

CoSDL - An Experimental Language for Collaboration Specification

Frank Rößler¹, Birgit Geppert¹, and Reinhard Gotzhein²

¹ Avaya Labs, Software Technology Research, 233 Mount Airy Road, Basking Ridge, NJ 07920, USA
{roessler, bgeppert}@research.avayalabs.com

² Comp. Networks Group, Univ. of Kaiserslautern, P.O. Box 3049, D-67653 Kaiserslautern, Germany
gotzhein@informatik.uni-kl.de

ABSTRACT The decomposition of distributed systems is often driven by its process structure only, focusing on the behaviour of individual agents. In previous work, we have argued that this is not always an adequate modularization of distributed systems, and have proposed “cross-cutting” collaboration modules instead. In this paper, we discuss language support for the specification of collaboration modules that goes beyond the capabilities of SDL and MSC. In particular, we introduce the experimental formal description technique CoSDL, which was designed as a “proof of concept” for collaboration-based design with SDL. We believe that the lessons learned from CoSDL are valuable for collaboration-oriented extensions of MSC, SDL, and corresponding tool environments.

KEYWORDS Collaboration-based design, scenario modelling, distributed systems, formal description technique, SDL, MSC, CoSDL

1 Introduction

In [4] and [6], we have introduced the concept of collaborations together with a collaboration-based design process for SDL systems [7]. The motivation for this work originated from the observation that distributed systems are often decomposed according to their process structure only, thus compromising intelligibility, changeability, and other desirable quality attributes. A process module is typically modelled as a communicating extended finite state machine with an interface of incoming and outgoing messages, encapsulating behaviour by one logical agent. Since a process-oriented decomposition is driven by the logical distribution of the system, there are possible disadvantages:

- Process modules might cooperate in complex ways and therefore can not be understood independently. Since their interaction behaviour is not directly represented, it is difficult to derive the global system behaviour.
- An important and complicated phase in distributed systems development is the transition from system behaviour to the behaviour of interacting components. As indicated above, a process-oriented decomposition often obscures global system behaviour and therefore can not support a systematic approach for this transition.
- If there is intense interaction, changes to one process module usually affect other process modules, which violates the information-hiding criterion for an adequate decomposition.
- Also, single process modules often can not be reused in another context.

In order to resolve these problems, we suggest to encapsulate the interaction behaviour by so-called *collaboration modules*, while providing a clean mapping between the resulting module structure and the actual process structure. For this purpose, we have defined a collaboration-based design process consisting of three steps [4]:

Step 1: Specification of single collaborations (using the experimental collaboration specification language CoSDL [6])

Step 2: Collaboration composition (using CoSDL)

Step 3: Collaboration transformation (from CoSDL to SDL)

The *concept* of collaborations as well as the collaboration-based *design process* have been discussed in [4]. In this paper, we focus on *language support*, and present details of CoSDL, an experimental language for formally specifying collaborations [6].

For a comprehensive specification of collaboration modules, we need to consider several aspects. Among them are a module's interface, its secrets, and its actual implementation. In this paper, we focus on the module's interaction behaviour. Note that a collaboration module cuts across all components of the process structure that participate in a specific distributed functionality. When specifying a collaboration, we want to encapsulate this functionality, in particular, the intended causal dependencies among the involved agents. This defines requirements on language support (such as send/receive events, conditionals, or states), which are typically covered by existing *scenario modelling notations* such as MSC [8], UML sequence diagrams [1], UML collaboration diagrams [1], and use case maps [2]. However, there are other language constructs that are essential for collaboration specification and not covered by these notations at all. This is due to the fact that scenario modelling describes example system traces for a fixed set of process instances. A collaboration module, however, specifies the complete interaction behaviour of the considered distributed functionality. This completeness demand and the fact that interacting agents are involved make it necessary to describe *agent topologies* as part of a collaboration specification, too. Furthermore, those topologies may change dynamically. In order to tackle these issues, we need a type concept for collaboration agents, and a way to describe allowable instantiations and to recursively create agent topologies. In Section 2, we introduce the collaboration specification language CoSDL, which provides these language features. In Section 3, we compare CoSDL to other scenario modelling notations, and draw some conclusions.

2 Collaborations in SDL Systems - CoSDL

We introduce CoSDL informally by means of a running example, which is taken from the Session Initiation Protocol [3], analysed in [5], and exemplify CoSDL by the INVITE collaboration. SIP is a signalling protocol for establishing, modifying, and terminating multimedia sessions over the Internet, where the INVITE collaboration is responsible to set up an initial call between the communicating parties. The language constructs are introduced step by step, while the example gains in complexity. Note that we do not intend to provide a complete CoSDL specification of the INVITE collaboration. The selected scenarios are just for illustration purposes. The formal syntax and semantics of CoSDL is defined in [6].

2.1 CoSDL operations

Collaboration agents¹ are capable of performing certain operations: sending/receiving messages, starting/ending collaboration threads, and setting timers. Additionally, agents apply specific control structures in order to organise an collaboration's control flow: decisions, iterations, states, forks and joins. Forks and joins specify concurrent behaviour both within and between agents. Agents do completely abstract from data, which includes both local and transmitted data. This section focuses on agent operations, while other language constructs are introduced in later sections.

Let us start with the first version of our running example, which specifies a simple scenario from the INVITE collaboration of SIP. The corresponding CoSDL specification in Figure 1a shows a sequence of causally related messages (written as plain text, e.g., `invite`) together with the corresponding sending and receiving agent instances (written in italics, e.g.,

¹ When a collaboration is monitored during runtime, we observe a set of participating agents. We allow that many instances of the same agent type can occur in a collaboration and that the concrete number of agent instances can vary.

proxyServer). Informally, the following collaboration is specified: if the agent instance *userAgentClient* sends an *invite* message, the agent instance *proxyServer* will receive it and forward it to the agent instance *userAgentServer*. Subsequently, *userAgentServer* will answer with a *response* message, which *proxyServer* simply relays back to *userAgentClient*. We expect a system implementing this collaboration to exactly execute this message sequence, whenever *userAgentClient* sends an *invite* message, and causal dependencies on other collaborations are met.

