Service Discovery and Component Reuse with Semantic Interfaces

Richard T. Sanders[1], Rolv Bræk[2], Gregor von Bochmann[3], and Daniel Amyot[3]

[1] SINTEF ICT,
NO-7465 Trondheim, Norway
Richard.Sanders@sintef.no
[2] Department of Telematics
Norwegian University of Science and Technology
NO-7491 Trondheim, Norway
rolv.braek@item.ntnu.no
[3] SITE, University of Ottawa
800 King Edward,
Ottawa (ON) Canada, K1N 6N5
{bochmann, damyot}@site.uottawa.ca

Abstract. Current trends in distributed computing and e-business processing suggest that many applications are evolving towards Service Oriented Computing (SOC) with technologies such as Web services. Services are autonomous platform-independent computational elements, and we observe an increasing need for core SOC technologies for dynamic discovery, selection, and composition of services. However, such technologies are often based on syntactic descriptions of the services and of their interfaces, which are insufficient to ensure that desired liveness properties are satisfied. In this paper, we propose an approach for the description, discovery, and selection of services based on role modeling and goal expressions that enables the definition of semantic interfaces and the evaluation of liveness properties. The same mechanisms also enable component reuse. We discuss how UML 2.0 can support the modeling of both the services and the desired properties. The approach is illustrated with telephony services.

1 Introduction

Many emerging distributed applications, platforms, and architectures, such as Web services and grid architectures, attempt to take advantage of the concept of service. A *service* is an autonomous platform-independent unit of work done by a provider to achieve desired end results for a consumer. The purpose of increasingly popular Service-Oriented Architectures (SOA) is often to promote the use and reuse of application-neutral services and components, and to achieve loose coupling between the participating entities. Such architectures contain three main parts: a provider, a consumer, and a registry. Providers publish or announce their services on registries, where consumers find and then invoke them. To support such Service-Oriented Computing, several protocols and languages

have been developed to characterize, register, discover, invoke, and compose services [16].

Of particular interest is the problem of selecting a service that can interoperate with a client application and that can meet the desired goals of the collaboration. As we move toward open environments where anyone can wrap existing functionalities or create new ones and then offer them as remote services, being able to select the most appropriate service (if any) becomes imperative. Current enabling technologies are often based on *syntactic* descriptions of the services and of their interfaces. For example, the Web-Service Description Language (WSDL) uses ports, operations, and message types to define the abstract interface and protocol bindings of a service [17]. A UDDI registry catalogs such service characteristics, together with business and category information [13]. Other service discovery protocols (e.g., SLP, SDP, Jini, Salutation, and UPnP) describe services with identifiers, types, attributes (including some quality of service), and/or static interfaces [3]. We believe such descriptions to be insufficient to ensure that liveness properties (the collaboration goals) desired by the service customer are satisfied. A discovered service may offer the required static interface but may not be able to achieve the desired goals; it should then not be selected.

To tackle this problem, we propose an approach for the description, discovery, and selection of services based on role modeling and simple goal expressions that enables the definition of *semantic* interfaces and the evaluation of liveness properties. These mechanisms are generic enough to address the related issue of component reuse. Section 2 presents how UML 2.0 [14] can support the modeling of both the services and the desired properties. Typical usages of semantic interfaces are discussed in Section 3. The approach is illustrated with telephony services, but is not limited to the telecommunication domain. Our conclusions follow.

2 Semantic Interfaces

2.1 Distributed Systems Architecture

Fig. 1 suggests an architecture for service-oriented systems which is characterized by horizontal and vertical decomposition. On the horizontal axis, several computational object (*actors*) are identified that may reside in different computing environments. This axis represents the physical and logical distribution of the system. On the vertical axis, several services are identified that are provided by the distributed systems. In the simplest situation, these services are provided independently of one another. In practice, however, there are usually constraints relating to resources of an actor that are shared by the components involved in the different services, which leads to dependencies between the different services.

The main concern in this paper is the compatibility of the different service components involved in the provisioning of a given service. (We note that one or several of these components may constitute a user agent). In the following, we

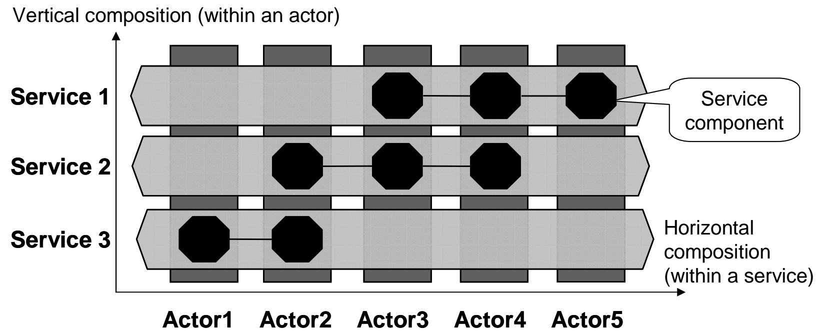

Fig. 1. Two-dimensional view of a Service-Oriented Architecture

call these service components simply "*components*". Typically, each component interacts with several other components within the same horizontal service. For a given component, we identify a number of *interfaces*, one for each other component with which it cooperates. Fig. 2 shows an example of two components, CA and CB, each having two interfaces, and where the two components interact with one another through the interfaces A and B, respectively.

