps3 caps4 : -> CapSet .
ops configmanager, a, a', b : -> Oid .
ops testC1 testC2 testC3 testC4 testC5 testC6 testC7 : -> Configuration .
op cm : -> Type .

*** Some set-operations:
op __ : CapSet CapSet -> CapSet [assoc comm id: nilcaps prec 15] .
op in : Cap CapSet -> Bool .
op subseteq : CapSet CapSet -> Bool .
op _-_ : CapSet Cap -> CapSet .            *** Set minus one Oid
op _-_ : CapSet CapSet -> CapSet .         *** Set minus set

***the same variables in the RoleFigureModule
vars A A' A'' B B' C D : Oid . vars N N' N'' : OidSet .
vars M : Content .
vars TYPE : Type .
vars ST1 ST2 : State .
vars STS1 STS2 STS3 STS4 : StateSet .
vars BH1 BH2 : Beh .
vars BHT1 BHT2 : BehType .
vars SIG1 SIG2 : Signal .
vars SIGS1 SIGS2 : SignalSet .
vars CAP1 CAP2 CAP3 CAP4 CAP5 CAP6 : Cap .
vars CAPS1 CAPS2 CAPS3 CAPS4 : CapSet .
vars MET1 MET2 : Met .
vars METS1 METS2 : MetSet .
vars MES1 MES2 : Mes .
vars MESS1 MESS2 : MesSet .
vars LOC1 LOC2 : Loc .
vars L1 L2 : MachineInt .

*** Functions on sets:
eq CAP1 CAP1 = CAP1 .     *** This equation makes the multiset into a set.
eq in(CAP1, CAP2 CAPS1) = CAP1 == CAP2 or in(CAP1,CAPS1) .
eq in(CAP1,nilcaps) = false .
eq subseteq(CAP1 CAPS1,CAPS2) = in(CAP1,CAPS2) and subseteq(CAPS1,CAPS2) .
eq subseteq(nilcaps,CAPS2) = true .
eq (CAP1 CAPS1) - CAP1 = CAPS1 - CAP1 .
ceq CAPS1 - CAP1 = CAPS1 if not in(CAP1,CAPS1) .
ceq (CAPS1) - (CAP1 CAPS2) =  ((CAPS1 - CAP1) - (CAPS2 - CAP1)) if in(CAP1,CAPS1) .
ceq (CAPS1) - (CAP1 CAPS2) =  ((CAPS1) - (CAPS2 - CAP1)) if not in(CAP1,CAPS1) .
eq (CAPS1) - nilcaps = CAPS1 .

eq testC1 =
(capAllocation-src a cap1-a)
< configmanager : ConfigManager | aCap : cap1-a cap1-b cap1-c cap1-d cap1-e cap1-f
cap2-a cap2-b cap2-c cap2-d cap2-e cap2-f , uCap : nilcaps >
< a : RoleFigure | Location : loc1, Int : nil, Beh : bht1, Cap : caps1, Que : mess1, Met : mets1 > .

eq testC2 =
(capRelease-src a cap1-a cap1-b)
< configmanager : ConfigManager | aCap :  cap1-d cap1-e cap1-f
cap2-a cap2-b cap2-c cap2-d cap2-e cap2-f , uCap : cap1-a cap1-b cap1-c >
< a : RoleFigure | Location : loc1, Int : nil, Beh : bht1, Cap : caps1, Que : mess1, Met : mets1 > .

eq testC3 =
(pi-tar director src b name a location loc1 with bht2 cap1-a cap1-b cm)
< configmanager : ConfigManager | aCap : cap1-a cap1-b cap1-c cap1-d cap1-e cap1-f
cap2-a cap2-b cap2-c cap2-d cap2-e cap2-f , uCap : nilcaps >
< director : Director | Int : b >
< b : RoleFigure | Location : loc1, Int : director, Beh : bht1, Cap : caps1, Que : mess1, Met : mets1
> .