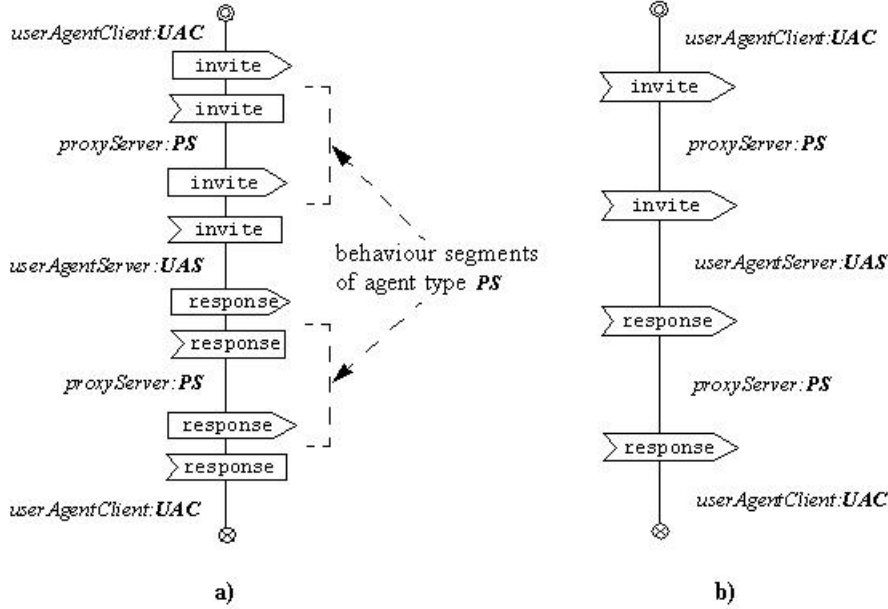







Figure 1. INVITE collaboration: simple scenario

By default, a collaboration diagram assumes a control flow from top to bottom along a specified flow line (straight line in Figure 1a). Whenever necessary, arrows can be used in addition to make causal relationships between operations clear. Figure 1a employs four kinds of operations. First of all, we have the send  and receive  operation. Send- and receive operations always occur in pairs with the same message name. This describes the transmission of a message from its sending agent to its receiving agent. We assume an implicit transport mechanism for this purpose. Take, for instance, the topmost send/receive pair in Figure 1a. It shows that an *invite* message is sent from *userAgentClient* to *proxyServer*. In other CoSDL diagrams, we will also use the send-receive symbol  as an abbreviation for such send/receive pairs. In fact, Figure 1b makes use of the send-receive symbol and describes exactly the same collaboration as Figure 1a.

Furthermore, Figure 1 shows the use of the start-  and stop  operation. They mark beginning and ending of collaboration threads², respectively. We think of the start operation as a recipient of external stimuli which can occur at any time - even repeatedly. Whenever such a stimulus occurs, the operations that follow the start symbol on the flow line will be executed. In particular, this implies that several collaboration threads can be active simultaneously, i.e., *userAgentClient* may issue additional *invite* messages before previous threads have been completed. The different threads process concurrently and can overtake each other. If such behaviour should not be allowed, we need additional control structures which will be explained later. Each start operation is assigned to a specific agent instance. In Figure 1, it belongs to

² During runtime, collaboration behavior will be triggered by external stimuli. All interactions that occur as a causal effect to such a stimulus and belong to the given collaboration are called a *collaboration thread*.

userAgentClient. It is, however, not known when *userAgentClient* will receive an external stimulus and start a new collaboration thread.

Whenever a thread reaches a stop operation, it terminates. Stop operations can also be thought of as transition points for stimuli to the environment. Note that termination of a collaboration thread does not imply that the involved agent terminates also. Agent threads exist independently from collaboration threads. In fact, CoSDL will completely abstract from the creation and termination of agent instances. Their existence is assumed when a collaboration thread binds an agent that does not yet belong to the thread topology³.

In spite of its simplicity, Figure 1 already demonstrates the central characteristic of a collaboration description language: it is shown that operations of all involved agents are explicitly put in a causal order. We call a consecutive sequence of operations of the same agent a *behaviour segment*. In Figure 1, *proxyServer* has two behavior segments: first, an *invite* from *userAgentClient* followed by an *invite* to *userAgentServer*. Second, a response from *userAgentServer* followed by a response to *userAgentClient*. The segments follow strictly one after the other and are interleaved with a behaviour segment of *userAgentServer*.

As behaviour segments of individual agents are scattered across the specification, we attach the instance and type name of the corresponding agent as a label to each behavior segment. The label syntax is: the instance name in *italics* is followed by a bold type name, while instance and type name are separated by a colon. Labels are drawn close to their corresponding behaviour segments.

2.2 CoSDL control structures

With sequence as the only control structure, we were able to specify single scenarios. This section introduces further control structures: decision, iteration, and states. Let us first extend the collaboration of Figure 1 by a possibility for refusing a call request. The resulting specification is shown in Figure 2. The decision symbol \diamond expresses that reception of an *invite* either causes a *response(acc)* or a *response(ref)* message sent by *userAgentServer* and received by *proxyServer*. This makes *proxyServer* report back a *response(acc)* or a *response(ref)*, respectively, to *userAgentClient*.

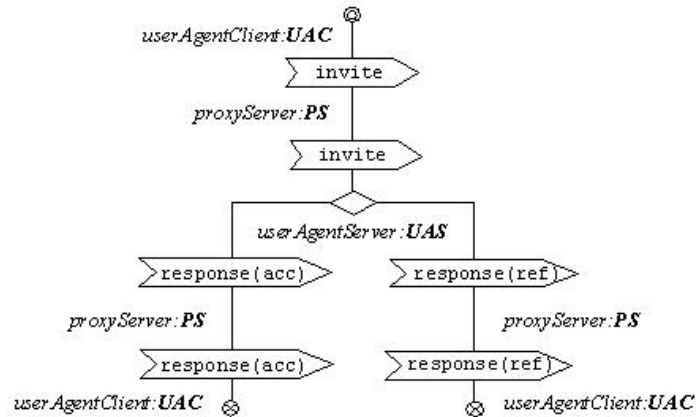


Figure 2. INVITE collaboration: alternative scenarios

For the next extension of the running example, it is required to ring more than one target location in order to find a called party. This behaviour is part of the user location capability of

³ We call the set of agent instances that have participated in a collaboration thread at a specific moment during runtime the *thread topology*. When a collaboration thread transfers control to a new agent instance for the first time, it is said to *bind* this agent instance to the thread.

SIP. If a proxy server maps a SIP address onto several target addresses, it can try each location one after the other. Consider the specification of Figure 3, which extends our specification, so that *proxyServer* can try two different locations when establishing a call. If either *userAgentServer1* or *userAgentServer2* accepts the call by responding *response(acc)*, the connection is established. Otherwise, the call gets refused. Note that an accepted call is handled identically by *proxyServer*, irrespective of the answering user agent server. We therefore specified the corresponding behaviour segment only once and used an *unbranch* to connect the behaviour segments of *userAgentServer1* and *userAgentServer2* to it. It is straightforward to extend the specification for any fixed number of called parties. However, for the general case we need a loop construct (iteration).

A loop is established when a decision branch is directed back into a preceding behaviour segment. CoSDL requires that the backward branch originates and ends in a behaviour segment of the same agent instance, because otherwise the specification can get cluttered easily. If a loop is required that connects two different agent instances, one can use macros (see later section) instead. The behaviour segments within the loop are iterated an indeterminate number of times. As CoSDL abstracts from data aspects, the concrete number of iterations cannot be derived from the specification.