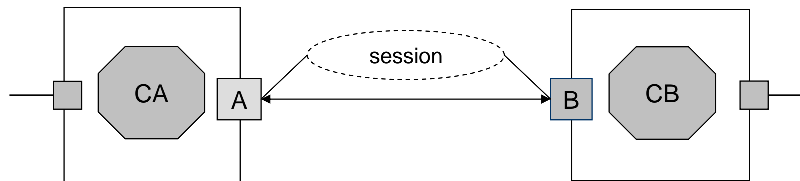

Fig. 2. Two interacting components with two interfaces each

The figure also indicates that messages are exchanged between these two components. What is labeled *session* in the figure, actually represents the *collaboration* between these two components. Different graphical notations have been proposed in UML for representing collaborations [14]. In this article, we consider the description of collaboration behaviors at different levels of abstraction. At the highest level, we use UML activity diagrams which only represent the order in which certain phases of the service collaboration proceeds. If we want to describe globally the messages exchanged during a collaboration, we use UML interaction diagrams. These two descriptions are not necessarily complete; they typically concentrate on certain use cases. For a complete description of a collaboration, we consider first the state machine description of the behavior of each of the components. Since such a *component behavior* description includes all interactions of the given component over all interfaces, it is more complex than required, if we are only interested in the collaboration of the component over

a specific interface. Therefore we also consider the projection of the component behavior over a given interface, which we call the interface *role behavior* of the given component. It is the behavior it exhibits over the interface where it plays a particular role in the collaboration.

We use in the following the term *semantic interface* to denote a collaboration including the role behavior of the participating components and the progress goals (see below) that should be reached by the collaboration.

Note that this approach is not bound to UML: activity diagrams could be replaced with Use Case Maps (UCM), interaction diagrams with Message Sequence Charts (MSC), and state diagrams with the Specification and Description Language (SDL).

2.2 A Simple Example

We consider the following simple example from telephony in order to explain the concepts introduced in this paper. The telephony service involves two components, the user agent of the calling user, named A, and the user agent of the called user, named B. The diagram of Fig. 3 shows the structural aspect of this collaboration; only the interfaces between these two components are shown, not the interfaces with their respective terminals and users. The diagram also shows a progress goal that should be reached by the collaboration: The system should reach a global state where VoiceCnt(A, B) is true, that is, A has a voice connection to B and B has a voice connection to A.

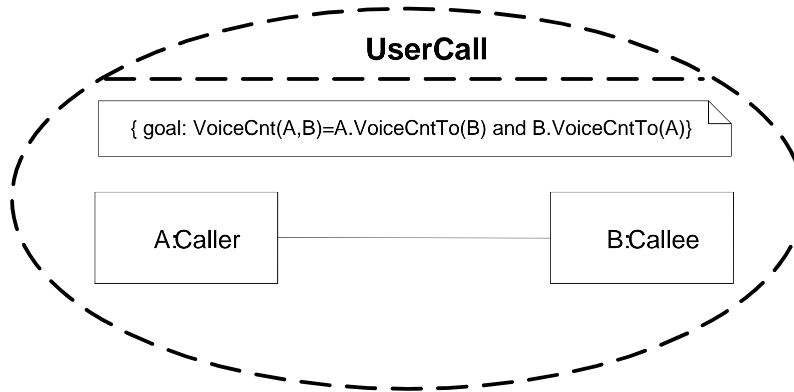

Fig. 3. The UserCall collaboration structure

We consider that a UserCall collaboration may involve the following phases: Invite, Calling, and Busy, as shown in the activity diagram of Fig. 4. This is a very basic behavior that may be enriched with more service features as will be illustrated in Section 3.

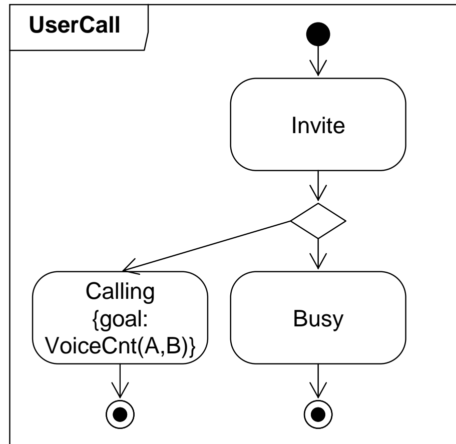

Fig. 4. Phases of the UserCall collaboration

The following sequence diagrams give more details about these phases. They show the overall sequencing of the phases, similar to the activity diagram above as well as the interactions leading up to the connected state. Reaching the connected state is clearly a goal for the UserCall service, and this is represented by the goal expression goal: VoiceCnt(A,B) where VoiceCnt(A,B) is a predicate over properties of the two participating roles A and B, for instance:

$$\mathsf{VoiceCnt(A,B)} = \mathsf{A.VoiceCntTo(B)\ and\ B.VoiceCntTo(A)}$$

In order to keep the example simple, we only show a basic call handling service with one feature, WaitOnBusy, in Fig. 5. Using the same general approach a much richer set of features can be designed.

Finally, the two state transition diagrams in Fig. 6 show the role behavior of the two collaborating components in a featureless basic call service. Since only one interface is considered for each component so far, the role behavior at these interfaces is identical to the overall behavior of these components. In these diagrams, we have omitted behavior needed to resolve conflicts in the case of initiative collisions which may occur in states with both input and output transitions (called mixed initiative states in [5,8]). In reality, the role behaviors must be extended to include behavior for resolving such conflict situations (e.g., see the work of Gouda [9] and Floch [8]).

It should be noted here that the diagrams in Fig. 6 define what we call the *semantic interface* of the UserCall collaboration. In addition to defining the static interface in terms of the signal types interchanged in each direction (not explicitly shown here), it also defines the interface behavior in terms of sequences of interactions and the goals in terms of predicates specifying properties of desirable states and events.