*** this configuration shows a competing plug in requests on shared capabilities
eq testC4 =
(pi-tar director src b name a location loc1 with bht2 cap1-a cap1-b cm)(pi-tar director src b name c
location loc1 with bht2 cap1-a cap1-c cm)
< configmanager : ConfigManager | aCap : cap1-a cap1-b cap1-c cap1-d cap1-e cap1-f
cap2-a cap2-b cap2-c cap2-d cap2-e cap2-f , uCap : nilcaps >
< director : Director | Int : b >
< b : RoleFigure | Location : loc1, Int : director, Beh : bht1, Cap : caps1, Que : mess1, Met : mets1
> .

eq testC5 =
(cc-tar b src a p cap1-a cap1-b cm)
< configmanager : ConfigManager | aCap : cap1-a cap1-b cap1-c cap1-d
cap2-a cap2-b cap2-c cap2-d , uCap : cap1-e cap1-f cap2-e cap2-f >
< director : Director | Int : b a >
< a : RoleFigure | Location : loc1, Int : director b, Beh : bht1, Cap : cap2-e cap2-f, Que : mess1,
Met : mets1 >
< b : RoleFigure | Location : loc1, Int : director a, Beh : bht1, Cap : cap1-e cap1-f, Que : mess1,
Met : mets1 > .
```

```
eq testC6 =
(po-tar b src director name b cm)
< configmanager : ConfigManager | aCap : cap1-c cap1-d cap1-e cap1-f
cap2-a cap2-b cap2-c cap2-d cap2-e cap2-f , uCap : cap1-a cap1-b  >
< director : Director | Int : b >
< b : RoleFigure | Location : loc1, Int : director, Beh : bht1, Cap : cap1-a cap1-b, Que : mess1, Met
: mets1 > .

*** capability rewriting rules

*** the configuration manager handles the allocation and release of capabilities
crl [capAllocation] :
  (capAllocation-src A CAPS3)
  < configmanager : ConfigManager | aCap : CAPS1, uCap : CAPS2 > =>
  < configmanager : ConfigManager | aCap : CAPS1 - CAPS3, uCap : CAPS2 CAPS3 >  (capAll-
return-tar A CAPS3)
  if subseteq(CAPS3,CAPS1) .

crl [capAllreturn] :
  (capRelease-src A CAPS3)
  < configmanager : ConfigManager | aCap : CAPS1, uCap : CAPS2 > =>
  < configmanager : ConfigManager | aCap : CAPS1 CAPS3, uCap : CAPS2 - CAPS3 >
  if subseteq(CAPS3,CAPS2) .

*** Plug in with consideration to capability allocation by the configuration manager
crl [PlugInwithCapBehCap] :
  (pi-tar D src A name B location LOC1 with BHT2 CAPS2 TYPE)
  < director : Director | Int : N > =>
  < director : Director | Int : B N > < B : RoleFigure | Location : LOC1, Int : director A, Beh : BHT2,
  Cap : nilcaps, Que : nilmess, Met : nilmets >
  (ci-tar A src director j B) (capAllocation-src B CAPS2)
  if (TYPE == cm) and (D == director) .

*** this shows how an existing Role-Figure adds an allocated capset to its capability definition
rl [PlugInwithCapBehCapReturn] :
  (capAllreturn-tar B CAPS2)
  < B : RoleFigure | Location : LOC1, Int : N, Beh : BHT1, Cap : CAPS1, Que : MESS1, Met :
METS1 > =>
  < B : RoleFigure | Location : LOC1, Int : N, Beh : BHT1, Cap : CAPS1 CAPS2, Que : MESS1,
Met : METS1 > .

*** Capability Change:
crl [CapabilityChangeSemanticsCap] :
  (cc-tar A src B p CAPS2 TYPE)
  < A : RoleFigure | Location : LOC1, Int : N, Beh : BHT1, Cap : CAPS1, Que : MESS1, Met :
METS1 > =>
  < A : RoleFigure | Cap : CAPS1 > (capAllocation-src A CAPS2) if (TYPE == cm) and not sub-
seteq(CAPS2,CAPS1) .

*** Plug out method with consideration to capability allocation by the configuration manager
crl [PlugOuttoActorSemanticsCap] :
  (po-tar A src B name C TYPE)
  < A : RoleFigure | Location : LOC1, Int : N, Beh : BHT1, Cap : CAPS1, Que : MESS1, Met :
METS1 > =>
  (multimsg remInt from A to N) (capRelease-src director CAPS1)
  if (TYPE == cm) and (C == A) and (B == director) and in(B,N) .

endom)

***some commands for system runs

(rew testC1 .)
(rew testC2 .)
(rew testC3 .)
(rew testC4 .)
(rew testC5 .)
(rew testC6 .)

