DYNAMIC ROLE BINDING IN A SERVICE ORIENTED ARCHITECTURE

Humberto Nicolás Castejón and Rolv Bræk
NTNU, Department of Telematics, N-7491 Trondheim, Norway
{humberto.castejon, rolv.braek}@item.ntnu.no

Abstract Many services are provided by a structure of service components that are dynamically bound and performed by system components. Service modularity requires that service components can be developed separately, deployed dynamically and then used to provide situated services without undesirable service interactions. In this paper we introduce a two-dimensional approach where service components are roles defined using UML 2.0 collaborations and system components are agents representing domain entities such as users and terminals. The process of dynamic role binding takes place during service execution and provides general mechanisms to handle context dependency, personalisation, resource limitations and compatibility validation. A policy framework for these mechanisms is outlined.

1. Introduction

A *service* may generally be defined as an identified partial functionality provided by a system to an end user, such as a person or other system. The most general form of service involves several system components collaborating on an equal basis to provide the service to one or more users. This understanding of service is quite general and covers both client-server and peer-to-peer services as described in [6]. A common trait of many services is that the structure of collaborating components is dynamic. Links between components are created and deleted dynamically and many services and service features depend on whether the link can be established or not, and define what to do if it cannot be established (e.g. busy treatment in a telephone call). Indeed, setting up links is the goal of some services. For example, the goal of a telephone call is to establish a link between two system components, so that the users they represent can talk to each other. In the past this problem has often been addressed in service specific ways. It may however, be generalised to a problem of *dynamic role binding*, i.e. requesting system components to play roles, such as for example requesting a `UserAgent` to play the `b`-subscriber in a call. The response to such a request may be to alert the end-user (if free and available), to reject the call, to forward it or to provide some waiting functionality (if busy). Which feature to select depends on what is subscribed, what other features are active, what resources are available, what the current context is and what the preferences of the user are. By recognising dynamic role binding as a general problem, we believe it is possible to find generic and service independent solutions. In fact, many crucial mechanisms can be associated with dynamic role binding: service discovery; feature negotiation and selection; context dependency resolution; compatibility validation of collaborating service components, and dependency resolution.

Modularity is a well-known approach for easing service development. Service modularity requires a separation of service components from system components, allowing the former ones to be specified and designed separately from the latter ones, then be incrementally deployed and finally be linked dynamically during service execution to provide actual services without undesirable service interactions. We will show that dynamic role binding mechanisms are crucial to achieve the desired separation and modularity and still be able to manage the complex mutual dependencies between service components and system components. It is desirable that such dependencies are not hard coded, but represented by information that can be easily configured and interpreted by general mechanisms, i.e. by some kind of policies.

In this paper we present a service architecture where service modularity and dynamic linking is supported by means of roles and general mechanisms for dynamic role binding. In Sect. 2 the main elements of the architecture are presented: agents mirroring the environment as system components; UML 2.0 collaborations and collaboration roles as service-modelling elements; and UML active classes as service components. In Sect. 3 we show how the proposed architecture provides structure to service-execution policies and how dynamic role binding enables policy-driven feature selection with compatibility guarantees. Finally we conclude with a summary of the presented work.

2. Agent and Role Based Service Architecture

Fig. 1 suggests an architecture for service-oriented systems which is characterized by horizontal and vertical composition. On the horizontal axis, system components are identified that may reside in different computing environments. This axis reflects domain entities (such as users,

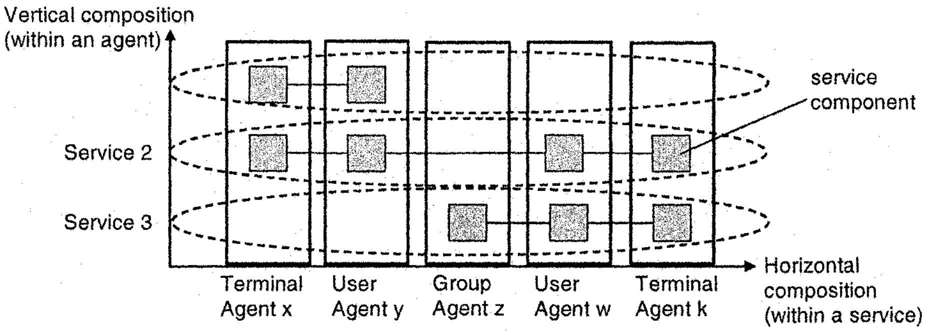

Figure 1. Service Oriented Architecture

user communities and terminals) and resources that must be represented in a service providing system regardless of what services it provides. On the vertical axis, several services and service components are identified that depend on the system components of the architecture. This two-dimensional picture illustrates the crosscutting nature of services which is a well-known challenge in service engineering [4, 8][9, 2].