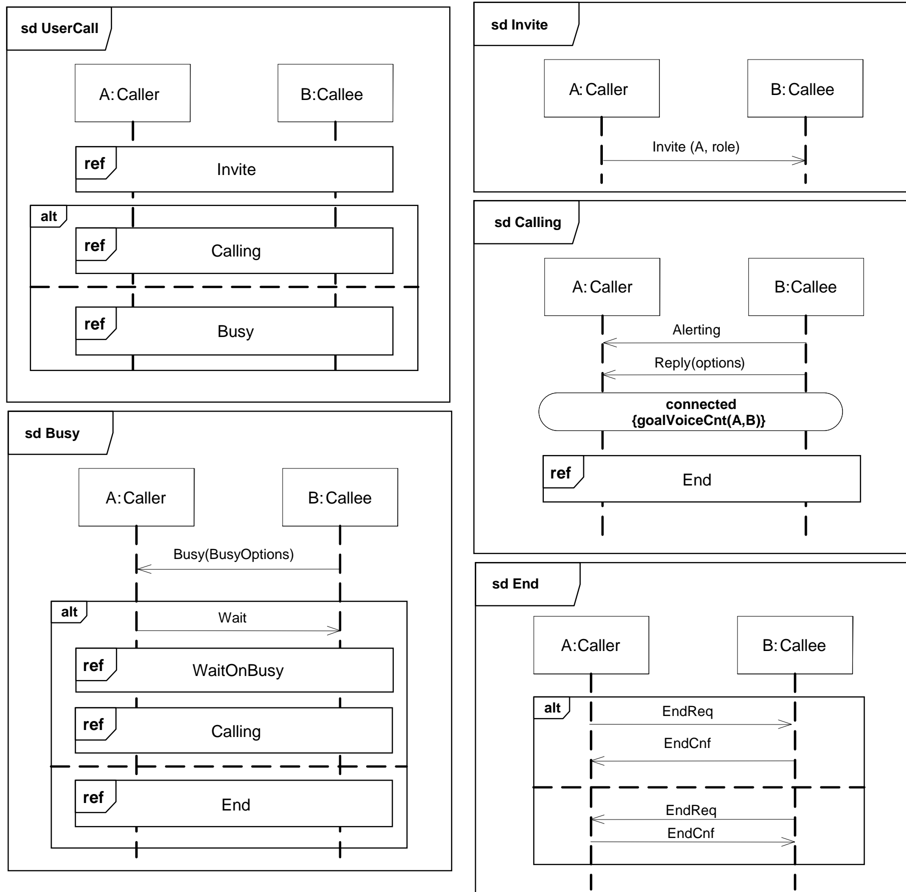

Fig. 5. UserCall interactions

2.3 Criteria for Component Reutilization and Service Discovery

To set the stage for the following discussion, we consider two situations where it is important to compare different component specifications and implementations:

1. We consider the following scenario of *service discovery* (refer to Fig. 2): A component CA is given; and we are looking for a "service" that presents an interface *similar* to the one presented by component CB, such that CA and CB may perform a collaboration similar to the one described above.
2. We consider a scenario of *component reuse*, where during system design we identify a system component C with a given requirements specification S_C. Now we are looking for an existing implementation I that could be used as component C in the implementation of the system.

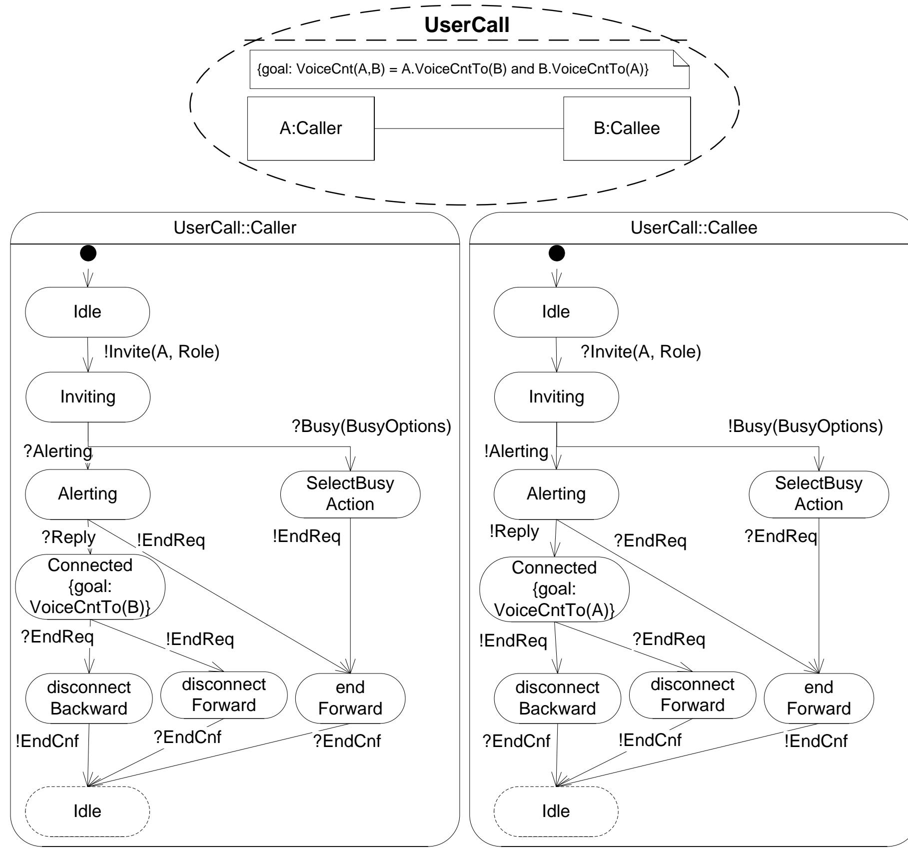

Fig. 6. A semantic interface with a goal and interface role behaviors

We assume in the following that a requirements specification S is given in the form of some logical predicate that must be satisfied by the implementation. This predicate may define a state machine, or it may be of a more general form. We also assume that each implementation I can be described by a logical predicate P_I that characterizes all the properties of the implementation. Then we have the following lemmas/definitions:

Lemma 1 (Conforming implementation). *An implementation I conforms to the requirements S if $P_I \Rightarrow S$ (logical implication).*

Lemma 2 (Specialization). *A specification S' is a specialization of another specification S if $S' \Rightarrow S$. (One also says that S' is a* subtype *of S).*

Lemma 3 (Reuse of a component). *An implementation I that conforms to the requirements S' can be reused as an implementation for a component that must satisfy the specification S, if $S' \Rightarrow S$.*

A requirement has often the form of an implication: If certain assumptions about the environment of the component are satisfied then the component will have to satisfy certain guaranteed properties [1]. We therefore assume that a requirements specification is given the form $S = (As \Rightarrow GP)$, where As represents the assumptions and GP the guaranteed properties. Then we can say that S' is a specialization of S if $(As' \Rightarrow GP') \Rightarrow (As \Rightarrow GP)$ which is equivalent to $(As \Rightarrow As') \wedge ((As \wedge GP') \Rightarrow GP) \vee (\neg(As \Rightarrow As') \wedge GP)$. This means that S' is a specialization of S if, and only if, the assumptions of S' are weaker than or equal to those of S and the guarantees of S' (when the assumptions of S are satisfied) are stronger than or equal to those of S, or the guarantees of S are satisfied independently of any other assumptions.

We note that component requirements can usually be divided into structural (static) properties and dynamic properties that relate to the dynamic behavior of the component. The following paragraphs discuss shortly the structural properties; aspects related to the dynamic behavior are discussed in the next subsection.

Definitions of interfaces (in the sense of object-oriented languages or SDL channels) represent structural properties. An interface definition states what kind of methods must be provided by a class (a guarantee provided by the component); it also states that the component may make the assumption that the environment will not try to call a method that is not defined. The latter assumption is usually checked by the compiler according to the type checking rules of the language.

The declaration of the type of a parameter in an input message for a given component specification represents the assumption that the environment will only provide input parameters of the specified type. Conversely, the declaration of the type of a parameter in an output message (or return value of a method call) represents a guarantee to be provided by the component that only values of that type should be presented in the output message. Together with the definition of specialization given above, this leads to quite general type checking rules, as for instance defined for the Emerald language [2].

2.4 Formalizing Safety and Liveness Properties of Collaborations

Subtyping of State Machines. We assume in the following that the dynamic behavior of a component is specified in the form of a (deterministic) finite state machine where each transition is either associated with an input or an output. We also assume that the specification of the static structure defines the set of input and output interactions that may occur at each of the interfaces of the component. Assuming that a state machine specification only defines safety properties of the component (we will discuss liveness properties later in this section), we interpret a given state machine specification for a component C as follows:

1. If the component produces an output, then the state machine has an output transition with this output as label from the current state of the component.

When doing the output, the component enters the new state defined by that transition. This is in fact a guarantee that any produced output is one of those allowed by the specification.

2. If the component receives an input, it will enter a new state as defined by the transition (from its current state) labeled by this input. The specification makes in fact the assumption that only inputs that are defined for the current state in the specification will be produced by the environment of the component.

Based on this interpretation of state machine specifications, we note that a state machine specification S', which is obtained from a given specification S by adding some additional input transitions, has weaker assumptions than S and defines therefore a subtype behavior of S. Note that a new input transition may lead to an existing state or a new state; additional input and output transitions may be added from the new states while keeping the subtyping relationship. We call this form of subtyping *extension*, indicated by the label "ext" in the UML icon for inheritance (see Fig. 8). The diagram in Fig. 7 provides an example where the CalleeW role is an extension of the Callee role from Fig. 6.

Similarly, if S' is obtained from S by removing some output transitions, then S' defines a subtype behavior of S. In this case, some liveness properties may get lost, because the subtype restricts the interaction possibilities. We call this form of subtyping *reduction*, indicated by the label "red" in the UML icon for inheritance. (This is similar to the reduction of nondeterminism considered in [6]). As an example, the Caller role in Fig. 6 is a reduction of the CallerW role in Fig. 7.

Safety Compatibility Requirements for Collaborations. We now consider that a component CA should collaborate with a component CB, as shown in Fig. 2. If we are only interested in the compatibility of these two components for the interactions taking place over the common interface, we first can make abstraction of the interaction of these components over other interfaces. This operation of abstraction is often called *projection* and consists of hiding all interactions that do not occur over the interface of interest. This projection operation, applied to the state machine S defining the overall behavior of the component, leads in general to a nondeterministic machine. We assume in the following that the well-known determination algorithm ([10], which is of exponential complexity) has been applied in order to obtain a deterministic state machine $Proj_{IF}(S)$ showing the behavior of the component S at the interface IF of interest.