**************************************************************************************************************
**************************************************************************************************************
**************************************************************************************************************
**************************************************************************************************************
```

```
*****************************************************************************************************************
*********                              The Mobile Maude module                                         ********
*****************************************************************************************************************

*** this imports the Capability module and the Role-Figure module
*** and executes a simple behaviour example with mobility

(omod  Mobile is
protecting RoleFigureModel .
protecting Capability .
protecting MACHINE-INT .

*** Mobile Role-Figure inheriting from the generic Role-Figure class, and it can handle the mobility
management properties
class  MRoleFigure | cSt : State, Stb : StateSet, Prg : StateSet .

subclass MRoleFigure < RoleFigure .

*** constants:

ops s1 s2 s3 s4 s5 s6 : -> State .
ops director, a, a', b : -> Oid .
ops testM1 testM2 testM3 : -> Configuration .
op mob : -> Type .
op idle : -> State .
op nilsts : -> StateSet .

*** Some set-operations:
op __ : StateSet StateSet -> StateSet [assoc comm id: nilsts prec 15] .
op in : State StateSet -> Bool .
op subseteq : StateSet StateSet -> Bool .
op _-_ : StateSet State -> StateSet .       *** Set minus one Oid!

***the same variables in the RoleFigureModule
vars A A' A'' B B' C D : Oid . vars N N' N'' : OidSet .
vars M : Content .
vars TYPE : Type .
vars ST1 ST2 : State .
vars STS1 STS2 STS3 STS4 : StateSet .
vars BH1 BH2 : Beh .
vars BHT1 BHT2 : BehType .
vars SIG1 SIG2 : Signal .
vars SIGS1 SIGS2 : SignalSet .
vars CAP1 CAP2 : Cap .
vars CAPS1 CAPS2 : CapSet .
vars MET1 MET2 : Met .
vars METS1 METS2 : MetSet .
vars MES1 MES2 : Mes .
vars MESS1 MESS2 : MesSet .
vars LOC1 LOC2 : Loc .

*** Functions on sets:
eq ST1 ST1 = ST1 .     *** This equation makes the multiset into a set.
eq in(ST1, ST2 STS1) = ST1 == ST2 or in(ST1,STS1) .
eq in(ST1,nilsts) = false .
eq subseteq(ST1 STS1 ,STS2) = in(ST1,STS2) and subseteq(STS1,STS2) .
eq subseteq(nilsts,STS2) = true .
eq (ST1 STS1) - ST1 = STS1 - ST1 .
ceq STS1 - ST1 = STS1 if not in(ST1,STS1) .

*** in the following configuration, we control the way how Maude executes the rewrite rules
eq testM1 =
(mo-tar a src director location a' loc2 with moving mob)
< configmanager : ConfigManager | aCap : cap1-c cap1-d cap1-e cap1-f
cap2-a cap2-b cap2-c cap2-d cap2-e cap2-f , uCap : cap1-a cap1-b  >
< director : Director | Int : a >
< a : MRoleFigure | Location : loc1, Int : director, Beh : bht1, Cap : cap1-a cap1-b, Que : mess1,
Met : mets1, cSt : s3, Stb : s3, Prg : s5 s6 > .

eq testM2 =
(po-tar b src director name b cm)
< configmanager : ConfigManager | aCap : cap1-c cap1-d cap1-e cap1-f
cap2-a cap2-b cap2-c cap2-d cap2-e cap2-f , uCap : cap1-a cap1-b  >
< director : Director | Int : b >
< b : MRoleFigure | Location : loc1, Int : director, Beh : bht1, Cap : cap1-a cap1-b, Que : mess1,
Met : mets1 > .

*** rules

*** A simple PlugIn, and PlugIn with cap/beh by which the director performs
crl [PlugInType] :
   (pi-tar D src A name B location LOC1 with TYPE)
   < director : Director | Int : N > =>
   < director : Director | Int : B N > < B : MRoleFigure | Location : LOC1, Int : director A, Beh :
nilbht,
   Cap : nilcaps, Que : nilmess, Met : nilmets, cSt : idle, Stb : nilsts, Prg : nilsts >
   (ci-tar A src director j B)
   if not in(B,N) and (TYPE == mob) and (D == director) .

