Deriving component designs from global service and workflow specifications

Gregor v. Bochmann

School of Information Technology and Engineering (SITE)
University of Ottawa, Canada
bochmann@site.uottawa.ca

Abstract. This paper is concerned with the early development phases of distributed applications, service compositions and workflow systems. It deals with the transformation of a global requirements model, which makes abstraction from the physical distribution of the different system functions, into a system design that identifies a certain number of distributed components. At this design level, some of the global activities may be seen as collaborations among several components. The temporal constraints of the global requirements on the execution of the different activities imply certain coordination messages between the different system components. The paper presents a transformation algorithm that derives, from a given global behavior, the local behaviors for each of the system components including the exchange of coordination messages for the global synchronization of the activities. The global behavior is defined in terms of standard sequencing operators that correspond to the concepts found in UML Activity Diagrams and similar formalisms; for sequential execution, weak and strong sequencing is distinguished. The derived component behaviors ensure that their joint execution satisfies the ordering constraints of the global requirements model, they avoid any possible race conditions, and introduce a relatively small number of coordination messages. In many cases, these messages are required anyway for carrying the dataflow which underlies the global requirements model.

Keywords. Distributed applications, workflow, model transformations, Activity Diagrams, component design, protocol derivation, distributed system design, Web Services, design derivation.

1 Introduction

Various kinds of system models can be used during the system development process. In this paper, we are concerned with the transformation from a global requirements model, which describes the functional behavior of a distributed system in an abstract manner, to a distributed system design where the different system components are identified and their behavior must be determined such that their interactions give rise to a behavior satisfying the global requirements model. This transformation is called *design synthesis* in Figure 1. At the design level, the behavior of the different system components are often modeled using communicating state machines or modeling languages such as SDL or UML State Diagrams. The translation from these models into implementation code can be largely automated.

We consider in this paper distributed applications, for instance systems providing communication services, workflow management systems, e-commerce applications, etc. Various notations have been proposed for defining the global requirement models for such system. We mention in particular UML Activity Diagrams, Use Case Maps (UCM), the Process Definition Language (XPDL) of the Workflow Management Coalition, the Business Process Execution Language (BPEL), and the Web Services Choreography Description Language (WS-CDL) developed by W3C. These different notations contain many common concepts, but also show important differences. They all have in common that the overall workflow behavior can be decomposed into several sub-activities, and further into sub-sub-activities. Most of these notations assume that the basic (primitive) activities in this behavior decomposition are activities that are allocated to a single system component within the architectural design of the system. However, for many of these applications, the basic building blocks of the behavior are activities that are actually collaborations between several system components, for instance a service operation between a client and a server. Therefore we have proposed to use the UML Collaborations as the basic building blocks for

constructing global requirement models [1]. Our approach was to use the sequencing operations of UML Activity Diagrams and use Collaborations as the basic activities; the temporal order among these collaborations is then defined by the flow relations of the Activity Diagrams (an example is discussed in Section 2.3).

Before the transformation into a design model, it is important to define the architectural design and to identify the different system components that are involved in providing the different functions of the system. For each of the primitive collaboration activities identified in the global requirements model, one has to determine which system component will implement each of the collaboration roles involved. This goes hand in hand with the allocation of system resources and is very important for obtaining the desired system performance characteristics. This question of what is the best architectural design, resource allocation and allocation of collaboration roles to different system components, in short "architectural choices", is not further developed in this paper. Instead, we concentrate here on the subsequent question: What should be the dynamic behavior of each of the system components in order to coordinate the activities in such a manner that the sequencing rules of the global requirements model will be satisfied.

We note that the same kind of question has been addressed by many papers during the last 10 years in a context where the global requirements are defined in terms of Message Sequence Charts (MSCs) or UML Sequence Diagrams. In this context one usually wants to describe the behavior of each system component in the form of a state machine. This approach encountered many difficulties; a review of these issues is included in [1]. In many cases, a given MSC execution scenario may only be realizable by the given set of components if at the same time other so-called *implied scenarios* would also be realized [16]. Furthermore, the distributed nature of the design often gives rise to so-called *race conditions* which means that certain messages may arrive before they are expected, or in a different order than expected [17].

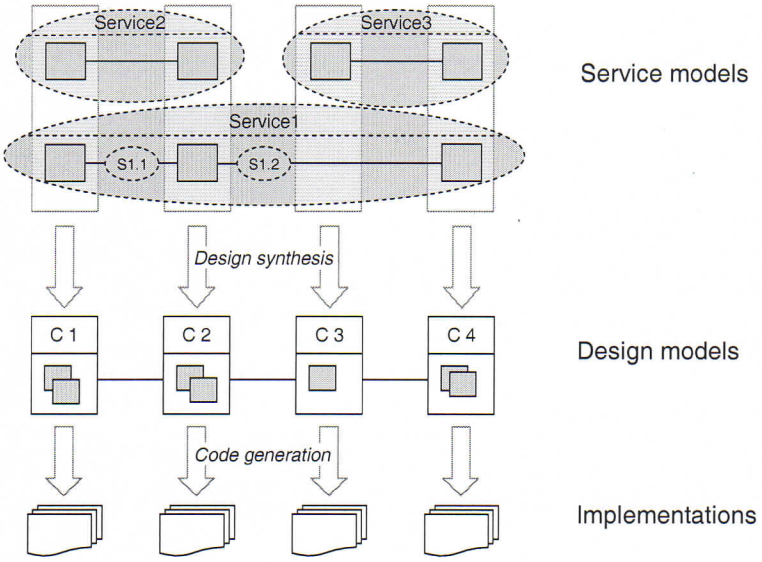

Fig. 1. Different types of system models (taken from [1])