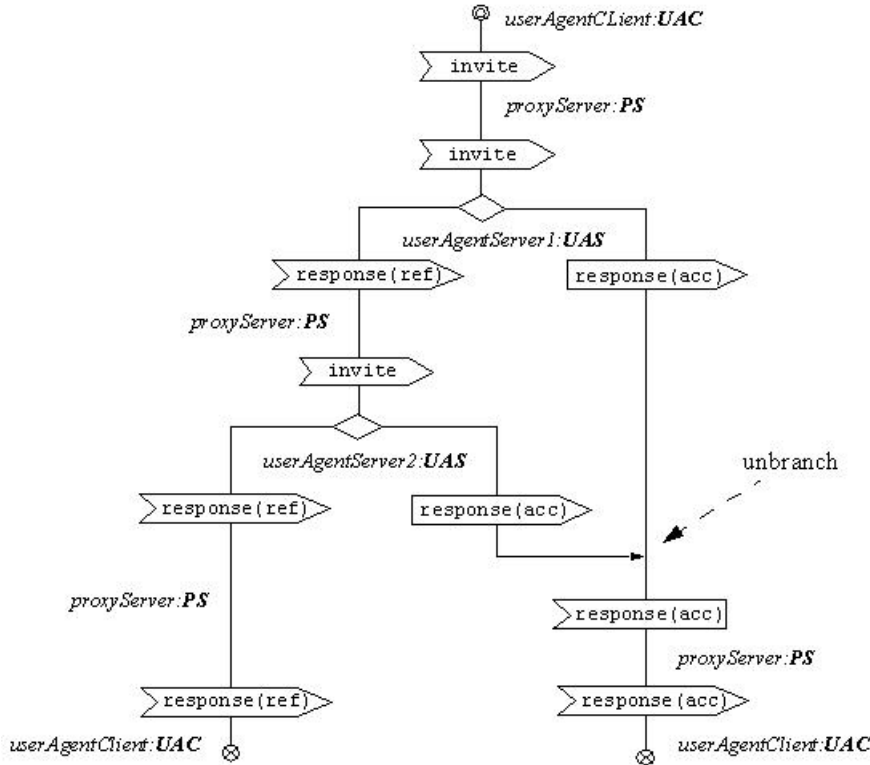


Figure 3. INVITE collaboration: scenario with two targets

Except for *proxyServer*, the loop in Figure 4 involves only one additional agent instance per iteration, though in general any number of agents is possible, even none. For the current example, however, the more important question is whether individual iterations address the same or different agent instances of type *UAS*. The shadowed rectangle (containing *UAS*'s behaviour segment) together with its modifier "*bind to loop (loop-unique)*" and its multiplicity *1*, enforces exactly the semantics that is expected from the INVITE collaboration: each iteration calls a different user agent server. The new CoSDL language construct is called *replica* and will be discussed in a later section. It only needs to be known at this point that

replicas allow multiple instances of the same agent type to be bound. Thus, the collaboration topology implied by Figure 4 allows for an agent instance *userAgentClient* of type *UAC*, an agent instance *proxyServer* of type *PS*, and an indeterminate number of agent instances of type *UAS*. For those instances, the instance name *userAgentServer* does not really identify the instances any more. The CoSDL semantics uses instance IDs for identification purposes.

We have already mentioned that several simultaneous collaboration threads can be created at a start operation. We will introduce other language constructs also that split off new message flows. With multiple threads in place, there is automatically a need for adequate synchronisation mechanisms, which allow an agent to control the order of threads propagating through his behaviour segments. Whenever a message is received, the reaction can theoretically follow promptly, as all causal dependencies are met (CoSDL leaves timing issues open). In case of Figure 4, e.g., this means that whenever *userAgentClient* receives an external stimulus, a specific set of user agent servers will be called. That is, several of these threads may be conducted concurrently (under racing conditions of course).

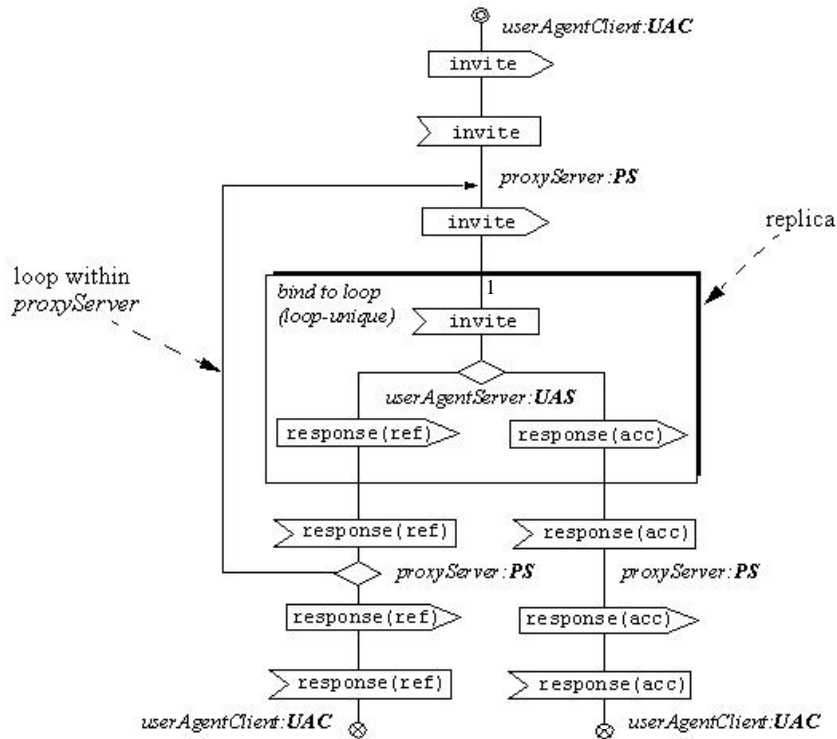


Figure 4. INVITE collaboration: scenario with n targets

For handling the sequencing of concurrent collaboration threads within an agent, the concept of state is additionally employed. A state is a static condition an agent instance meets between processing steps. States are preconditions for processing steps and can also change afterwards. In order not to clutter the graphical CoSDL specification too much, an operation by default turns the state of an agent instance into a unique state (*implicit* state) that is different from all explicit states. Syntactically, explicit states \square are used as shown in Figure 5. States that act as a precondition have an outgoing arrow that points onto the conditional operation. In Figure 5, *proxyServer* will only process incoming *invite* messages, if it is in state *idle*. Otherwise, the message will be stored for later processing (see *save* attribute). After processing an *invite*, *proxyServer* enters the implicit state, as no explicit state is specified. This differs when *proxyServer* is sending a *response(ref)* or a *response(acc)* message. Then the state will change to *idle* again. Note that operations with no state attached to them do

not change the agent's state at all. They can be triggered in any state and do remain in that state when completing the processing.