***  plugin request for moving actors
crl [PlugInMType] :
   (pi-tar D src A name A' location LOC1 with M TYPE)
   < director : Director | Int : N > =>
   < director : Director | Int : A' N > < A' : MRoleFigure | Location : LOC1, Int : director A, Beh :
nilbht,
   Cap : nilcaps, Que : nilmess, Met : nilmets, cSt : idle, Stb : nilsts, Prg : nilsts  >
   (mor-tar A src director to A' LOC1 with TYPE)
   if not in(A',N) and (M == moving) and (TYPE == mob) and (D == director) .

*** this is an extended BehaviourChange rule
*** Behaviour Change:
crl [BehaviourChangeType] :
   (bc-tar A src B beh BHT2 ST1 STS1 STS2)
   < A : MRoleFigure | Location : LOC1, Int : N, Beh : BHT1, Cap : CAPS1, Que : MESS1, Met :
METS1, cSt : ST2, Stb : STS3, Prg : STS4 > =>
   < A : MRoleFigure | Beh : BHT2, cSt : ST1, Stb : STS1, Prg : STS2 > if in(B,N) .

*** in these two rules we check if the current state is in the set Stable states to control the move
```

```
crl [ActorMoveType] :
   (mo-tar A src B location A' LOC2 with M TYPE)
   < A : MRoleFigure | Location : LOC1, Int : N, Beh : BHT1, Cap : CAPS1, Que : MESS1, Met :
METS1,
   cSt : ST1, Stb : STS1, Prg : STS2 > =>
   < A : MRoleFigure | Int : N > (pi-tar director src A name A' location LOC2 with M mob)
   if in(ST1,STS1) and in(B,N) .

crl [ActorMoveReturnType] :
   (mor-tar A src D to A' LOC2 with TYPE)
   < A : MRoleFigure | Location : LOC1, Int : N, Beh : BHT1, Cap : CAPS1, Que : MESS1, Met :
METS1,
   cSt : ST1, Stb : STS1, Prg : STS2 > =>
   < A : MRoleFigure | Int : A' N >
   (cc-tar A' src A p CAPS1 TYPE) (multimsg creInt from A' to N) (bc-tar A' src A beh BHT1 ST1
STS1 STS2)
   (po-tar director src A' name A TYPE)
   if in(ST1,STS1) and (TYPE == mob) and (D == director) .
*** this may correspond to an existing Role-Figure instance
crl [PlugInwithCapBehCap] :
   (pi-tar D src A name B location LOC1 with BHT2 CAPS2 TYPE)
   < director : Director | Int : N > =>
   < director : Director | Int : B N > < B : MRoleFigure | Location : LOC1, Int : director A, Beh :
BHT2,
   Cap : nilcaps, Que : nilmess, Met : nilmets, cSt : idle, Stb : nilsts, Prg : nilsts >
   (ci-tar A src director j B) (capAllocation-src B CAPS2)
   if (TYPE == mob) and (D == director) .

*** Capability Change:
crl [CapabilityChangeSemanticsCap] :
   (cc-tar A src B p CAPS2 TYPE)
   < A : MRoleFigure | Location : LOC1, Int : N, Beh : BHT1, Cap : CAPS1, Que : MESS1, Met :
METS1 > =>
   < A : MRoleFigure | Cap : CAPS1 > (capAllocation-src A CAPS2)
   if (TYPE == mob) and not subseteq(CAPS2,CAPS1) .

*** A simple PlugOut method request by which the director is performing the plugout process
crl [PlugOuttoActorSemanticsCap] :
   (po-tar A src B name C TYPE)
   < A : MRoleFigure | Location : LOC1, Int : N, Beh : BHT1, Cap : CAPS1, Que : MESS1, Met :
METS1 > =>
   (multimsg remInt from A to N) (capRelease-src director CAPS1)
   if (TYPE == mob) and (C == A) and (B == director) and in(B,N) .
endom)

***some commands for system runs
 (rew testM1 .)
(rew testM2 .)
**********************************************************************************************************
**********************************************************************************************************
**********************************************************************************************************
**********************************************************************************************************
```

```
***********************************************************************************************************
*********                    The Client/Server Maude module                    ********
***********************************************************************************************************
*** It first defines ActorModel module, which is a subset of the semantics of the RoleFigureModel
*** it then specifies a simple Client/Server service system

(omod ActorModel is
sort Type .
sorts OidSet Content State StateSet Beh BehType Signal SignalSet Cap CapSet Mes MesSet Met
MetSet .
subsort Oid < OidSet .
subsort State < StateSet .
subsort Signal < SignalSet .
subsort Cap < CapSet .
subsort Met < MetSet .
subsort Mes < MesSet .