In the following sections we will present our particular realization of this architecture.

2.1 Agents as System Components

In [6], Bræk and Floch identify two principal system architectures: the *agent oriented* and the *server oriented.* The agent oriented architecture follows the principle that a system should be structured to mirror objects in the domain and environment it serves [3]. This is a general principle known to give stable and adaptable designs. Agents may represent and have clear responsibilities for serving domain/environment entities and resources and thereby provide a single place to resolve dependencies. In the case of personalised communication services accessible over a number of different terminal types, this mirroring leads to a structure of `TerminalAgents` and `UserAgents` as illustrated in Fig. 1. In addition there may be agents corresponding to user communities (e.g. the `GroupAgent` in Fig. 1), service enablers and shared service functionality. Several authors have proposed similar architectures, for example [21] and [1].

Note that such an agent structure reflects properties of the domain being served and not particular implementation details, nor particular services. It is therefore quite stable and service independent. At the

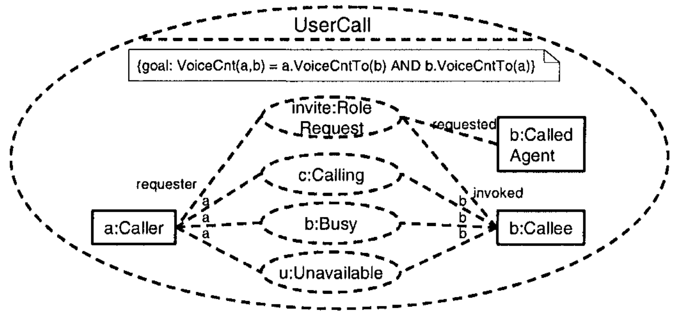

Figure 2. The UserCall Service as a UML 2.0 Collaboration

same time agents provide natural containers for properties and policies of domain entities like users, terminals and service enablers.

2.2 UML Collaborations as Services and Roles as Service Components

As illustrated in Fig. 1, a service is a partial functionality provided by a collaboration between service components executed by agents to achieve a desired goal for the end users. UML 2.0 *collaborations* [13] are well suited for service modelling as they are intended to describe partial functionalities provided by collaborating roles played by objects. An interesting characteristic of UML collaborations is that they can be applied as *collaboration uses* and employed as components in the definition of larger collaborations. This feature enables a compositional and incremental design of services, as we explain in [17, 7]. For instance, Fig. 2 shows a collaboration specifying a `UserCall` service in terms of collaboration roles linked by collaboration uses, which correspond to different phases and features of the service. It also shows a simple goal expression representing a desired goal state for the collaboration. Behaviour descriptions can be associated with the collaboration to precisely define the service behaviour, including precise definitions of the visible interface behaviour that objects must show in order to participate in the collaboration. Collaborations thereby provide a mechanism to define *semantic interfaces* that can be used for service discovery and to ensure compatibility with respect to safety and liveness (i.e. reaching the desired goal states) when linking service components, as we discuss in [18].

Ideally, service models should be independent of particular system structures. It can be argued, however, that it is necessary and beneficial to take a minimum of architectural aspects into account [20]. The challenge is to do this at an abstraction level that fits the nature of the services without unduly binding design solutions and implementations. In our architecture (see Fig. 1) the horizontal axis represents the agent structure and the vertical axis represents the services modelled as collaborations with roles that are bound to agents. The service-independent agent structure is therefore instrumental, since it helps to identify and shape roles, without introducing undue bindings to implementation details. At the same time it provides an architectural framework for role composition, role binding and role execution.