These difficulties are increased by the use of **weak** sequencing operators in the description of the global system behavior. This is related to the partial-order semantics discussed in 1978 by Leslie Lamport [21]. **Strong sequencing** between two activities A1 and A2 means that all sub-activities of A1 must be completed before any sub-activity of A2 may start; this is the normal meaning of sequencing, also sometimes called global sequencing. In contrast, **weak sequencing** between A1 and A2 (only) means that each system component locally applies sequencing to the local sub-activities of A1 and A2, that is, a component may start with sub-activities that belong to A2 as soon as it has completed all its local sub-activities that are part of A1. Strong sequencing implies weak sequencing, but not inversely. In particular, if a component is not involved in A1, it may start with sub-activities of A2 even before A1 begins its execution. We note that weak sequencing was introduced in High-Level MSCs (HMSCs) as the normal sequencing operator between different sequence charts. It is also supported in UML Sequence and Interaction Overview diagrams.

We think that weak sequencing is an important concept for modeling abstract requirements of distributed systems, because it requires less synchronization messages than strong sequencing. Therefore we consider in this paper weak and strong sequencing. We use a number of temporal ordering operators, similar to those found in Activity Diagrams, XPDL and BPEL, to build the global requirements model for a system. In this paper, we show how such an abstract model, together with the allocation of collaboration roles to the system components identified by the architectural design, can be automatically transformed into a set of component behavior models. These component models are correct by construction, that is, they will give rise to a global system behavior that satisfies the global requirements model.

The transformation algorithm presented in this paper is inspired by some of our early work under the title "Deriving protocol specifications from service specifications" in the 1980ies [3, 4, 5, 6], where we concentrated our attention on strong sequencing. The main contribution of this paper is the extension of the previous work to requirement specifications that contains weak sequencing. Some inspiration also came from my collaboration with Humberto Nicolás Castejón and Rolv Bræk on the modeling of distributed applications using the concept of collaborations [1, 2] and the discussion of problems that must be solved during the development of the component behaviors. In this context, much attention was given to weak sequencing.

The paper is structured as follows. After a short overview of important concepts for describing the temporal ordering of activities in a global requirements model in Section 2.1, we present in Section 2.2 the ordering operators that we assume in this paper for describing the global requirement model of a distributed system. In Section 2.3, a simple example of a telemedicine application is presented.

The main body of the paper is Section 3. After a review of past work on the transformation from global requirements to component behaviors in Section 3.1, we describe in Section 3.2 the principles of our automatic transformation approach. One important question, not addressed by the earlier work mentioned above, is the following: In the case of choices, it is not evident how a component involved in some specific sub-activity may determine when this sub-activity is completed and the next sub-activity (in weak sequence) may be started ? – This problem is solved by the so-called *choice indication messages*, introduced in Section 3.2.

After the explanation of the general approach to the derivation of the component behaviors, we present in Section 3.3 an algorithm that does this transformation automatically. Then some examples are discussed in Section 4, including the telemedicine example of Section 2.3.

Section 5 comments on the correctness of the derivation algorithms and gives an outlook into the inclusion of data flow considerations, and generalizing our automatic transformation to the case that more general flow relations exist in the global requirement model, as may occur in descriptions given in the form of UML Activity Diagrams or concurrent BPEL behaviors including so-called "links". Finally, Section 6 provides our conclusions.

2 Describing composed collaborations and work flow applications

2.1 Review of notations for describing the dynamic behavior of collaborations and workflow applications

As mentioned above, various notations have been proposed for describing global requirements for distributed applications, workflows, or communication services. We consider here in particular UML Activity Diagrams (AD). They include the following concepts for defining the order of execution of activities: sequential execution, alternative choice, concurrency, as well as loops and partial-order dependencies. In addition, they support interruptible regions of activites which are useful for modeling exception handling and external priority interrupts. ADs also support the explicit modeling of dataflow relationships between different activities and the specification of the type of data exchanged (using UML Class Diagrams). XPDL and BPEL have similar constructs for describing the control flow of applications. In the case of BPEL, partial-order dependencies are modeled by so-called links. All these notations support hierarchical decomposition where an activity shown at one level of abstraction as a basic, non-divisible activity can be described at a more detailed level to be composed out of a number of smaller units with a specific control structure defining the order of execution of these more basic activities. It is our intention to

support these same concepts for describing the control structure using the notation introduced in the next subsection 2.2; note that arbitrary partial-order dependencies are discussed in Section 5.