*** defining a generic actor instead of a Role-Figure, no location is applied here
class  Actor | Int : OidSet, Beh : BehType, Cap : CapSet,
            Que : MesSet, Met : MetSet .
class  Director | Int : OidSet .

*** applying similar messages to TAPAS messages:
msg ActorPlugIn_from_ : Oid Oid -> Msg .
msg ActorPlugIn_from_with_ : Oid Oid Content -> Msg .
msg ActorPlugIn_from_with_ : Oid Oid Type -> Msg .
msg ActorPlugIn_from_with__ : Oid Oid Content Type -> Msg .
msg ActorPlugIn_from_with___ : Oid Oid BehType CapSet Type -> Msg .
msg ActorPlugIn_from_with__ : Oid Oid BehType CapSet -> Msg .
msg ActorPlugOut_to_from_ : Oid Oid Oid -> Msg .
msg CreateInterface_at_ : Oid Oid -> Msg .
msg BehaviourChange_with_ : Oid BehType -> Msg .
msg CapabilityChange_with_ : Oid CapSet -> Msg .
msg ActorMove_to_with_ : Oid Oid Content -> Msg .
msg ActorMove_to_with__ : Oid Oid Content Type -> Msg .
msg ActorMoveReturn_to_ : Oid Oid -> Msg .
msg ActorMoveReturn_to_with_ : Oid Oid Type -> Msg .

*** general messages:
msg msg_from_to_ : Content Oid Oid -> Msg .

*** multimessage-declaration:
op multimsg_from_to_ : Content Oid OidSet -> Configuration .

*** constant declaration:
ops st1 st2 : -> State .
ops sts1 sts2 : -> StateSet .
ops bh1 bh2 : -> Beh .
ops m remInt creInt moving : -> Content .
ops director a a' b c e : -> Oid .                    *** Actors
ops n n' : -> OidSet .
ops bht1 bht2 : -> BehType .
ops sig1 sig2 : -> Signal .
ops sigs1 sigs2 : -> SignalSet .
ops cap1 cap2 : -> Cap .
ops caps1 caps2 : -> CapSet .
ops met1 met2 : -> Met .
ops mets1 mets2 : -> MetSet .
ops mes1 mes2 : -> Mes .
ops mess1 mess2 : -> MesSet .

*** [multi]set constructors:
op nil : -> OidSet .
op nilOid : -> Oid .
op nilcaps : -> CapSet .
op nilbht : -> BehType .
op nilmess : -> MesSet .
op nilmets : -> MetSet .

op __ : OidSet OidSet -> OidSet [assoc comm id: nil prec 15] .

*** set operations:
op in : Oid OidSet -> Bool .
op subseteq : OidSet OidSet -> Bool .
op _-_ : OidSet Oid -> OidSet .       *** Set minus one Oid!

*** variable declaration
vars A A' B C D : Oid .
vars N N' N'' : OidSet .
vars M : Content .
vars ST1 ST2 : State .
vars STS1 STS2 : StateSet .
vars BH1 BH2 : Beh .
vars BHT1 BHT2 : BehType .
vars SIG1 SIG2 : Signal .
vars SIGS1 SIGS2 : SignalSet .
vars CAP1 CAP2 : Cap .
vars CAPS1 CAPS2 : CapSet .
vars MET1 MET2 : Met .
vars METS1 METS2 : MetSet .
vars MES1 MES2 : Mes .
vars MESS1 MESS2 : MesSet .

*** Functions on sets:
eq A A = A .
eq in(A, B N) = A == B or in(A,N) .
eq in(A,nil) = false .
eq subseteq(A N ,N') = in(A,N') and subseteq(N,N') .
eq subseteq(nil,N') = true .
eq (A N) - A = N - A .
ceq N - A = N if not in(A,N) .