There are two additional issues to be clarified: what initial state does an agent instance enter when created, and how are incoming messages handled by conditional receive operations? The state at the top of Figure 5 has an incoming arrow denoting the initial state of an agent instance. Such an arrow must only be used once for each agent type. Remains the question what happens to messages at a conditional receive operation, if the precondition does not hold. We have two possibilities: with the *save* attribute stated, the incoming messages will be stored. Otherwise, i.e., without any attributes, the messages will be discarded.

It is now possible to understand the semantics of the specification in Figure 5: call requests, which can be initiated at any time by *userAgentClient*, are all handled sequentially by *proxyServer*, while in Figure 4, this is done concurrently.

2.3 Concurrency in CoSDL: fork and join

Except for concurrent collaboration threads, the example specifications above have prescribed sequential behaviour only. It is, however, possible for a SIP proxy server to call multiple target locations in parallel. This saves time during connection establishment, if the called party is expected at several different places. Figure 6 shows an updated version of Figure 3 that supports concurrency. The fork operation splits off two concurrent message flows and sends an *invite* along each path. From that point on, there are two message flows, which traverse independently through the behaviour segments. In fact, both of them produce either a *response(acc)* or a *response(ref)* that is transmitted back to *userAgentClient*. Note that *proxyServer* does not synchronize the concurrent message flows. In particular, the unbranch lets each message pass as they arrive. SIP supports the concept of stateless proxies, which means that responses are simply forwarded upstream without further processing. In that case, *userAgentClient* is responsible for coordinating incoming messages from the different target locations. Note that we do not show detailed behaviour of user agent clients in the CoSDL diagram.

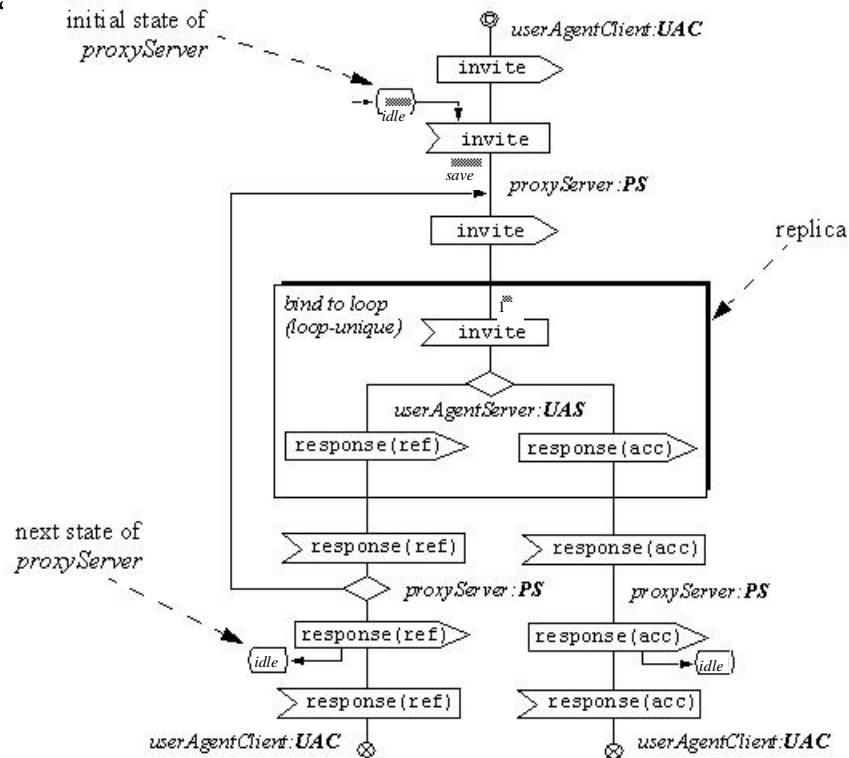


Figure 5. INVITE collaboration: n targets, sequential handling of call requests

In the example, the fork operation splits off only two message flows, though any number is allowed by CoSDL. Furthermore, CoSDL has a join operation (not shown in the example) which is used to synchronise message flows. A join operation blocks incoming message flows, until it has received a message from each of them. We use the same symbol for both the join and fork operation.

2.4 CoSDL replicas

So far, the number of participating agent instances has always been fixed for our running example - except for Figure 4 and Figure 5, which have already used CoSDL replicas. The given examples specified exactly one participating user agent client, one proxy server, and one or two user agent servers. With the CoSDL language constructs that have already been introduced, it is possible to extend these collaborations by any specific number of participating agent instances. The name labels that are attached to behaviour segments show which agent instances execute them⁴. We made use of this mechanism in all of the previous CoSDL diagrams - also for binding two instances of the same type (Figure 3).

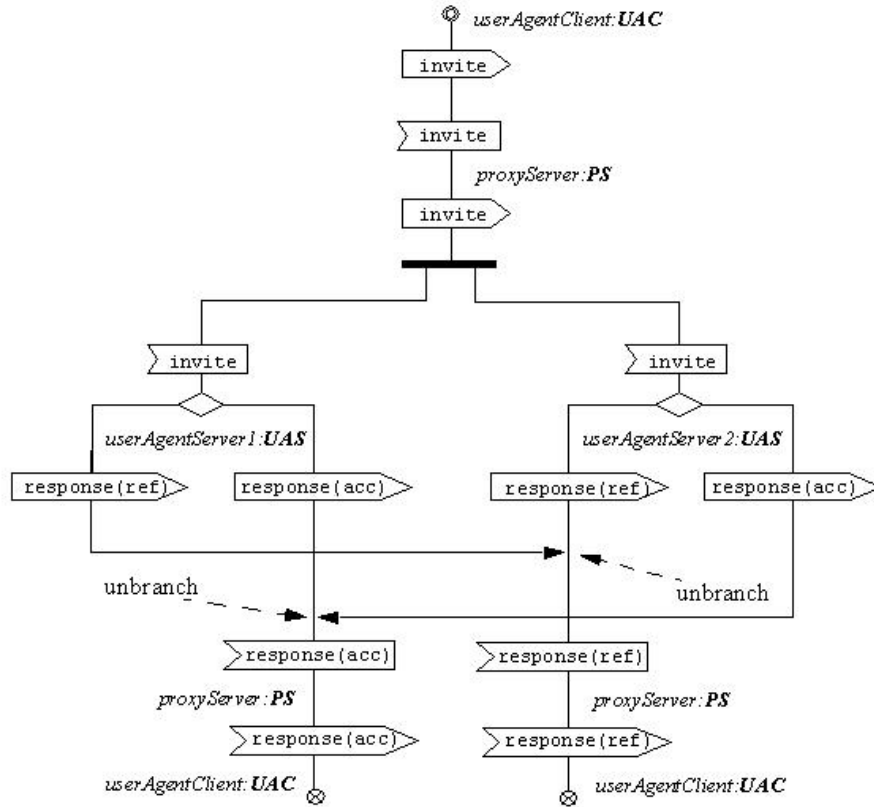


Figure 6. INVITE collaboration: two targets and simultaneous call requests

In reality, however, SIP provides for much more flexible topologies. For instance, the number of intermediate proxies between a user agent client and user agent server can vary for each thread of an INVITE collaboration. In addition, an indeterminate number of user agent clients and servers will use the proxies. Note that the number of different agent types is fixed for each collaboration. However, the number of participating agent instances may vary. In order to express this variety of topologies, we need a mechanism for specifying *instance sets* of the same agent type. Instance sets allow an indeterminate number of agent instances to be