We note that most of these notations assume that the basic (primitive) activities in this behavior decomposition are activities that are allocated to a single system component within the architectural design of the system. This is also the case for Message Sequence Charts (MSCs) or UML Sequence Diagrams, where the primitive actions are the sending or reception of messages by specific system components. However, for many applications, especially at a higher level of abstraction, the basic activities in the description of the behavior are often collaborations between several system components, for instance a service operation between a client and a server. Therefore we have proposed to use the UML Collaborations as the basic building blocks for constructing global requirement models [1]. As shown in the example discussed in Section 2.3, we use the control flow operations of ADs and use Collaborations as the basic activities.

It is important to note that at a high level of abstraction the global requirements model of an application may not be concerned with the architectural structure of the system and its decomposition into physical components. However, already ADs and UCMs foresee notations for introducing the concept of system components (called "swim lanes" in the case of ADs). In this paper, we use for this purpose an allocation function, called Alloc(), which maps the roles of the collaborations that are part of the global requirements model, onto the set of system components that are supposed to be known.

In this context one may ask the question whether different notations are required for describing the dynamic behavior of the global requirements model, on the one hand, and the behavior of the different system components, on the other hand. For instance, BPEL appears to be designed for describing the behavior of a particular system component participating in an e-commerce application, while WS-CDL is intended to describe a global system behavior. We also note that BPEL, as its name says, is an "execution language" which appears to be intended for implementation, and not so much for describing a requirements model. In this paper, we use the same control flow operations for describing the behavior of the global requirements model and the behaviors of the different system components which are derived from the former. However, the distinction between weak and strong sequencing disappears when one deals with the behavior of a single component.

2.2 Operators for well-structured behavior descriptions

We use in this paper a set of sequencing operators that are closely related to the sequencing operators of UML Activity Diagrams and High-Level MSCs. We distinguish between strong and weak sequence. Following the spirit of "Structured Programming", we restrict ourselves in the main part of this paper to flow control constructs that have a single entry point and a single exit point.

We assume that the behavior of a collaboration is described by an expression C which is build out of the operators shown in Table 1. We adopt the following notation to define the behavior of a collaboration with name <name> and which involves a set of roles R:

$<name>^{(R)} = C$.

The expression C may include the invocation of the execution of other collaborations, including itself. This means that a behavior description may include several levels of abstraction, possibly recursively. It is also possible to invoke a collaboration that has no explicitly defined behavior; in this case, its behavior may be defined by some other formalism, such as a sequence diagram or an implementation in some programming language.

A behavior expression C is composed out of primitive actions, the invocation of collaborations and certain sequencing operators, as defined in Table 1. We refer to the sub-expressions C1, C2, and C3 in the table as sub-collaborations of the collaboration C.

We note that our notation does not include the equivalent of the Join and Merge operators used in Activity Diagrams. However, the presence of a Merge node is implied at the end of a choice expression, and a Join node is implied at the end of the concurrency construct.

Table 1: Operators used in behavior expressions

Construct	Notation	Explanation of the semantics
primitive activity	<action>$^{(r)}$	Execution of a local action with name <action> performed by role r
invocation of a sub-collaboration	<subcol>$^{(R)}$	Execution of a collaboration with name <subcol> involving the set of roles R (the set of participating roles)
strong sequence	$C_1 ;_s C_2$	C_2 is executed after C_1 in **strong** sequence, that is, all actions of C_1 are completed before C_2 can start
weak sequence	$C_1 ;_w C_2$	C_2 is executed after C_1 in **weak** sequence, that is, only local order is enforced by each participating role
choice	$C_1 [] C_2$	Either C_1 or C_2 is executed; this may be a **local choice** (that is, the choice is performed by a single role / component) or or a non-local choice, such as competing initiatives from several roles (for a more detailed discussion, see [2])
strong while loop	$C_1 *_s C_2$	C_1 is executed zero, one or more times and then C_2 will be executed; more precisely, the behavior starts with a choice between C_1 and C_2 ; if C_1 is executed, there is **strong** sequencing between the end of C_1 and the choice of executing C_1 again or terminating the loop with C_2 . We assume that the choice is local (performed by the starting role of C_1) and that C_2 is empty or is started by the same role as C_1.
weak while loop	$C_1 *_w C_2$	As above, except that **weak** sequencing is used between the end of C_1 and the choice of executing C_1 again or terminating the loop with C_2
concurrency	$C_1 \| C_2$	C_1 and C_2 are executed concurrently
interruption	$C_1 \|> C_2$ else C_3	C_1 is executed, but may be interrupted by C_2 which represents a choice with priority; C_2 is enabled as soon as C_1 starts. If C_2 does not occur during the execution of C_1 (or occurs when C_1 is already terminated) then C_3 will occur after C_1 (this is the other choice alternative); thereafter the two choices join.