*** multimessage definition:
ceq multimsg M from A to (B N) =
        (msg M from A to B) (multimsg M from A to (N - B)) if not M == creInt .
eq multimsg creInt from A to (B N) =
        (CreateInterface A at B) (CreateInterface B at A) (multimsg creInt from A to (N - B)) .
eq multimsg M from A to nil = none .
```

```
*** rewriting rules: applying similar rules to the RoleFigureModel rules
crl [PlugOuttoActor] :
  (ActorPlugOut A to B from C)
  < A : Actor | Int : N, Beh : BHT1, Cap : CAPS1, Que : MESS1, Met : METS1 > =>
  (multimsg remInt from A to N) if (B == A) and (C == director) .

crl [PlugOuttoDirector] :
  (ActorPlugOut A to director from B)
  < director : Director | Int : N > =>
  < director : Director | Int : N > (ActorPlugOut A to A from director) if in(A,N) .

crl [RemoveInterfacesAtActor] :
  (msg remInt from B to A)
  < A : Actor | Int : N > =>
  < A : Actor | Int : N - B >  if in(B,N) .

crl [RemoveInterfacesAtDirector] :
  (msg remInt from B to A)
  < A : Director | Int : N > =>
  < A : Director | Int : N - B >  if in(B,N) .

crl [PlugInNotMoving] :
  (ActorPlugIn D from B with M)
  < director : Director | Int : N > =>
  < director : Director | Int : D N > < D : Actor | Int : director B, Beh : nilbht, Cap : nilcaps, Que :
nilmess, Met : nilmets >
  (CreateInterface D at B) if not in(D,N) and (M =/= moving) .

crl [PlugInNoContent] :
  (ActorPlugIn D from B)
  < director : Director | Int : N > =>
  < director : Director | Int : D N > < D : Actor | Int : director B, Beh : nilbht, Cap : nilcaps, Que :
nilmess, Met : nilmets >
  (CreateInterface D at B) if not in(D,N) .

crl [PlugInMoving] :
  (ActorPlugIn D from B with M)
  < director : Director | Int : N > =>
  < director : Director | Int : D N >
  < D : Actor | Int : director B, Beh : nilbht, Cap : nilcaps, Que : nilmess, Met : nilmets >
  (ActorMoveReturn D to B)
  if not in(D,N) and (M == moving) .

rl [PlugInwithCapBeh] :
  (ActorPlugIn D from B with BHT2 CAPS2)
  < director : Director | Int : N > =>
  < director : Director | Int : D N > < D : Actor | Int : director B, Beh : BHT2, Cap : CAPS2, Que :
nilmess, Met : nilmets >
  (CreateInterface D at B) .

rl [CreateInterfaceAtActor] :
  (CreateInterface B at A)
  < A : Actor | Int : N, Beh : BHT1, Cap : CAPS1, Que : MESS1, Met : METS1 > =>
  < A : Actor | Int : B N > .

rl [CreateInterfaceAtDirector] :
  (CreateInterface B at A)
  < A : Director | Int : N > =>
  < A : Director | Int : B N > .

rl [BehaviourChange] :
  (BehaviourChange A with BHT2)
  < A : Actor | Int : N, Beh : BHT1, Cap : CAPS1, Que : MESS1, Met : METS1 > =>
  < A : Actor | Beh : BHT2 > .

rl [CapabilityChange] :
  (CapabilityChange A with CAPS2)
  < A : Actor | Int : N, Beh : BHT1, Cap : CAPS1, Que : MESS1, Met : METS1 > =>
  < A : Actor | Cap : CAPS2 > .

rl [ActorMove] :
  (ActorMove A to A' with M)
  < A : Actor | Int : N, Beh : BHT1, Cap : CAPS1, Que : MESS1, Met : METS1 > =>
  < A : Actor | Int : N > (ActorPlugIn A' from A with M) .

rl [ActorMoveReturn] :
  (ActorMoveReturn A' to A)
  < A : Actor | Int : N, Beh : BHT1, Cap : CAPS1, Que : MESS1, Met : METS1 > =>
  < A : Actor | Int : A' N >
  (CapabilityChange A' with CAPS1) (multimsg creInt from A' to N) (BehaviourChange A' with
BHT1) (ActorPlugOut A to director from A') .

endom)

*** here we define the Client/Server module that imports the ActorModel

(omod  ClientServer is
protecting ActorModel .
protecting MACHINE-INT .

sorts Status Client Server Cong congectionValue .
***capability definition: Congestion
subsort Cong < Cap < CapSet .

***Actor types may be used for naming later
subsort Client < Type .
subsort Server < Type .

***Application Role-Figures inheriting from the generic actor class
class  ActorClient | Count : MachineInt, Backup : Oid .
class  ActorServer | Trans : MachineInt .

subclass ActorServer < Actor .
subclass ActorClient < Actor .