⁴ This assumes that an instance name (as part of the name label) identifies an agent uniquely. We will see later, that this is only true as long as the instance name is not used within a replica.

bound (called *multiple binding*). CoSDL specifications can apply this mechanism within loops, in connection with multicast messages, or in recursions. When used in a loop, it can bind new instances for each iteration. When used in a multicast, it binds the set of destinations. And when used in a recursion (see next section), each macro application binds a new instance of the specified agent type. Since CoSDL abstracts from data aspects, we refrain from specifying the concrete number of bindings.

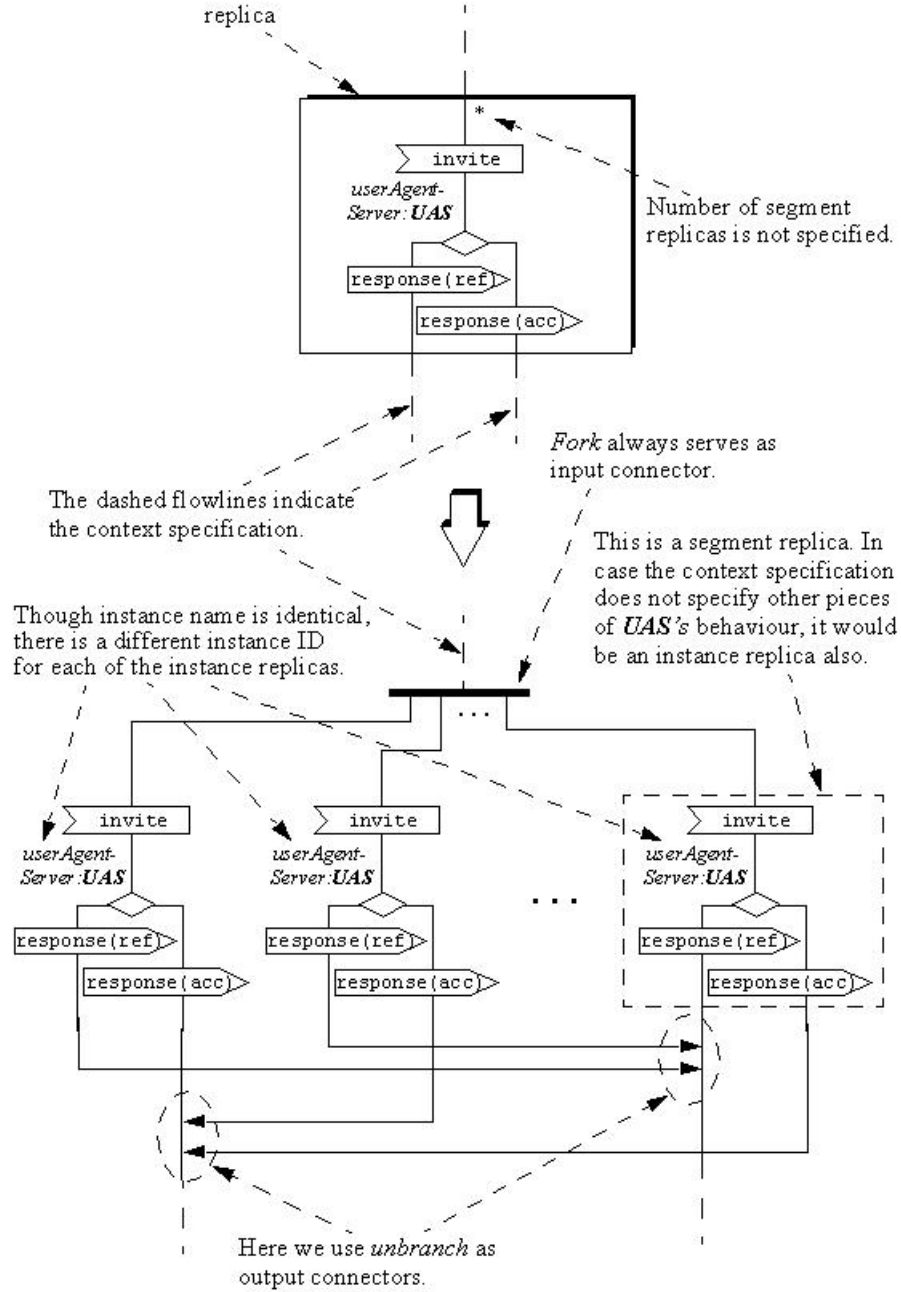


Figure 7. INVITE collaboration: CoSDL replicas in a multicast context

The CoSDL mechanism for instance sets and multiple binding is called *replica*. The defined symbol for a replica is a shadowed rectangle that frames at least one behaviour segment (Figure 4, Figure 5). Note that each behaviour segment has a name label that specifies an instance name and an agent type. If no replicas are used, the instance name denotes a specific agent instance. However, in a replica it actually denotes the instance set. If a replica

contains more than one behaviour segment, this means that more than one instance set and/or more than one agent type is involved. As a consequence, a replica can bind multiple instance sets of several agent types. In the following discussion, we only consider the simpler case where replicas refer to exactly one instance set and one agent type, and therefore contain only one behaviour segment.

In a multicast context, a replica stands for an indeterminate number of copies of the contained behaviour segment, where each copy belongs to another agent instance. Figure 7 illustrates how a replica can be pictured then. The upper part shows a replica as it could be specified in a regular CoSDL specification. The lower part shows how this replica be transformed into a set of behaviour segment copies. Of course, this is not valid CoSDL anymore, but illustrates the semantics of CoSDL replicas. Replicas are an obvious mechanism for generating multiple copies of one behaviour segment. We have to consider, however, that other behaviour segments may belong to the instance set that are scattered across the CoSDL diagram. Thus, it may be necessary to use extra replicas, because the complete behaviour description must be covered for generating an instance set. This case is illustrated in Figure 9, where two replicas are used to frame all behaviour segments of instance set *userAgentClient*.

For the following discussion, we must distinguish between the replication of complete behaviour descriptions of an instance set and the replication of the behaviour segment that is contained in a specific replica. We will call the former *instance replicas* and the latter *segment replicas* (Figure 7).