2.3 An example

We consider the example of the telemedicine consultation service described in [1]. A patient is being treated over an extended period of time for an illness that requires frequent tests and consultations with a doctor at the hospital to set the right doses of medicine. Since the patient may stay at home and the hospital is a considerable distance away from the patient's home, the patient has been equipped with the necessary testing equipment at home. The patient will call the hospital on a regular basis to have remote tests done and consult with a doctor. A consultation may proceed as follows: The patient calls the telemedicine reception desk to ask for a consultation session with one of the doctors. The receptionist will register the information needed, and then see if the doctor is available. If the doctor is available, the patient will be assigned to the doctor and the consultation can start. Otherwise, the patient is put on hold, possibly listening to music, until a doctor is available. If the patient does not want to wait any longer, he/she may hang up (and call back later). The consultation itself is defined by a separate diagram and consists of a voice communication collaboration during which certain tests are invoked using some test equipment at the patient's location, but controlled by the doctor over distance.

This behavior is shown in Figure 2, where each activity is shown as a rounded box and the roles involved in each activity are shown (P stands for patient, R for receptionist, D for doctor, and S for the test equipment). In addition, the roles starting and terminating the activity are indicated: a dot indicates a starting role, a vertical bar a terminating role. As explained in [1], these activities are in fact collaborations, since they involve generally several participating roles.

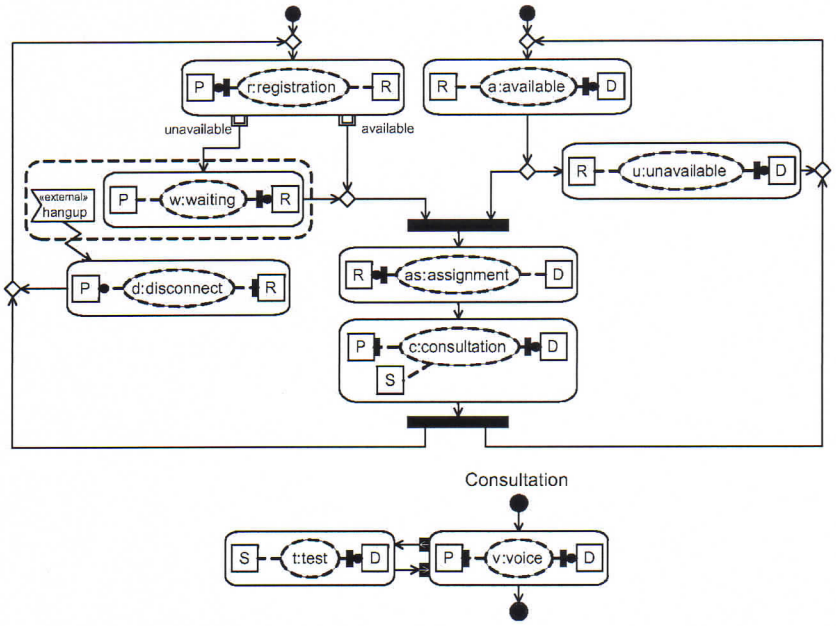

Fig. 2. Order of collaboration activities within the telemedicine example application (taken from [1])

A simplified version of this behavior can be described using the operators defined in Table 1 as follows:

<telemed> = <registr>{$_sP_t$, R}

$;_w$ ((<wait>{P_t, $_sR$} $*_w$ ε) |> <h-up>{$_sP_t$} else (<assign>{$_sR_t$, D} $;_w$ <consult>{P_t, $_sD_t$}))

The roles involved in each sub-collaboration are indicated in brackets. This definition of the <telemed> workflow indicates that the registration of the patient (action <registr>) is followed by a waiting period that may be empty (ε); this waiting period may be interrupted when the patient hangs up the telephone (action <h-up>). The waiting period, if not interrupted, is followed by the assignment of the patient to the doctor, which is initiated by the receptionist (as indicated in Figure 1), and by the consultation, which is initiated by the doctor. We note that the detailed interactions involved in each of these sub-collaborations are not specified, and we do not need to know them for what we discuss in this paper. The indexes s and t next to the roles indicate the starting and terminating roles, respectively, as explained in Section 3.2.

We may also introduce the sub-behaviors <w> ("waiting period") and <act> ("actions") as follows:

<w>{P, R} = <wait>{P_t, $_sR$} $*_w$ ε and

<act>{P, R, D} = <assign>{$_sR_t$, D} $;_w$ <consult>{P_t, $_sD_t$}

Then the definition of <telemed> can be rewritten as

<telemed> = <registr>{$_sP_t$, R} $;_w$ (<w>{P, R} |> <h-up>{$_sP_t$} else <act>{P, R, D})

The behavior can also be represented by the UML Activity Diagrams shown in Figure 3.