In this service oriented architecture, service specific behaviour is the responsibility of (service-) roles while domain specific behaviour and policies are the responsibility of agents. Interactions between roles and agents are needed primarily in the process of creating and releasing dynamic links, that is, the process of *dynamic role binding.*

Roles need to be mapped to well defined service components, which can then be deployed and composed in agents to provide (new) services without causing safety or liveness problems. We do this by defining service components as UML active classes with behaviour defined by state machines. Note that service components may implement one or more UML collaboration roles composed by means of collaboration uses, as it is roughly illustrated in Figs. 2 and 4. Each of these collaboration roles will correspond to a different service or service feature. We assume that service components are typed with semantic interfaces with well defined feature sets. This information is exploited when service components are dynamically linked within a service collaboration in order to ensure their compatibility in terms of safety and liveness criteria, as explained in [18]. In addition, it is necessary to ensure that the service components can actually be bound to the intended agents. Their feature sets may also be restricted and dynamically selected during the binding process. These aspects will be discussed in the following sections.

Note that, for the sake of simplicity, in the rest of the paper we will use the word *role* to name service components.

2.3 Dynamic Role Binding

Dynamic Role binding has three distinct phases:

1 *Agent identification,* which aims at identifying an agent by consulting a nameserver or performing a service discovery. Some service features are related to the agent identification, e.g. aliasing, business domain restrictions or originating and terminating screening features in telephony.

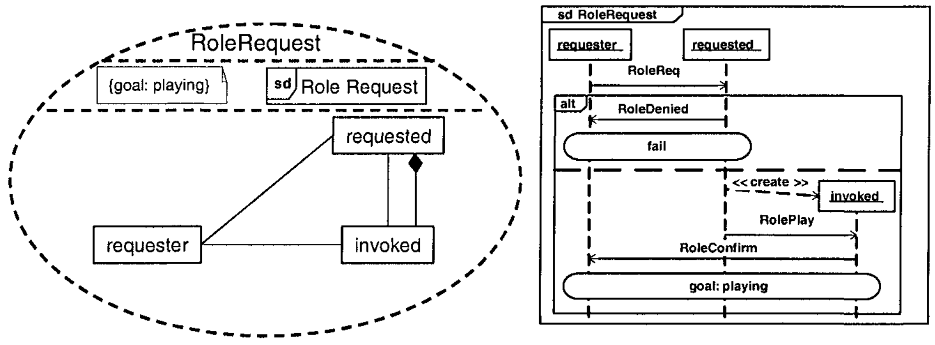

Figure 3. Role Request Pattern

2 *Role request,* which aims at creating a dynamic link according to a semantic interface with an agreed feature set. This means to request the agent identified in phase 1 to play a role with a certain feature set, which can be negotiated. The role with the agreed feature set is finally invoked. The role request pattern [5] described in Fig. 3 provides one partial solution to this. Using this protocol any role can request an agent to play a complimentary role. If the agent is able to play the requested role, it invokes it and a link is dynamically established between the requesting and the requested roles, so that they can collaborate. This is illustrated in Fig. 2, where a `UserAgent` is requested (invited) to play the `b` role in a `UserCall` collaboration. The response of the `UserAgent` in this case is to enable one of three features, represented as collaboration uses: `Calling`, `Busy` or `Unavailable`.

3 *Role release,* which signals that a role is finished and has released whatever resources it had occupied.

Once a role is invoked it can proceed autonomously, until it reaches a state where interaction with agents is again required for one of the following reasons:

- it needs to bring a new role into the collaboration (i.e. create another dynamic link). In this case it first needs to identify the agent that should play the role and then initiate a role request to this agent, as explained above;
- it needs to check what feature or feature set to select at a certain point in its behaviour, if this depend on agent policy (e.g. if mid-call telephony services are allowed);

- it needs to signal to its own agent that it is available for additional linking, in response to an incoming role request (e.g. to perform a call waiting feature); or

- it is finished and performs role release.

Note that all cases except the second are related to dynamic role binding. Also note that a large proportion of features are discovered, selected and initiated in connection with dynamic role binding.