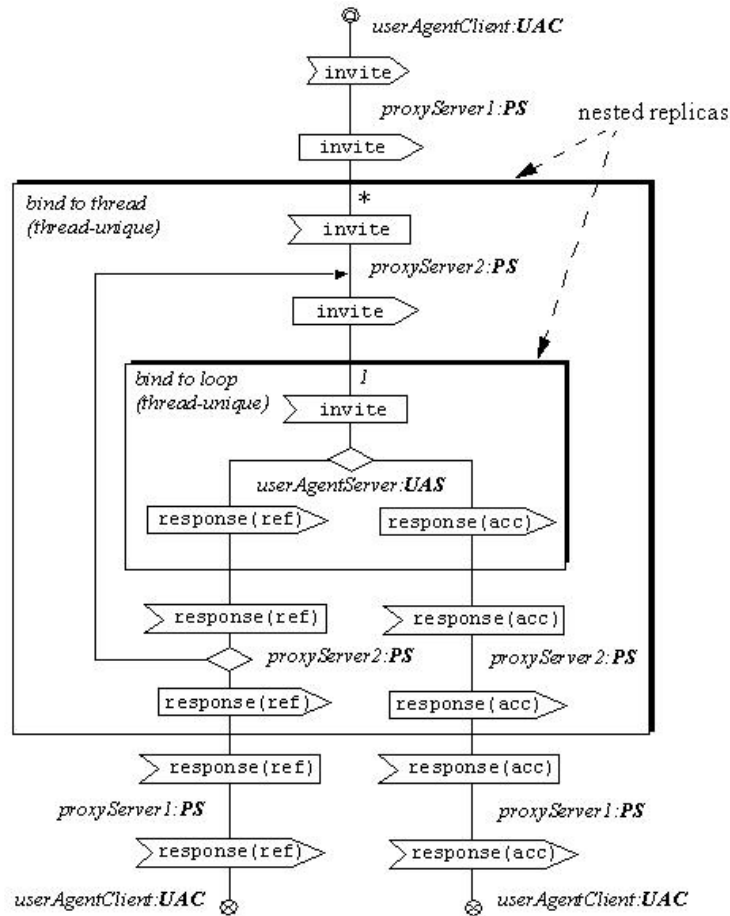


Figure 8. INVITE collaboration: multiple proxies and targets

For the definition of the semantics, we need to find a logical connection between segment replicas and the rest of the specification (called *context specification*). CoSDL always applies a

fork as an input connector and offers two choices as an output connector, namely unbranch⁵ and join (Figure 7). Thereby, input and output simply relate to the direction of the message flow. Having a fork as an input connector means that during runtime, all segment replicas are split off as concurrent message flows. Figure 7 applies an unbranch for each of the two outputs, which means that message flows from the different segment replicas proceed concurrently at this point. If we had used a join as output connector, the message flows would have been synchronised and combined into one message flow that continues processing.

As mentioned before, Figure 7 describes replicas in a multicast context. That is, the context specification sends an *invite* message to the complete instance set, which binds every instance in the set at once. We have described earlier that instance sets can also be bound within a loop or a recursion. In this case, instances of the set are bound one after the other, while the collaboration thread is progressing. Thus, we need an additional modifier that describes - among other things - the multiplicity of bindings at a time. The multiplicity is contained in the replica symbol. CoSDL only allows 1 or *, which means that exactly one or an indeterminate number of segment replicas are generated, respectively. That is, multiplicity * must be applied to establish a multicast context. In loops or recursions that do not require sending of multicast messages, we will apply multiplicity 1.

In Figure 5, for instance, the multiplicity is 1, so that the *invite* sent by *proxyServer* will be received by exactly one (user agent server) instance. However, for each iteration, a new agent instance will be bound⁶. As there is no special output connector symbol in Figure 5, unbranch is used by default. With only one segment replica, this is identical to a simple flow line.

Figure 8 shows a more advanced example for replicas. It applies *nested replicas*. For this case, the outer replica is first resolved into segment replicas, which still contain the inner replica. For each of these segment replicas, the (newly created) inner replicas are then resolved separately.

Figure 8 describes topologies with one user agent client, multiple user agent servers, and exactly two intermediate proxies between each client-server pair. The first proxy is the same for all client-server pairs, while the second can vary. The inner replica is interpreted as discussed above (in the context of Figure 5). The outer replica has multiplicity *, which means that we have an indeterminate number of segment replicas. Each of them can be pictured as being connected to a fork symbol as illustrated in Figure 7. When *proxyServer1* sends an *invite*, the fork operation splits off a new message flow for each segment replica and sends an *invite* down that path (multicast message). From that point on, these message flows operate concurrently. In particular, each *invite* will be received by a different proxy server instance, which in turn calls a set of user agent servers. The responses from all user agent servers are then reported back to *userAgentClient*. The *response* messages follow the same path as their corresponding *invite* messages, only in reverse direction. Note that the possible topologies described by the CoSDL diagram of Figure 8 do still not match with the real SIP INVITE collaboration. To be more realistic, we need to combine replicas with the macro construct, which will be introduced in the next section.

Instance sets and multiple binding raises two basic issues: first, during runtime it is possible that a specific replica be executed several times. In that case, will CoSDL bind a new instance set or will the already bound instances execute the behaviour segment again? Second, if many replicas must be used to cover all behaviour segments of an instance set, we want to make sure that a message flow that was processed by a specific agent instance in one replica will not switch to another instance of the instance set in a subsequent replica. We call the first

⁵ Unbranch is the default output connector, when no special connector symbol is used.

⁶ This is achieved by the additional modifier *bind to loop (loop-unique)*.

issue the *binding problem* and the second the *finding problem*. The modifiers such as “*bind to thread (thread-unique)*” or “*bind to loop (loop-unique)*” that occurred in the replicas of Figures 4REF, REF5, and REF8 deal with exactly those problems.

Let us first elaborate on the binding problem. The same replica can be processed many times during the lifetime of a collaboration. This happens for two possible reasons: first, new collaboration threads can be triggered at a start operation, which causes a specific replica to be traversed multiple times. Furthermore, the execution of loops that contain replicas has the same effect⁷. For both cases, it is important to specify whether a repeated execution of a replica implies the binding of new agent instances or not. CoSDL distinguishes three cases: first, the binding is fixed for the whole lifetime of the collaboration (*bind to collaboration*). Second, the binding is only fixed for the current thread and can change for another thread (*bind to thread*). Third, the replica is part of a loop and the binding is only fixed for the current iteration of that loop. Each iteration can bind different instances (*bind to loop*).