We now consider the collaboration between the components CA and CB with specifications SA and SB, respectively. We note that compatibility between CA and CB means that CA only sends interactions to CB that CB can handle in its current state, and inversely, that CB only sends interactions to CA that CA can handle in its current state. In other words, the guarantees of $Proj_{IF}(SA)$ imply the assumptions of $Proj_{IF}(SB)$, and the guarantees of $Proj_{IF}(SB)$ imply the assumptions of $Proj_{IF}(SA)$.

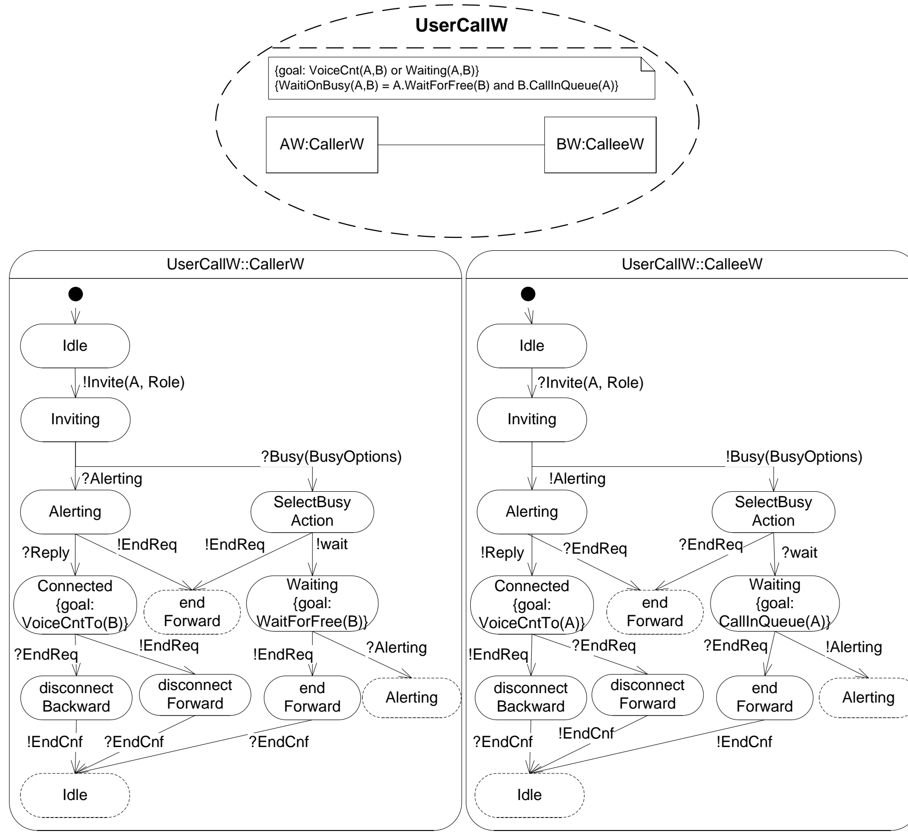

Fig. 7. Semantic interface for UserCallW: UserCall with WaitOnBusy feature added

We now may make the assumption that the interactions over the interface are immediate, that is, an output interaction generated by one component is immediately consumed as input by the other component, without any intermediate queuing. We call such an interface a *direct coupling interface*. Although not very realistic for distributed systems, this kind of interface is used for many theoretical models, e.g., Input/Output Automata [11]. It has the advantage that no cross-over of messages in opposite directions may occur over the interface.

Lemma 4. *Given a component CA with dynamic behavior $Proj_{IF}(SA)$ over the interface IF, the most general (in the sense of our specialization relation) behavior at the interface for the collaborating component CB is given by the state machine obtained from $Proj_{IF}(SA)$ by exchanging for each transition the direction of interaction (replace input by corresponding output or output by corresponding input). See for instance the work of Gouda [9] or Drissi [7].*

In the more realistic case where outputs are queued within the communication medium before they are consumed as input by the destination component, the compatibility conditions are more complex because messages may cross over within the medium and the order of inputs and outputs occurring at one component may be different than the order of the corresponding outputs and inputs at the other components. Gouda [9] and Floch [8] propose some interesting approaches for deriving a most general behavior at the interface for a component CB collaborating with a given component CA.

Considering Liveness Properties in Collaborations. While safety talks about constraints that must be satisfied for any valid execution sequence that may occur, liveness properties talk about certain progress that should be made or states that should be reached. Various approaches have been proposed for describing liveness (or progress) properties. We propose in this paper the notion of a *goal* which is a predicate on the local or global state space of the system. We say that a system satisfies a given goal G if one of the execution paths of the system leads to a state s for which $G(s)$ is true. This is equivalent to a statement in branching time temporal logic saying that there exists a branch that leads eventually to a state for which G holds. In general, the requirements of a given system may include several goal predicates that should be reachable. If a single state should be reachable that satisfies a set of goals $G_1, \ldots, G_n$ simultaneously, this can be expressed by a new goal of the form $G = G_1 \wedge \ldots \wedge G_n$.

In the example of the telephone system presented previously, the activity diagram of Fig. 4 contains the mention goal: VoiceCnt(A,B) for the activity Calling. This means that the Calling phase should be reachable, while the text VoiceCnt(A,B) has no formal meaning at this point. In Fig. 5, the location of this goal (within the reachable global state space) is further refined. Also, in Fig. 6, the same goal is described from the point of view of one of the components participating in the collaboration; here we see that a particular state of the state machine should be reachable. The additional predicate VoiceCntTo(B) may be evaluated within the component A by referring to additional variables not shown in the diagram.