*** signal definitions
```

```
msg s1_from_to_ : Status Oid Oid -> Msg .
msg s1_from_to_myBU_ : Status Oid Oid Oid -> Msg .
msg s2from_to_ : Oid Oid -> Msg .
msg s3from_to_ : Oid Oid -> Msg .
msg s4 : -> Msg .

***these capability instances show the network congestion condition
op Congested : -> Cong .
op Ncongested : -> Cong .
op congestionValue : -> MachineInt .

***parametr for acknowledgement
op Ok : -> Status .
op Nok : -> Status .

op cl : -> Type .
op sr : -> Type .

***operation to return the type of certain actors, to be used later
op ActorType : Oid -> Type .

ops director, a, a', b : -> Oid .

var stus : Status .
var CONG :  Cong .
var COUNT : MachineInt .
var TRANS : MachineInt .
ops cap1 cap2 : -> Cap .

***the same variables in the
vars A A' A'' B B' C D : Oid . vars N N' N'' : OidSet .
vars M : Content .
vars TYPE : Type .
vars ST1 ST2 : State .
vars STS1 STS2 : StateSet .
vars BH1 BH2 : Beh .
vars BHT1 BHT2 : BehType .
vars SIG1 SIG2 : Signal .
vars SIGS1 SIGS2 : SignalSet .
vars CAP1 CAP2 : Cap .
vars CAPS1 CAPS2 : CapSet .
vars MET1 MET2 : Met .
vars METS1 METS2 : MetSet .
vars MES1 MES2 : Mes .
vars MESS1 MESS2 : MesSet .

ops testCS1 testCS2 testCS21 testCS3 : -> Configuration .

eq ActorType(a) = cl .
eq ActorType(b) = sr .
eq congestionValue = 3 .     *** this sets the value for congested network

*** system runs to test these configurations
eq testCS1 =
  (s2from b to a)
  < director : Director | Int : a b >
  < a : ActorClient | Int : director b , Beh : bht1, Cap : Ncongested, Que : mess1, Met : mets1,
Count : 0, Backup : a' >
  < b : ActorServer | Int : director a , Beh : bht2, Cap : caps2, Que : mess2, Met : mets2, Trans : 1
> .

eq testCS2 =
  (s2from b to a)
  < director : Director | Int : a b >
  < a : ActorClient | Int : director b , Beh : bht1, Cap : Ncongested, Que : mess1, Met : mets1,
Count : 0, Backup : a' >
  < b : ActorServer | Int : director a , Beh : bht2, Cap : caps2, Que : mess2, Met : mets2, Trans : 5
> .

*** a moving is deterministic to one backup, which if congested doesn't know where to move
eq testCS3 =
  (s2from b to a)
  < director : Director | Int : a b >
  < a : ActorClient | Int : director b , Beh : bht1, Cap : Ncongested, Que : mess1, Met : mets1,
Count : 0, Backup : a' >
  < b : ActorServer | Int : director a , Beh : bht2, Cap : caps2, Que : mess2, Met : mets2, Trans :
10 > .


*** rules
*** a client receiving a signal from the server
crl [receiveNotcongested] :
  (s2from B to A)
  < A : ActorClient | Int : N, Beh : BHT1, Cap : CONG, Que : MESS1, Met : METS1, Count :
COUNT, Backup : A' > =>
  < A : ActorClient | Cap : Ncongested, Count : COUNT + 1 > (s1 Ok from A to B)
  if (COUNT <= congestionValue) .

crl [receiveCongested] :
  (s2from B to A)
  < A : ActorClient | Int : N, Beh : BHT1, Cap : CONG, Que : MESS1, Met : METS1, Count :
COUNT, Backup : A' > =>
  < A : ActorClient | Cap : Congested, Count : 0 >
  (s1 Nok from A to B myBU A')
  if (COUNT > congestionValue) .

*** if there is no Backup for the client Role-Figure
crl [receiveCongestedNoBackup] :
  (s2from B to A)
  < A : ActorClient | Int : N, Beh : BHT1, Cap : CONG, Que : MESS1, Met : METS1, Count :
COUNT, Backup : A' > =>
  < A : ActorClient | Cap : Congested, Count : 0 >
  (s1 Nok from A to B )
  if (COUNT > congestionValue) and (A' == nilOid) .

*** a client receives the end of transmission
 rl [Finish] :
  (s3from B to A)
```

```
   < A : ActorClient | Int : N, Beh : BHT1, Cap : CONG, Que : MESS1, Met : METS1, Count :
COUNT, Backup : A' >
   => < A : ActorClient | Count : 0 > .