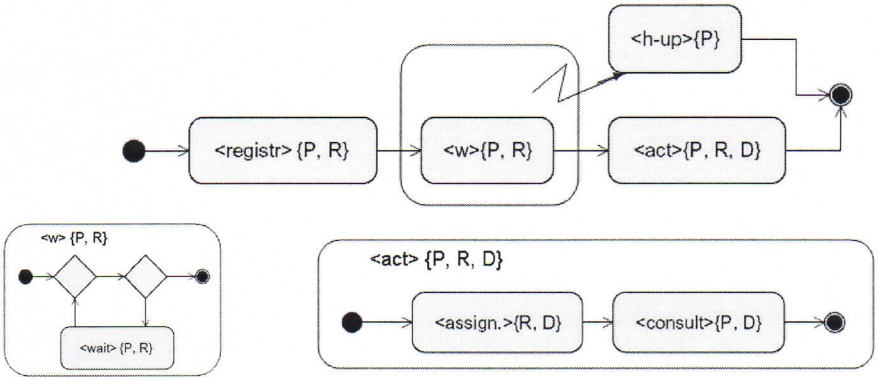

Fig. 3. Dynamic behavior of the Telemedicine collaboration, including two sub-collaborations

3 Deriving component-based designs

3.1 Previous work

Our early work in this area [3, 4, 5] covered behavior expressions containing primitive actions, invocations of behaviors without recursion, strong sequence, choice and concurrency. Coordination messages were introduced for a strong sequence "C1 ;s C2" to ensure that all activities of C1 are completed before any activity of C2 can start. The various messages introduced by the derivation algorithm included a parameter that avoided any ambiguities concerning the choices that were made during the execution of the behavior. A later paper [6] dealt with recursive behavior invocations and interruption.

In the above references, it was assumed that a choice between different branches of execution is always made by a single component. This is called a "local choice". In the case of a "non-local choice" where several components are involved [7], distributed algorithms for making a decision may be introduced, for instance, based on a circulating token. Gouda showed in 1984 [8] how a choice involving competing initiatives from two different components may be resolved by giving priority to one of the parties.

During the last 10 years, much research was concerned with weak sequencing and related race conditions. Most of this work was in the context were the system behavior is defined in terms of MSCs or Sequence Diagrams; and it was pointed out that one sequence diagram, when implemented by a set of components, may necessarily give rise to other so-called "implied sequences" [14]. The difficulties of coordination for distributed behaviors including weak sequencing have been summarized in [2]. An interesting observation was made by Mooij [9, 10] who points out that many race conditions can be avoided by making a distinction between the reception of a message by a component and the consumption of this message by the behavior of a role played by this component. He assumes that received messages are put into a buffer pool from where appropriate messages may be fetched when the destination role is ready to process them. A similar idea is the use of the SDL SAVE construct to reorder the sequence of received messages [15].

In the area of Web Services and workflow management, several approaches have been described for deriving distributed execution environments for services/workflows that are specified in a global (centralized) view. For instance, the decentralized execution of composite Web Services specified in BPEL has been proposed in [18]. Here the global BPEL specification is partitioned into small code fragments which are then combined (based on data flow relations) into several local partitions that are executed on the different servers identified in the original BPEL specification. The resulting implementation is in general more efficient since the number of required messages is reduced. A proposal for workflow fragmentation and distributed execution [19] is also based on data flow and uses a variant of Petri nets for describing the workflow to be performed. The workflow is partitioned into fragments of which the first is executed locally and the others may be distributed to other servers. The choice of these servers can be performed dynamically during the execution of the current fragment. A theoretically oriented paper [22] considers a formalization of WS-CDL for the specification of the global behavior and the π–calculus for the behaviors of the components. We note that these approaches consider that the basic activities to be performed can be allocated to a single component (server). Therefore they cannot deal with the more general situation considered in this paper, where the basic activities are collaborations that may involve, each, several collaborating components.

3.2 Proposed derivation method

The derivation method described here uses the following ideas described previously: (a) coordination messages for strong sequencing [3, 4, 5], (b) the idea that messages should have an identifier that indicates to which sub-expression of the behavior expression they belong (particular methods of obtaining such an identifier were proposed by Nakata [11] and for Application Protocols in the ASN.1 standard), and (c) the idea of buffering received messages until they are processed, as proposed in [9, 10]. The proposed derivation method extends the previous work by providing a method to deal with weak sequencing. It also introduces the treatment of loops and a particular form of interruption. For the treatment of non-local choices, the reader is referred to [2].

3.2.1 General ideas

The main ideas underlying the proposed derivation method, specifically for dealing with weak sequencing, can be summarized as follows:

1. Each role knows which sub-collaborations are currently active. Message re-ordering at reception is used to accept only those messages that relate to active collaborations.
2. It is assumed that sub-collaborations that may be concurrently active have disjoint sets of messages that can be received by a given role; or simply that their message sets are disjoint.
3. At a given role, each sub-collaboration is in one of the following phases: (1) **inactive** (messages for this sub-activity are not accepted), (2) **enabled** (the role is not a starting role, the messages of this sub-activity are accepted), (3) **active** (local activities for this sub-activity have started, messages are accepted). We say that a sub-collaboration **ends** when the role knows that no further actions pertaining to this sub-collaboration are required. When a sub-collaboration ends, it goes back into the **inactive** phase.
4. The transition from *inactive* to *enabled* occurs when the "previous" sub-collaboration ends. If the role is a starting role, it may immediately go into the *active* state. A non-starting role enters the *active* state when it receives (and accepts) the first message pertaining to this sub-collaboration. When the sub-collaboration *ends*, the "following" sub-activity goes into the *enabled* or *active* state.
5. It is therefore important that each role knows when an active collaboration ends. This happens when the final action (for this role) is performed. If the role is participating, it should know when all actions pertaining to this sub-collaboration have been performed (**termination decision**). If it is not participating, there is no point in doing anything.
6. If the sub-collaboration contains no choice, then each role knows what actions must be locally performed. The *termination decision* is easy: the sub-collaboration ends when all these actions have been performed. If the sub-collaboration consists of a choice C1 [] C2 there are the following cases:
    - The role participates in both alternatives: choice propagation is assured by the disjointness of the message sets of the two alternatives. The participating role will know which alternative is performed and will therefore know which actions must be performed.
    - The role does not participate in any alternative: there is no participation at all.
    - The role participates in C1 but not in C2 (or inversely): If C1 is chosen, there is no problem. If C2 is chosen, we have to introduce a special kind of coordination message sent to this role by a role participating in C2 which indicates that C2 was chosen. We call this message a *choice indication message*. On the reception of this message, the given role can consider that the choice has ended.

3.2.2 Starting and terminating roles

The above discussion indicates that we have to identify for each collaboration or sub-collaboration the following items which are defined based on the partial order between the actions that compose the collaboration:

- **Special kinds of actions of a collaboration**
    - *Initial action(s):* An action of a collaboration is initial if there is no other action in that collaboration that precedes it.
    - *Final action(s):* An action of a collaboration is final if there is no other action in that collaboration that succeeds it.
    - *Last action(s)* of a given role: During the execution of a collaboration, an action is a last action for a given role if the action is performed by that role and there is no other action in that collaboration that must be performed by that role after the given action.
- **Different roles involved in a collaboration**
    - *Starting* role: this is a role that performs an initial action of the collaboration or an initial action of an initial sub-collaboration.
    - *Terminating* role: this is a role that performs a final action of the collaboration or a final action of a final sub-collaboration.
    - *Participating* role: this is a role that executes a primitive action of the collaboration or of a sub-collaboration. This includes the starting and terminating roles.

Table 2 shows how the sets of starting, terminating and participating roles are calculated for a behavior expression depending on the sequencing operators used within the expression.

Table 2: Rules for calculating the starting, terminating and participating roles of a collaboration

Construct	Notation	Starting roles (SR)	Terminating roles (TR)	Participating roles (PR)
primitive activity	<action>$^{(r)}$	{r}	{r}	{r}
invocation of a sub-collaboration	<subcol>$^{(R)}$	SR(<name>)	TR(<name>)	PR(<name>) = R
strong sequence	C_1 ;$_s$ C_2	SR(C_1)	TR(C_2)	PR(C_1) U PR(C_2)
weak sequence	C_1 ;$_w$ C_2	SR(C_1) U (SR(C_2) - PR(C_1))	TR(C_2) U (TR(C_1) - PR(C_2))	PR(C_1) U PR(C_2)
choice	C_1 [] C_2	SR(C_1) U SR(C_2) (for a local choice, this should be a single role)	TR(C_1) U TR(C_2) Note: In general, this set is larger than necessary, since only one of the alternatives will be executed.	PR(C_1) U PR(C_2)
strong while loop	C_1 *$_s$ C_2	SR(C_1) = SR(C_2) = {r} (assumption: single starting role)	TR(C_2) Note: In the case that C_2 is empty, the terminating role is SR(C_1), which does the final choice of terminating the loop.	PR(C_1) U PR(C_2)
weak while loop	C_1 *$_w$ C_2	*same as for strong while loop*	TR(C_2) U (TR(C_1) -PR(C_2)) (the second part corresponds to the case that C1 is executed at least once; in the case that C_2 is empty, we obtain SR(C_2) U TR(C_1))	PR(C_1) U PR(C_2)
concurrency	C_1 || C_2	SR(C_1) U SR(C_2)	TR(C_1) U TR(C_2)	PR(C_1) U PR(C_2)
interruption	C_1 |> C_2 else C_3	SR(C_1)	TR(C_2) U TR(C_3)	PR(C_1) U PR(C_2) U PR(C_3)

In the example given in Section 2.3, the particating roles are indicated in the brackets after the name of a sub-collaboration, and the starting and terminating roles are identified by an index s in front of the name of the role, and by an index t after the name, respectively. For instance, the notation <consult>{P_t, $_sD_t$} means that for the sub-collaboration <consult>, the role D is the starting role and both roles, P and D are terminating roles. Using the rules of Table 2 and the information about participating, starting and terminating roles given in Section 2.3 for the sub-collaborations <registr>, <wait>, <h-up>, <assign> and <consult>, we can determine these roles for the composed sub-collaborations <w> and <act> and finally for the whole <telemed> behavior. For instance, the <act> behavior is formed by the weak sequencing of <assign> and <consult>, and rule in Table 2 determines that the only starting roles of <act> is R, and all roles R, D and P are terminating roles. For <w>, the starting role is R (the starting role of <wait>) and the terminating role is also R (because the sub-collaboration C_2 is empty, see note in Table 4). Similarly, one finds that the starting role of <telemed> is P (the starting role of <registr>) and the terminating roles are P, R and D (those of <act>).