As mentioned earlier in this section, when one replaces, within a given system, a component A with behavior S by another component A' with behavior S' where S' is a subtype of S, it could be that certain global (and local) states that were reachable with S are not reachable anymore with S' (except in the case when S' is a pure extension of S). If such states are associated with goals, these goals would not be reachable anymore. We conclude that the definition/lemma (3) mentioned in Section 2.3 must be revised by indicating that the relationship $S' \Rightarrow S$ implies that there is no problem for reuse as far as safety properties are concerned, but there may be a problem concerning progress, unless the subtype relationship is a pure extension. We do not know of any general rule for solving this dilemma, however, we know that the relevant progress properties may possibly not be satisfied by the replacement component S'. These progress properties should

therefore be checked. This could be done by performing a reachability analysis of the collaboration in question.

We conclude this section by noting that the compatibility of two components participating in a collaboration over a given interface has two aspects: safety properties and progress properties. These properties can be checked by considering the role behavior of the components which is the behavior of the components projected onto the collaboration interface. Safety compatibility means that any output produced by one component is acceptable by the other component when it is received as input. In the case of interfaces without any message transfer delay (excluding cross-over of messages) this relationship is easily checked by comparing the respective role behaviors. In the case of message delays over the interface, the verification of compatibility is much more complex, and can for instance be solved by reachability analysis, which however may involve arbitrarily long message queues. The safety-oriented subtyping relationship of dynamic behaviors is useful for deciding these compatibility questions.

We introduce in this paper a notation for specifying progress properties of collaborations. It is important to note that they are not implied by the behavior specifications that are commonly represented by finite state machines or other similar formalisms. We show in this paper how progress properties can be taken into account during the definition of collaborations and for identifying components that are compatible with a given component, not only as far as safety is concerned, but also concerning the progress properties.

3 Using Semantic Interfaces

3.1 Service and System Composition

We assume now that each component type may have a number of interfaces, and that each interface is defined (typed) by referring to a semantic interface. Safety and liveness properties may be analyzed once for each semantic interface as explained in Section 2. This analysis needs not be repeated for each component, but it is necessary to check for each component type that its behavior is consistent with the role behaviors attached to its interfaces. This means to check that the role behavior is a projection of the component behavior as explaines in Section 2.4. When this is done, the semantic interfaces can be used to check compatibility of static and dynamic links between component instances in a service or system structure.

Traditional model checking is performed on instance structures and does not scale well when the structures change and grow. By analyzing each component type and semantic interface separately and building maps over subtype relationships, much of the computation intensive work can be done once and for all at design time and thus reduce the work needed at runtime. In this way, semantic interfaces provide an enabler for scalable, runtime compatibility checks in dynamic system structures. This is especially important in systems where new services and components may be added dynamically, as for instance in the

emerging service oriented computing paradigm, but it is important in any system with dynamic link structures.

As a simple example consider the case presented in Fig. 7. Here the UserCall has been extended with a WaitOnBusy feature. More precisely, the Callee role has been extended so that CalleeW ⇒ Callee (extension). In order to fully explore this extended role behavior, corresponding output must be added to the Caller role as shown in the CallerW role. As a consequence, the Caller ⇒ CallerW (reduction), as illustrated in Fig. 8.

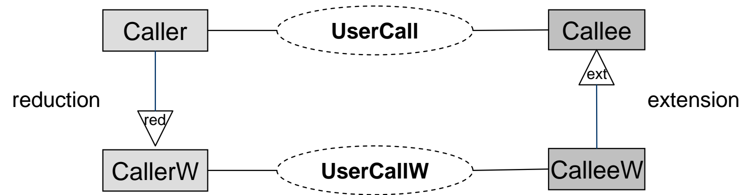

Fig. 8. Subtyping relationships for the semantic interfaces of UserCall and UserCallW

Now, consider four components that subscribe to these interfaces as illustrated in Fig. 9. Obviously two components that provide dual roles of the same interface will fully satisfy safety and liveness properties when connected across the interface. In addition, components providing the Caller role may inter-work safely with components providing the CalleeW role. However, in this case the WaitOnBusy goal is not reachable. CallerW and Callee are incompatible, as indicated in Fig. 9.

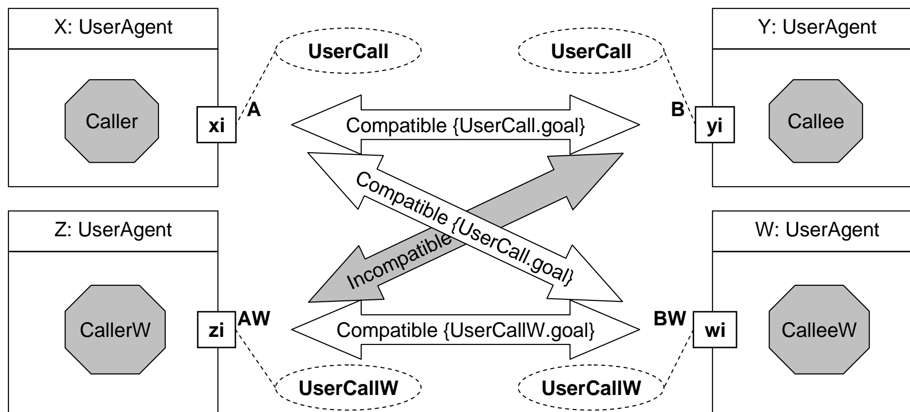

Fig. 9. Using semantic interfaces

In service providing systems, it is quite common that links between components are dynamic. In telecom services for instance, the links between user agents, terminals and other objects change from call to call. The service features available at a given instant will normally depend on subscription information, user preferences, current state and available resources. Therefore, it may be necessary to check the compatibility for each dynamic link that is established. In our examples this has been omitted, but could be added as checks or possibly negotiations performed during the Invite phase.