3. Governing Service Execution with Policies

Dynamic role binding is clearly a very central mechanism in service execution. As we have already pointed out, associated with dynamic role binding are such key issues as: service discovery, feature negotiation and selection, context dependency resolution, compatibility validation and feature interaction detection/avoidance. The challenge is to find general, scalable and adaptable ways of dealing with those issues.

In general a role can only be bound to an agent without undesired side-effects if certain (pre-/post-) conditions hold. By explicitly expressing these conditions as constraints, we may check them upon role-binding and only allow the role to be invoked if they are satisfied. It is also important to give users the possibility to express their preferences to control the selection of features when, for example, the requested service can not be delivered. In the following we will use the term *policy* to cover both general role binding constraints and user preferences. In doing that we adhere to the usual definition of policy that can be found in the computer-science literature: *a rule or information that modifies or defines a choice in the behavior of a system* [11].

The agent architecture discussed in the previous section provides a natural way to structure policies into three groups:

- *Role-binding* policies, which constrain the binding of roles to agents at run-time.

- *Collaboration* policies, which express constraints that must hold for a collaboration (i.e. a service) as a whole when it is executed. They aim at preventing actions that may compromise the intentions and goals of the collaboration.

- *Feature-selection* policies, which control the triggering of context-dependent service features.

We will take a closer look at each of these policies in the following.

3.1 Role-binding and Collaboration Policies

Role-binding policies represent conditions that must be satisfied for a role to be bound to an agent. These policies may be associated with agent types, so they shall hold for all instances of that type, or they may be defined for specific agent instances (usually describing user preferences and/or user permissions). Finally, role-binding policies may also be associated with role types and they shall hold for all instances of that role type.

The role-binding policies associated with a role type define constraints that the role imposes on any agent it may be bound to and, thereby, indirectly on system resources. For example, the role-binding policy of a role may require that role to be bound to a `TerminalAgent` representing a specific terminal type with specific capabilities (e.g. a PDA).

The role-binding policies associated with an agent represent, on the contrary, constraints that the agent imposes on the roles it can play. When these policies are associated with an agent type, they represent constraints on the type and multiplicity of the roles that can be bound to the agent, as well as other constraints imposed by the service provider (e.g. that the user must hold a valid subscription to play a certain role). When they are associated with a particular agent instance, they represent user preferences and/or user permissions specifying when that particular agent should or should not play a certain role. These preferences/permissions can be seen to express context dependency (e.g. on location, calendar, presence or availability). For example, a user may define a role-binding policy for her `UserAgent` to express that it should only participate in a `UserCall` service, playing the `callee` role, if the invitation was received between 8 am and 11 pm.

Collaboration policies express constraints that must hold for a collaboration (i.e. a service) as a whole when it occurs. We may associate collaboration policies with a UML collaboration, so that they shall hold for all occurrences of that collaboration. For instance, a collaboration policy may be associated with a conference-call collaboration to prohibit the agent playing the conference-controller role to temporally interrupt its participation in the service. This policy would specifically prohibit the conference-controller role to invoke the *hold* feature. Agent instances may also hold specialised collaboration policies that, in this case, shall only be satisfied for those occurrences of the collaboration where the agent participates. These policies may then represent user preferences. An important use of such user-defined collaboration policies is to constrain who participates in a service session. For personal communication services the identity of the agents participating in a service is important (e.g. a calling user wants a specific user's UserAgent to play the b role - see Fig. 2). It is not only important who must be invited to a service, but also who cannot be invited. Some users may not want to talk to certain people, or they may not like, for example, to talk to a machine. We can easily solve this problem using collaboration policies by constraining the type and/or id of agents that can participate in a service session, as well as the roles they can play. For instance, if a user does not want to be redirected to an automatic-response machine, she may define a collaboration policy for the UserCall service constraining the participation of IVRAgents[1]. This policy would be held by her UserAgent, thus only affecting UserCall services in which she may participate.

Role-binding and collaboration policies are checked upon a role request by both the requester and the requested agents. The requester agent checks the collaboration policies associated to the service being (or to be) executed before the role request is sent. This is done to confirm that inviting the requested agent would not violate those policies (e.g. that a UserAgent representing an undesired user would not be invited when performing a forwarding). At the reception of the role request, the requested agent checks first the collaboration policies for conformance on joining the collaboration (e.g. to ensure that all other participating agents are welcome). Thereafter, it checks the role-binding policies concerning the requested role, which is only bound if those policies are satisfied.