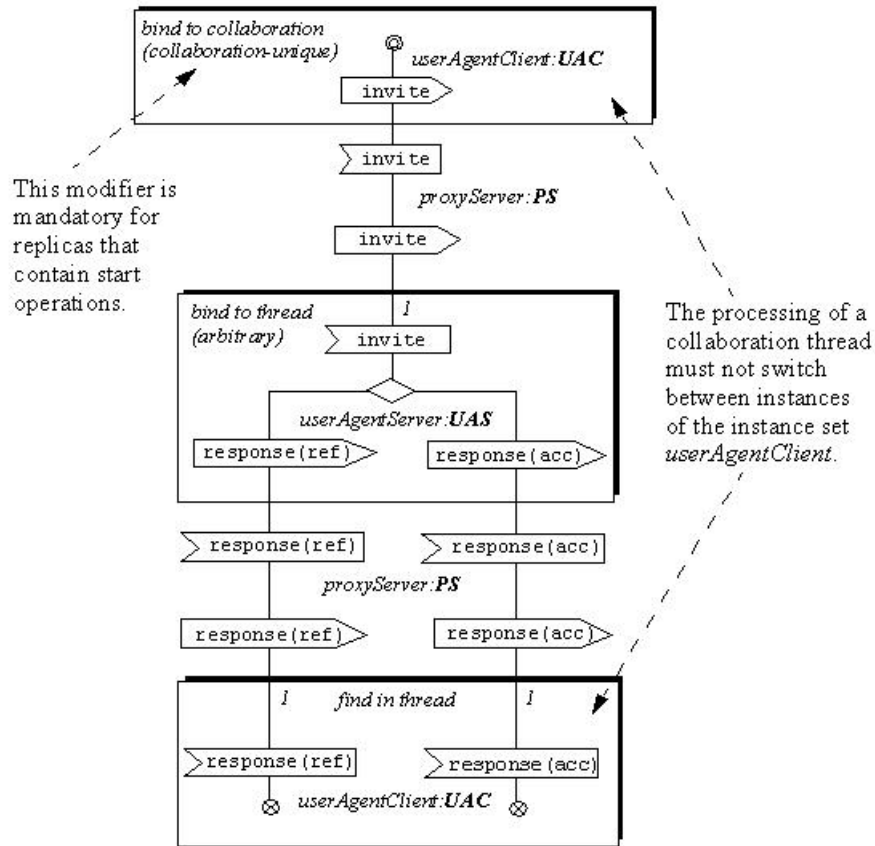


Figure 9. Finding problem

When new instances are bound, again different alternatives are possible then. The newly bound agent instances can be arbitrarily selected or unique within the collaboration-, thread-, or loop topology. Thereby, the loop topology is defined as the set of agent instances that have been bound, while a specific agent instance is executing the loop. In [6], we define the exact semantics of bindings with respect to the different choices that are offered by CoSDL. A modifier for binding the agent instances of a replica is built by choosing a scope descriptor and concatenating a uniqueness constraint in parentheses. The modifier is then placed somewhere

⁷ Actually, there is a third possibility for a replica to be executed multiple times, namely when concurrent message flows lead to the same replica (via a fork- and subsequent unbranch operation). However, this case does not need specific language constructs and can be handled together with the other cases.

inside the replica symbol. Note that modifiers of an outer replica do not apply to behaviour segments that are specified in inner replicas. In Figure 8, for instance, binding user agent servers is only ruled by the modifier “*bind to loop (thread-unique)*”.

Not every replica necessarily implies the binding of agent instances. If several replicas are needed to cover the behaviour segments of an instance set, only one of them, namely the first in the control flow of the collaboration, will bind the instances. For the other replicas, the above mentioned finding problem becomes relevant.

For illustration purposes, Figure 9 modifies the INVITE collaboration from Figure 4: other topologies are possible now, because an arbitrary number of user agent clients is participating. However, only one user agent server will be called per collaboration thread.

According to the explanations of the previous sections, we conclude that at any time, some user agent client can start calling a peer. The request will then be processed by *proxyServer* and reach some user agent server. Finally, *proxyServer* gets an answer back and forwards it to a user agent client. The question arises which user agent client will get the answer. Of course, that client is supposed to, that originally sent the *invite*. This, however, is not yet clear from the specification. Any instance of type *UAC* out of the current collaboration topology could be meant. That and similar ambiguities call for a mechanism that clearly determines, which agent instance is processing a behaviour segment in the context of a collaboration thread.

For a replica that is not a *binding replica*, the contained name label defines a set of eligible agent instances that could possibly execute the corresponding behaviour segment. The set of eligible instances for a specific name label contains all instances of the specified type that are part of the current topology and also share the instance name. In other words, the set of eligible instances for a specific name label is the instance set that was bound by the corresponding binding replica. We differentiate two scopes here: instances that are part of the current collaboration topology (*find in collaboration*) and instances that are part of the current thread topology (*find in thread*).

The set of eligible agent instances for a specific name label is further restricted by a condition that is called *sub role*. It is possible that agent instances of the same instance set follow alternative behaviour paths. In that case, we may only want instances that have executed a specific path to resume processing at the *finding replica*. An agent instance that follows a different behaviour path is entering a different *sub role*. We use states to indicate the sub role of an agent instance and conditional operations to further restrict the set of eligible instances in the finding replica. Depending on the replica’s multiplicity, CoSDL will select one or all of the eligible instances to continue the collaboration thread.

So far, we assumed that a replica contains only one behaviour segment. Since CoSDL actually allows more than one behaviour segment, we add some additional remarks here.

If the same replica binds different instance sets, those sets are linked in a special way. CoSDL basically keeps a relation that links those instances of the different sets that together executed the same segment replica⁸. We can use this relation in order to refine CoSDL’s finding mechanism. Consider, for instance, a softphone application that resides on a network node together with its SIP user agent client and server. In a real scenario, we will have many of these nodes connected to a network of proxy- and redirect servers. If we describe INVITE collaborations that also include softphones as agent instances, softphone instances will always occur together with either a user agent client or -server (this depends, whether the phone will initiate or receive a call). If we bind such agent instance pairs in the same replica, the relation between the instance sets will preserve their unity for later finding.

⁸ Note that the term “segment replica” refers to a copy of all behavior segments that are contained in a replica.

2.5 CoSDL macros

The collaboration topologies described by our example specifications allowed for at most two proxy servers on the path between a user agent client and a user agent server. In reality, the INVITE collaboration can have any number of intermediate proxies. In the following, we will explain how to specify a chain of identical proxy servers that forward the call request from a user agent client to a user agent server.