In many telecom services, such as the UserCall presented here, the identity of the objects playing the service roles are important. The Caller for instance wants to reach a particular user, not just any user. In other cases the identity is not so important. The problem is to find some object that can provide a service or part of a service, in other words to find an object that can play a given role. This is a case of service discovery.

3.2 Service Discovery

Service discovery has two dimensions:

1. Finding existing component types and instances that can provide a desired behaviour across a semantic interface. This is needed when a component with a given semantic interface needs to find and connect to a compatible component over that interface. We call this *discovery of complementary components*.
2. Finding new component types and new semantic interfaces that can provide new or enhanced services, e.g., finding out if an actor can perform new or enhanced services by obtaining a new type of component. We call this *role learning*.

In the following sections we present approaches to these challenges.

3.3 Discovery of Complementary Components

Discovery of complementary components is a mechanism by which a component can determine what other component types or instances are capable of playing compatible complementary roles. This may be accomplished by specifying a desired semantic interface and role, given knowledge of collaboration roles and their subtyping relationships. An example is presented below.

Components are defined with their semantic interfaces in Fig. 9. Given knowledge of the interface role subtyping relationships in Fig. 8, compatibility relationships and goal opportunities can be analyzed efficiently. Interoperable, complementary components can be found with which services can be performed, while incompatible components can be avoided:

– X looks for components that are compatible with the UserCall.Callee interface. This is obviously Y, since it subscribes to that interface. It is also W because UserCallW.CalleeW is an extension of UserCall.Callee, as shown in Fig. 8.

– Y looks for components that are compatible with the UserCall.Caller interface. Component X can be found, but not Z, since UserCallW.CallerW is incompatible with UserCall.B.
– Z looks for components compatible with UserCall.CalleeW. Component W can be found, but not Y, again due to incompatibility.
– W looks for components compatible with UserCall.CallerW. Z is compatible and can achieve all the goals of UserCallW. X is compatible, but can only reach the goals of UserCall, i.e., without the WaitOnBusy feature.

Given the subtype relationships calculated at design time, one can search for components with compatible interfaces, and determine that they satisfy safety and liveness properties, i.e., have the possibility of reaching service goals when interacting.

Discovery of complementary components is a static comparison of semantic interfaces of component types. It does not take the current state of components into consideration. The objective is not to discover or learn new behavior, an issue we discuss below.

Note that component interfaces can be classified as either initiating ("client" side) or offered ("server" side), depending on which interface is designed to take the first initiative in the collaboration. This can be exploited to simplify the discovery procedure, since a component is only interested in finding compatible offered interfaces.

3.4 Role Learning

Given knowledge of what interface roles are deployed in its environment, it may be desirable for an actor object to learn new behavior so that it can achieve more service goals when interacting with components towards its environment. It can for example use a lookup mechanism to search for service components that give a better match against offered roles, i.e., components that can achieve more progress.

Given a service brokering function or registry, an actor may perform a request for a component that can enable it to take part in a new service or a "better" service than previously. This will require that a new component with new semantic interfaces is downloaded. A possible collaboration pattern for this is depicted in Fig. 10.

In Fig. 10, a component X sends a request to component W to play a UserCall.Callee role. The role request is confirmed, since W is capable of a consistent collaboration with X. However, in the role confirmation, W supplies a description of its interface behavior CalleeW. As described earlier, CalleeW extends Callee with WaitOnBusy functionality. In step 3, X consults a service broker to check for the existence of a service component that is a better match for CalleeW than Caller, i.e., a service component that can achieve more service goals. Note that X supplies a description of its semantic interface Caller. Steps 4 thru 6 result in the service component CallerW being identified and retrieved from an appropriate

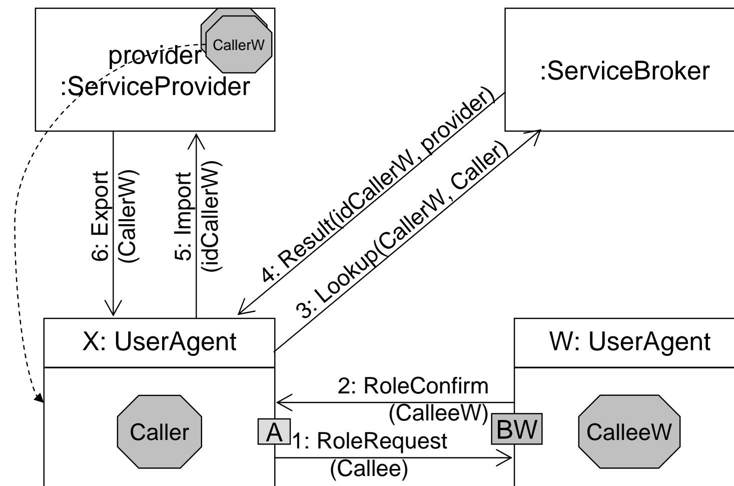

Fig. 10. Role learning pattern used to retrieve a new component

service provider. X has thus improved its functional repertoire by obtaining the CallerW component.

Whether X casts away the Caller component is an open issue; remember that CallerW cannot collaborate safely with Callee components, so the Caller component may be useful in another context. The example also does not indicate what component is used in the collaboration with W. Which component to select can be decided in the Invite phase of the call. Fig. 10 illustrates how this may be accomplished by replacing the Invite signal by an interaction consisting of a RoleRequest(requested-role) signal and a RoleConfirm(granted-role) signal. Compared with existing lookup services, the enhancements lie in the description of the semantic interface, and the learning factor made possible by issuing and granting role requests [4,12].