Policies defined for a collaboration (and its roles) are "inherited" when that collaboration is employed, as a collaboration use, in the specification of other collaborations. Therefore, when a collaboration is bound to a set of agents for its execution, all policies defined for the collaboration itself and for its sub-collaborations must hold. This is illustrated in Fig. 4. The upper part shows a collaboration specifying a FullCall service. This service is a collaboration between four roles and it is composed from the UserCall service described in Fig. 2 and two uses of a TermCall service, which specifies the collaboration between UserAgents and TerminalAgents. Policies have been defined for each of the roles and collaboration uses in FullCall (P2-P8). In addition a policy (P1) has been defined for the FullCall collaboration as a whole. The lower part of the figure illustrates a set of UserAgents and TerminalAgents, representing users and terminals, performing the FullCall service. It is important to note that for this collaboration use (i.e. fcx:FullCall) all and each of the policies defined for the FullCall collaboration must hold. This is indicated by the annotation P1+P2+...+P8. Moreover, each agent holds a set of role-binding, collaboration and feature selection policies, called P10-P13 in the figure, that also must hold in the execution

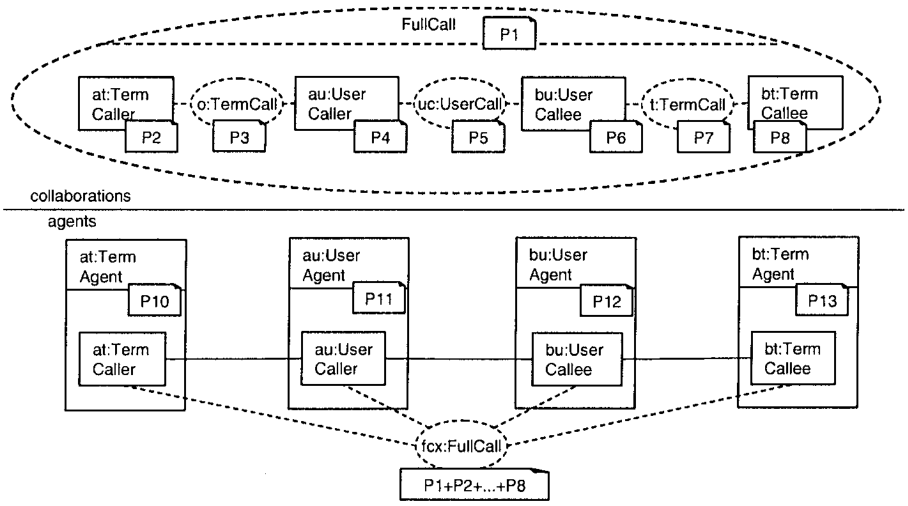

Figure 4. Role Binding

of `fcx:FullCall`. Therefore, if for example `at:TermCaller` had some collaboration policy that would not hold when inviting `bt:TermCallee`, `fcx:FullCall` could not be completed.

3.2 Feature Selection Policies

A feature can be defined as a unit of functionality in a base service. In general, we can differentiate two types of features, depending on how they are selected and triggered:

- features that are triggered within a role as part of its behaviour (e.g. call-transfer)
- features that are triggered upon role-binding depending on the agent's context and policy (e.g. call-forward on busy subscriber)

We refer to the first type of features as *mid-role-triggered* (or just *mid-role*), and to the second as *role-binding-triggered.* Mid-role-triggered features are selected as part of the role behaviour. If they may be disabled by policy decisions this should be agreed during a negotiation phase before the role is bound. Alternatively, the role may consult its containing agent concerning actual policies before invoking mid-role features. One example of such a feature is call-forward on no-answer. It describes an alternative behavior to the `UserCall` service and it is triggered when the latter does not achieve its goal (i.e. contacting the end-user).