3.2.3 Architectural choices

Before we can derive the behavior of the distributed system components that should implement the actions defined by the collaboration behavior, we have to determine how the different roles defined in the behavior of the collaboration are allocated to the different system components. In general, each system component should have some role to play, but several behavior roles may be allocated to the same system component. We assume in the following that a function Alloc() defines for each role the system component to which it is allocated. The choice of the system components and the mapping of the Alloc() function has clearly a strong impact on the performance of the system implementation. How to make these architectural choices is not discussed in this paper. We simply assume that the allocation function is given. Since the distributed system design for a given architectural choice can be automated by the method described in this paper, it is also conceivable that several prototype implementations would be developed for different architectural choices in order to later select the one that is most appropriate.

We note that in this paper we assume that the set of components is statically known. We also assume that each component is initialized to be ready to participate in the given collaboration. For many applications this assumption is not satisfied. For example, the following situations may apply:

- Dynamically created role instances: A server providing the service corresponding to a given role will usually create a new application instance (applet or process) for each user that invokes the given collaboration. A simple example is a FTP server. To distinguish the different instances of the same collaboration, one usually introduces the notion of a session, together with session identifiers to distinguish the messages belonging to different sessions, or separate transport (TCP or SSL) connections for transmitting the messages for each session.
- Dynamically linked component instances: In Web-Services applications, the server to be invited to participate in a given collaboration is often dynamically chosen by the initiator of the collaboration. For example, directory services may be used to identify a particular server that has appropriate quality of service parameters corresponding to the non-functional requirements of the initiator.

We believe that the design derivation method described in this paper can be adapted to these more general situations.

3.2.4 Coordination messages

After having calculated the starting (SR), terminating (TR) and participating (PR) roles for the collaboration and each of its sub-collaborations, we then can derive the behavior for each system component as follows. Basically, the control flow of the behavior of each system component follows the control flow of the collaboration behavior; it is obtained from the global behavior specification of the collaboration "by projection" onto the particular component. This means that actions not local to the component in question are dropped. Therefore any sub-collaboration for which no participating role is allocated to the component in question will also be dropped.

In addition, the following coordination messages between different system components are introduced, as explained in Table 3:

- Coordination messages for selecting a choice alternative in the case of non-local choice; (this is not further considered in this paper)
- *Flow message* for coordinating strong sequencing, abbreviated fm(x) or fim(x, i); each message includes a parameter x which indicates to which strong sequencing construct the message belongs within the syntactical structure of the overall collaboration behavior, as originally proposed in [3].
- *Choice indication message* for propagating the choice to a component that does not participate in the selected alternative, abbreviated cim(y) where y indicates to which choice construct the message refers; note that such a message is only required if the destination component is involved in some activities following the choice.
- *Interrupt and interrupt enable messages* for coordinating the interruption of an ongoing activity, abbreviated im(z) and iem(z), respectively, where z indicates to which interrupt construct the message refers.

It is noted that the rules of Table 3 sometimes imply the introduction of a coordination message from some component X to itself. Clearly, such messages are superfluous and are avoided by the algorithm defined in Table 4.

We note that instead of defining the while loop of the form " $C_1 *_s C_2$ " one could have used a simpler form of a while loop " $C_1 *^s$ " which would be equivalent to "$C_1 *_s \varepsilon$ " ; then $C = C_1 *_s C_2$ could be rewritten as $C = (C_1 *^s) ;_s C_2$. The reason for using the more complex form of the while loop is that it reduces the number of coordination messages required in the case that, for instance, C_1 and C_2 involve the same components. In this case, the choice indication message under point (a) in Table 3 are not required, whereas they are required for the expression " $C_1 *^s$ " (since it is equivalent to "$C_1 *_s \varepsilon$ " and all components not starting C_1 should receive such a message to be informed about the termination of the loop).