4 Conclusions

We have described and illustrated how semantic interfaces, composed of behavior expressions annotated with goals, can be used for the selection of services and of components while satisfying the liveness properties of the collaboration. This improvement over the use of static interfaces by existing service discovery mechanisms prevents the selection of services that would lead to unsatisfactory or even dangerous behavior. Goals can be attached to behavior models at various level of abstraction, for instance to UML 2.0 activity, interaction, or state diagram elements (or respectively to UCM, MSC, and SDL model elements). Subtyping relationships such as extensions and reductions, together with role-based projections, enable the efficient comparison between desired collaboration behavior and available services and components. Semantic interfaces can sup-

port service selection but also more advanced discovery functionalities such as role learning.

The telephony example used here illustrates in simple terms the description and selection mechanisms. However, applications for such technology are not limited to telecommunication. We anticipate practical use in many convergent services, where information technology services and telecommunication services unite (e.g., Web services and grid services, applied to many vertical domains).

Due to the well-defined projection relationship between component behavior and semantic interfaces, it is possible to provide tool support for deriving semantic interfaces, and for checking compatibility between semantic interfaces and component behaviors. We have developed prototype tools to demonstrate this. Rather than being an additional burden for the service engineer, semantic interfaces may be integrated into the service engineering process in ways that can support both productivity and quality. Scalability of the approach follows from the relative simplicity and compositional nature of the compatibility checks needed among component instances. These checks can be limited to checking compatibility among semantic interfaces, which may be pre-calculated for component types and interface types at design time.

Service discovery as outlined here relies on well-defined interface names and maps over inheritance relations between semantic interfaces. It also relies on a common understanding of goals and the relationship between goals, services and service features. One way to achieve this would be to define a suitable ontology over goals, services, and features, using approaches suggested in the semantic Web community [15]. In order to enable service discovery across different service providers, this ontology must be shared. We plan to investigate how emerging standards like the Web Ontology Language (OWL) [18] could improve the description of semantic interfaces and be used to allow matches across different domains.

Acknowledgments

This research was supported by Telenor R&D and by the Natural Sciences and Engineering Research Council of Canada, through its programs of Strategic Grants and Discovery Grants.

References

1. Abadi, M. and Lamport, L.: Conjoining specifications. ACM Transactions on Programming Languages & Systems, vol.17, no.3, May 1995, 507-534.
2. Black, A., Hutchinson, N., Jul, E., Levy, H., and Carter, L.: Distribution and Abstract Types in Emerald. IEEE Trans. on Software Engineering, Vol. SE-13, no. 1, January 1987, 65–76.
3. Bettstetter, C. and Renner, C.: A Comparison of Service Discovery Protocols and Implementation of the Service Location Protocol. Proc. EUNICE Open European Summer School, Twente, Netherlands, Sept 13-15, 2000.

4. Bræk, R., Husa, K.E., and Melby, G.: ServiceFrame: WhitePaper. Ericsson Norarc, 2002. `http://www.item.ntnu.no/lab/nettint1/ServiceFrame/ServiceFrame.html`
5. Bræk, R. and Haugen, Ø.: Engineering Real Time Systems. An Object Oriented Methodology using SDL. Hemel Hempstead, Prentice Hall, 1993.
6. Brinksma, E. and Scollo, G.: LOTOS specifications, their implementations and their tests. Protocol Specification, Testing and Verification VI (IFIP Workshop), Montreal, 1986, North Holland, 349–360.
7. Drissi, J. and Bochmann, G.v.: Submodule construction tool. M. Mohammadian (Ed.), Proc. Int. Conf. on Computational Intelligence for Modelling, Control and Automation, Vienna, Feb. 1999, IOS Press, 319–324.
8. Floch, J.: Towards Plug-and-Play Services: Design and Validation using Roles. Ph.D. thesis 2003:47 NTNU, Norway, 2003.
9. Gouda, M.G. and Yu, Y.-T.: Synthesis of communicating Finite State Machines with guaranteed progress. IEEE Trans. on Communications, vol. 32, No. 7, July 1984, 779–788.
10. Hopcroft, J.E., and Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley Publishing Company, 1979.
11. Lynch, N.A. and Tuttle, M.R.: An introduction to input/output automata. CWI Quarterly, 2(3), 1989, 219–246.
12. Melby, G. and Bræk, R.: Delivery of convergent telecom services on J2EE platforms. Int. Conf. on Intelligence in Service Delivery Networks, ICIN, Bordeaux, France, October 2004.
13. OASIS: Universal Description Discovery & Integration (UDDI), Version 3.02, February 2005. `http://www.oasis-open.org/committees/uddi-spec`
14. OMG: UML 2.0 specifications. `http://www.omg.org/uml`
15. Paolucci, M., Kawamura, T., Payne, T.R., and Sycara, K.P.: Semantic Matching of Web Services Capabilities. I. Horrocks, J.A. Hendler (Eds.): The Semantic Web-ISWC 2002, First International Semantic Web Conference, Sardinia, Italy, June 9-12, 2002, Proceedings. Lecture Notes in Computer Science 2342, Springer 2002, 333–347
16. Singh, M.P. and Huhns, M.N.: Service-Oriented Computing: Semantics, Processes, Agents. John Wiley & Sons, 2005.
17. World Wide Web Consortium: Web Services Description Language (WSDL) 1.1. March 2001. `http://www.w3.org/TR/wsdl`
18. World Wide Web Consortium: Web Ontology Language (OWL). February 2004. `http://www.w3.org/2004/OWL/`