Role-binding-triggered feature selection occurs when an agent receives a role request. In that case the agent (1) checks its feature selection policies to determine if, in the current context, there is an alternative feature to be selected, without even trying to invoke the requested role. This may be the case if the following feature selection policy existed: "when the `b` role is requested in a `UserCall` and the (called) user is not at home, always select call-forward instead". This checking returns either the requested role (if no feature selection policy was satisfied) or an alternative one. The selected role is then target (2) for checking the collaboration and role binding policies to decide whether it may be actually invoked. If yes, a confirmation message is sent back to the requesting role. Note that if the role that is finally selected to be invoked is not the originally requested one, the confirmation message may be replaced by a negotiation phase (not shown in Fig. 3). If otherwise collaboration and/or role binding policies are not satisfied, (3) a search is again performed for a substitute role that may be invoked, and, if found, the process is repeated from (2), until a role with specific features is agreed and invoked. In addition, if an invoked role does not achieve its goal during the service execution, a search for an alternative role, implementing a mid-role-triggered feature, can be made once more (e.g. to invoke call-forward on no-answer).

From the above explanation three generic events can be distinguished that trigger the selection of features describing alternative behavior. These events are:

- *OnRoleRequest,*
- *OnUnsuccessfulRoleBinding*, and
- *OnNonAchievedGoal* event.

Feature selection policies can then be defined, by for example end-users, as event-condition-action (ECA) rules, where the event is one of the three just mentioned, the condition is expressed in terms of the context and the action is the selection of a feature.

Note that up to now we have just talked about the use of feature selection policies to select features of a base service. However their potential is actually greater than that. There is nothing that prevents us from using feature selection policies to specify any service as an alternative to another one. That is, we may specify which event and condition leads to the substitution of a role X for a role Y, where roles X and Y are not necessarily related. In this case, the role at the requesting side must most likely be also substituted. A negotiation between the parties would then be necessary.

The use of policies for service-execution management and personalization is not novel. For example, the Call Processing Language (CPL) [10] is used to describe and control Internet telephony services. With CPL users can themselves specify their preferences for service execution. Reiff-Marganiec and Turner [16] also propose the use of policies to enhance and control call-related features. The novelty of our work lies in the structuring of policies we make, based on the proposed service architecture.

4. Conclusion

We have presented a two-dimensional service oriented architecture where service components are roles defined using UML 2.0 collaborations and system components are agents representing domain entities such as users and terminals. Service modularity is achieved by the separation of service components from system components, and by general policy-driven mechanisms for dynamic role binding that handle context dependency, personalisation, resource limitations and compatibility validation. Central parts of this architecture, such as the role request pattern and a simple form of XML-based role-binding policy, have been implemented in ServiceFrame [5] and have been used to develop numerous demonstrator services within the Program for Advanced Telecommunication Services (PATS) research program [14], which is a cooperation between the Norwegian University of Science and Technology (NTNU), Ericsson, Telenor and Compaq (now Hewlett-Packard). These experiments have confirmed that dynamic role binding is central not only to traditional telecom services, but also to a wide range of convergent services, and that explicit support for role-binding helps to manage the complexity of such services. The use of more advanced role-binding policies specified as BeanShell [12] scripts has also been studied in [19]. At the time of writing this paper, ServiceFrame has been extended with support for java-based role-binding, collaboration and feature selection policies that can be specified by both end-users and service providers to handle context dependency [15].

An interesting problem that has not been treated is undesirable interactions between two or more roles simultaneously played by an agent in different services. This is known as the feature interaction problem. We believe that our policy-driven mechanisms for dynamic role binding can help to avoid such interactions, if the agent maintains the consistency between the policies imposed in different services. We are also investigating in this direction.

Acknowledgments

We wish to thank NTNU and the ARTS project for partial funding this work. G. Melby and K.E. Husa at Ericsson have been extremely valuable through numerous discussions and by implementing the core of the architecture. We are also grateful to the many MSc students that have contributed by making services using these principles and by proposing and prototyping role binding mechanisms. Finally we appreciate all the input from our close colleagues J. Floch, R. T. Sanders, F. Krämer and H. Samset.

Notes

1. IVR stands for Interactive Voice Response machine

References

[1] Amer, M., Karmouch, A., Gray, T. and Mankovski, S. (2000). Feature-interaction resolution using fuzzy policies. In *Feature Interactions in Telecommunications and Software Systems VI*, pages 94–112, Glasgow, Scotland, UK.