*** the server receives an acknowledgment from the client with status of the network congestion
crl [adapt1] :
   (s1 stus from B to A)
   < A : ActorServer | Int : N, Beh : BHT1, Cap : CAPS1, Que : MESS1, Met : METS1, Trans :
TRANS >
   => < A : ActorServer | Trans : 0 > (s3from A to B)
   if (TRANS == 0) .

*** if the sending is to continue normally
crl [adapt2] :
   (s1 stus from B to A)
   < A : ActorServer | Int : N, Beh : BHT1, Cap : CAPS1, Que : MESS1, Met : METS1, Trans :
TRANS >
   => < A : ActorServer | Trans : TRANS - 1 > (s2from A to B)
   if (stus == Ok) and (TRANS =/= 0) .

*** the Role-Figure move here will create a general Role-Figure class, and not an ActorClient
crl [adapt3] :
   (s1 stus from B to A myBU B')
   < A : ActorServer | Int : N, Beh : BHT1, Cap : CAPS1, Que : MESS1, Met : METS1, Trans :
TRANS >
   =>  < A : ActorServer | Trans : TRANS - 1 >
   (ActorMove B to B' with moving cl) (s2from A to B')
   if (stus =/= Ok) and (TRANS =/= 0) .

*** this is executed if there is no Backup for the congested Role-Figure
 crl [adapt3nobackup] :
   (s1 stus from B to A)
   < A : ActorServer | Int : N, Beh : BHT1, Cap : CAPS1, Que : MESS1, Met : METS1, Trans :
TRANS >
   => < A : ActorServer | Trans : TRANS - 1 > (ActorPlugOut B to director from A) (ActorPlugIn B
from A )
   if (stus == Nok) and (TRANS =/= 0) .


*** A simple PlugIn, and PlugIn with cap/beh by which the director performs
crl [PlugInType] :
   (ActorPlugIn D from B with TYPE)
   < director : Director | Int : N > =>
   < director : Director | Int : D N > < D : ActorClient | Int : director B, Beh : nilbht,
   Cap : nilcaps, Que : nilmess, Met : nilmets, Count : 0, Backup : nilOid >
   (CreateInterface D at B)
   if not in(D,N) and (TYPE == cl) .

***  plugin request for moving Role-Figures
crl [PlugInMType] :
   (ActorPlugIn D from B with M TYPE)
   < director : Director | Int : N > =>
   < director : Director | Int : D N > < D : ActorClient | Int : director B, Beh : nilbht,
   Cap : nilcaps, Que : nilmess, Met : nilmets, Count : 0, Backup : nilOid >
```

```
  (ActorMoveReturn D to B with TYPE)
  if not in(D,N) and (M == moving) and (TYPE == cl) .

*** this may correspond to an existing Role-Figure instance
crl [PlugInwithCapBehType] :
  (ActorPlugIn D from B with BHT2 CAPS2 TYPE)
  < director : Director | Int : N > =>
  < director : Director | Int : D N > < D : ActorClient | Int : director B, Beh : BHT2,
  Cap : CAPS2, Que : nilmess, Met : nilmets, Count : 0, Backup : nilOid >
  (CreateInterface D at B)
  if (TYPE == cl) .

crl [ActorMoveType] :
  (ActorMove A to A'' with M TYPE)
  < A : ActorClient | Int : N, Beh : BHT1, Cap : CAPS1, Que : MESS1, Met : METS1,
  Count : COUNT, Backup : A' > =>
  < A : ActorClient | Int : N > (ActorPlugIn A'' from A with M TYPE)
  if (A'' == A') .

crl [ActorMoveReturnType] :
  (ActorMoveReturn A'' to A with TYPE)
  < A : ActorClient | Int : N, Beh : BHT1, Cap : CAPS1, Que : MESS1, Met : METS1,
  Count : COUNT, Backup : A' > =>
  < A : ActorClient | Int : A' N >
  (CapabilityChange A' with CAPS1) (multimsg creInt from A' to N) (BehaviourChange A' with
BHT1)
  (ActorPlugOut A to director from A')
  if (A'' == A') and (TYPE == cl) .

endom)

***some commands for system runs
(rew testCS1 .)
(rew testCS2 .)
(rew testCS3 .)

****************************************************************************************************************
****************************************************************************************************************
****************************************************************************************************************
****************************************************************************************************************
```