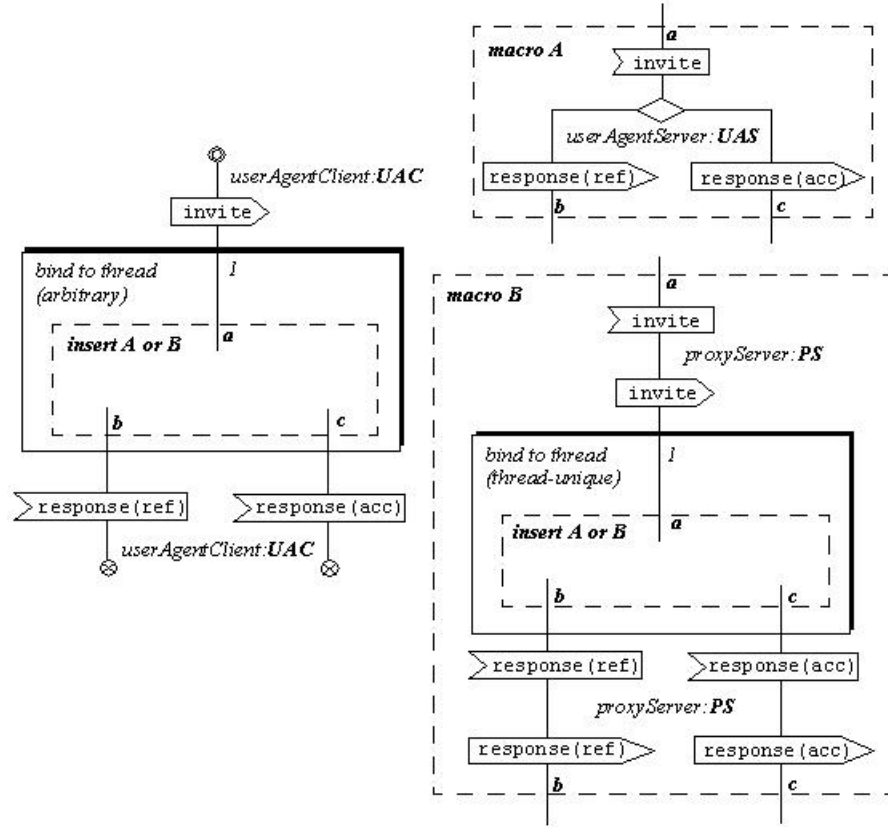


Figure 10. INVITE collaboration: alternative scenario and cascaded proxies

Similar to Figure 3, we can already manage to specify any fixed number of intermediate proxies, but for an indeterminate number of proxies, we need a recursive description. Consider the CoSDL diagram of Figure 10, which adds multiple intermediate proxy instances to the call setup procedure. The behaviour segments with a dashed border (on the right side) are substitutes (*macros*) for dashed areas elsewhere in the specification (*macro placeholders*). Macros are marked with the keyword *macro*, while macro placeholders are identified by the keyword *insert*. Macros are named, so that the places where they are to be inserted can be clearly indicated. In Figure 10, the macro placeholder on the left part can be replaced by macro A or macro B. Macro insertion is pictorial, i.e., macros are inserted as is. The insertion points are marked by small letters such as *a*, *b*, or *c*. If macro B is inserted, there will be again a placeholder for insertion. This can be continued recursively. As CoSDL abstracts from data aspects, the concrete number of recursion steps is not specified. The possible topologies specified this way include the case where the calling parties are directly connected as well as any number of intermediate proxies between. Note that SIP explicitly provides for the case where user agent client and user agent server connect directly. If we inserted macro B in the left placeholder first and then apply macro A in order to fill the placeholder created by macro B, a similar topology to Figure 2 would emerge. It should be mentioned that the binding of a different proxy instance for each recursive step is actually accomplished by the replicas. The modifiers make sure that loops in the proxy sequence cannot occur. Without replicas, the

pictorial macro insertion would yield a collaboration where only one proxy is involved, which sends itself first a couple of `invite` messages and then (depending on the user agent's answer), it sends itself the same number of `response(acc)` or `response(ref)` messages.

The CoSDL specification of Figure 10 is considered one collaboration diagram, i.e., it is only one collaboration specified. However, it includes many possible collaboration topologies during runtime of the collaboration.

The example illustrates two different uses of macros: first, we can express multiple binding through recursion. We did this in the above example by repeated insertion of macro B. Second, alternative topologies can be specified. This capability is not illustrated by the above example, but it is a simple concept: in SIP, a call request could also be answered by a redirect server instead of a user agent server. This alternative could be easily specified in Figure 10 by adding a third macro C that describes the behaviour of a redirect server. Whenever it is possible to insert macro A (the user agent server), we could use macro C (the redirect server) as an alternative. Note that this is different from the decision operator \diamond , because macro alternatives are not decided by an agent instance.

3 Conclusion

In this paper, we have presented CoSDL, an experimental language for specifying collaboration modules. A collaboration module describes a distributed functionality by encapsulating the required interaction behaviour of all involved agents.

Several notations including MSC, UML sequence diagrams, and UML collaboration diagrams have been advocated for scenario modelling. CoSDL differs from these notations in several respects:

- Unlike CoSDL, these notations focus on one particular sequence of interactions between a fixed set agents in one particular situation (with HMSC, alternatives and repetitions can be specified, too).
- With the exception of HMSC, no control structure can be specified. All events of a given instance are attached to a vertical axis or ordered by a numbering scheme. Therefore, collaborations can not be completely specified even if several diagrams are stated, but only a fixed set of scenarios. HMSC goes one step further by offering alternatives and repetitions, however, control structure and message events are separated into different diagrams. CoSDL also offers alternatives and repetitions, and in addition, support for forking and joining message flows.
- The CoSDL concept of collaboration threads provides a higher level of abstraction as compared to message exchange by grouping related interactions. Furthermore, a collaboration instance may consist of a set of collaboration, which can be understood as a form of composition on this abstraction level.
- CoSDL incorporates a type concept for collaborating agents as well as a way to describe instantiations and dynamic agent topologies. This makes collaboration modules complete and self-contained.

CoSDL can be seen as a companion notation to SDL that structures systems in a complementary way. While CoSDL captures collaboration modules, SDL focuses on process modules. In our work so far, we have sketched a method for transforming CoSDL specifications to SDL descriptions. Our notation has proven useful in several projects, including the collaboration-based redesign of the session initiation protocol SIP partially presented in this paper. We believe that the lessons learned from CoSDL are valuable for collaboration-oriented extensions of MSC, SDL, and corresponding tool environments.

References

1. G. Booch, J. Rumbaugh, I. Jacobsen: *The Unified Modelling Language User Guide*, Addison-Wesley, 1999
2. R. J. A. Buhr, R. S. Casselman: *Use Case Maps for Object-Oriented Systems*, Prentice Hall, 1996
3. M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg: *SIP: Session Initiation Protocol*, RFC 2543bis-02, IETF, 2000
4. F. Röbler, B. Geppert, and R. Gotzhein: *Collaboration-based Design of SDL Systems*, 10th SDL Forum, 2001
5. F. Röbler and B. Geppert: *Collaboration-Based Design - Exemplified by the Internet Session Initiation Protocol (SIP)*, 1st Working IEEE/IFIP Conference on Software Architecture, 2001
6. F. Röbler: *Collaboration-Based Design of Communicating Systems with SDL*, PhD thesis, University of Kaiserslautern, Germany, 2002
7. ITU-T Recommendation Z.100 (11/99) – *Specification and Description Language (SDL)*, International Telecommunication Union (ITU), 1999
8. ITU-T Recommendation Z.120 (11/99) – *Message Sequence Chart (MSC)*, International Telecommunication Union (ITU), 1999