[2] Bordeleau, F., Corriveau, J. P. and Selic, B. (2000). A scenario-based approach to hierarchical state machine design. In *ISORC '00: Proceedings of the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, page 78. IEEE Computer Society.

[3] Bræk, R. and Haugen, Ø. (1993). *Engineering Real Time Systems. An object-oriented methodology using SDL.* Prentice Hall.

[4] Bræk, R. (1999). Using roles with types and objects for service development. In *IFIP TC6 WG6.7 Fifth International Conference on Intelligence in Networks (SMARTNET)*, pages 265–278, Pathumthani, Thailand. Kluwer.

[5] Bræk, R., Husa, K. E. and Melby, G. (2002). ServiceFrame: WhitePaper. *White paper*, Ericsson Norarc. Available at: http://www.pats.no/devzone/platforms/ServiceFrame/doc/ServiceFrameWhitepaperv8.pdf.

[6] Bræk, R. and Floch, J. (2004). ICT convergence: Modeling issues. In *System Analysis and Modeling (SAM), 4th International SDL and MSC Workshop*, pages 237–256, Ottawa, Canada.

[7] Castejón, H. N. (2005). Synthesizing state-machine behaviour from UML collaborations and Use Case Maps. In Prinz, Andreas, Reed, Rick, and Reed, Jeanne, editors, *12th SDL Forum*, volume 3530 of *Lecture Notes in Computer Science*, pages 339–359, Grimstad, Norway. Springer.

[8] Floch, J. (2003). *Towards Plug-and-Play Services: Design and Validation using Roles.* PhD thesis, Department of Telematics, Norwegain Univ. Science and Technology, Trondheim, Norway.

[9] Krüger, I. H., Gupta, D., Mathew, R., Moorthy, P., Phillips, W., Rittmann, S. and Ahluwalia, J. (2004). Towards a process and tool-chain for service-oriented automotive software engineering. In *Proceedings of the ICSE 2004 Workshop on Software Engineering for Automotive Systems (SEAS).*

[10] Lennox, J., Wu, X. and Schulzrinne, H. (2004). Call Processing Language (CPL): A language for user control of internet telephony services. RFC 3880, IETF.

[11] Lupu, E. C. and Sloman, M. (1999). Conflicts in policy-based distributed systems management. *IEEE Trans. Softw. Eng.*, 25(6):852–869.

[12] Niemeyer, P. (1997). BeanShell - lightweight scripting for java. Available at: http://www.beanshell.org/.

[13] Object Management Group (2004). *UML 2.0 Superstructure Specification.*

[14] Program for Advanced Telecom Services (PATS). Accessible at: http://www.pats.no.

[15] Pham, Q. T. (2005). Policy-based service personalization. Master's thesis, Dept. of Telematics, Norwegian University of Science and Technology (NTNU).

[16] Reiff-Marganiec, S. and Turner, K. J. (2003). A policy architecture for enhancing and controlling features. In *Feature Interactions in Telecommunications and Software Systems VII*, pages 239–246, Ottawa, Canada.

[17] Sanders, R. T., Castejón, H. N., Kraemer, F. A. and Bræk, R. (2005). Using UML 2.0 collaborations for compositional service specification. In *ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, Montego Bay, Jamaica.

[18] Sanders, R. T., Bræk, R., Bochmann, G. v. and Amyot, D. (2005). Service discovery and component reuse with semantic interfaces. In *12th SDL Forum*, Grimstad, Norway.

[19] Støyle, A. K. (2003). Flexible user agent. Technical report, Dept. of Telematics, Norwegian University of Science and Technology (NTNU).

[20] Zave, P. (2003). Feature disambiguation. In *Feature Interactions in Telecommunications and Software Systems VII*, pages 3–9, Ottawa, Canada.

[21] Zibman, I., Woolf, C., O'Reilly, P., Strickland, L., Willis, D. and Visser, J. (1995). Minimizing feature interactions: An architecture and processing model approach. In *Feature Interactions in Telecommunications III*, pages 65–83, Kyoto, Japan.