Table 3: Required coordination messages

Construct	Notation	Required coordination messages
Primitive act	<action>$^{(r)}$	The action is included in the behavior of the component Alloc(r). - No message.
invocation of a sub-collab.	subcol$^{(R)}$	The invocation is included in the behaviors of the components in Alloc(R). - No message required.
strong sequence	C_1 ;$_s$ C_2	A *flow message* is sent by each component in Alloc(TR(C_1)) (when the execution of C_1 is completed) to all components in Alloc(SR(C_2)); these components will receive these flow messages before starting the execution of C_2. This was already described in [3, 4, 5]. We note that in the case that C_1 contains alternatives, the set Alloc(TR(C_1)) may be larger than necessary (see note in Table 2). Therefore we may introduce more flow messages than necessary. The approach described in [3, 4] avoids such unnecessary messages.
weak sequ.	C_1 ;$_w$ C_2	No coordination messages required.
choice	C_1 [] C_2	(a) If the choice is not local, that is, if Alloc(SR(C_1) U SR(C_2)) contains more than one component, then coordination messages for organizing the choice are required. (b) If a component is in Alloc(PR(C_1)) but not in Alloc(PR(C_2)), then the component should receive a message from one of the components in Alloc(PR(C_2)) when choice C_2 is performed (this is a *choice indication message*). The component may then enable the sub-collaboration that follows the choice.
strong while loop	C_1 *$_s$ C_2	(a) If a component is in Alloc(PR(C_1)) but not in Alloc(PR(C_2)) and is not the starting component of C_1, then it should receive a message from the latter when C_2 is performed (this is a *choice indication message*) or from one of the components in Alloc(PR(C_2)). (We note that this is not required if the component has no starting role for the collaboration(s) that follow the strong while loop, because it would be invited for following collaboration by a message). (b) As in the case of strong sequencing, there are *flow messages* sent, when C_1 ends, from the components in Alloc(TR(C_1)) to the component starting C_1.
weak while loop	C_1 *$_w$ C_2	As above, however, the *flow messages* under (b) are not required. In addition, if a component is in Alloc(PR(C_1)) and in Alloc(PR(C_2)), then the message starting the execution of C_2 in this component should contain a counter indicating the number of times C_1 was executed. The same parameter should be included in the choice indication messages under (a). Note: this allows delaying the consumption of this message until a sufficient number of messages concerning C_1 have already been consumed; this avoids the problem shown in Figure 9(c) of [1].
concurrency	C_1 || C_2	No coordination messages required.
interruption	C_1 |> C_2 else C_3	(a) We assume that C_2 has a single starting role, and is of the form " <action>$^{(r)}$;$_s$ C_2' ". (b) When a starting role of C_1 starts the execution of C_1, it sends an *interrupt enable message* to the starting role of C_2; when such a message is received, C_2 becomes enabled. (c) When the interrupt action <action>$^{(r)}$ occurs, the component performing role r will send *interrupt messages* to all components in Alloc(PR(C_1)); when these components receive this message, they will know that an interrupt occurred. If they have not yet completed the execution of C_1 they will be in the "interrupted" state. (b) *Flow messages* will be sent by all components in Alloc(PR(C_1)) to all starting components of C'$_2$ and C_3. These messages include a Boolean parameter indicating whether the component was "interrupted". The starting components of C'$_2$ and C_3, when they have received these messages, will know whether C'$_2$ or C_3 should be executed (C_3 will be executed only if no component was "interrupted").

3.3 An algorithm for deriving the component behavior from the global behavior expression

In the following, we define an algorithm that realizes the derivation method explained in the last sub-section. We assume that the overall workflow is defined by a "main" collaboration and several sub-collaborations, identified by their name, which are invoked by the main-collaboration or some activated sub-collaboration. Each of these collaborations is defined by a behavior expressions C which is formed by primitive actions, sub-collaboration invocations and the operators introduced in Table 1. For each of the components c that implements the roles of these collaborations, we define in the following a translation functions T_c that translates the behavior expressions of the collaborations into local behavior expressions to be performed by the component in question. These local behavior expressions will include those primitive actions of the collaborations that are performed by the component in question, in addition to the sending and receiving of coordination messages as required by the behavior expressions. Overall, the syntactic structure of the resulting behavior expressions for all these components resembles the syntactic structure of the original expression of the global collaboration behavior.

In the following we make the assumption that all choices are local. We note that certain standard approaches to solving non-local choices could be easily integrated with our derivation algorithm. However, as explained in [1, 2], the nature of non-local choices may vary a lot in practice and it appears necessary to allow for ad-hoc solutions to fit the specific requirements in particular cases

Table 4 contains the definition of the translation function T_c (C) that defines for a given global behavior expression C the behavior of the system component c. It is defined recursively by the rules in the table. The resulting component behavior expression is constructed using the same sequencing operators as for describing the global behavior, however, since the behavior is performed locally by a given component, there is no point in making a distinction between weak and strong sequencing. We simply use the operator ";" to denote sequential execution.

The text defining the translation function in the table uses a notation similar to Java Server Pages, namely a mixture of text that represents the generated specification of the component behavior, and of text that represents actions to be performed during the execution of the translation. The latter is written in italics. We also include some comments (written between "(*" and "*)") and notes for making the definition of the translation more readable.

As mentioned at the beginning of Section 3.2, the parameters of the coordination messages and the buffering of received messages before their consumption are important elements for the correct operation of the distributed system derived by the algorithm of Table 4. We make the following assumptions:

1. Each coordination message contains the following parameters: (a) source role, (b) destination role, (c) name of sub-collaboration it belongs to, (d) the particular sequencing operator instance it refers to within the global behavior expression of the given sub-collaboration – these are the parameters named x, y, and z in Table 4. As noted earlier, the parameters (c) and (d) above are important for non-ambiguous choice propagation (see also [3, 4, 5]). In addition, the messages also need addressing information in order to be transmitted through the network to the right computer and the responsible application.
2. The reception of coordination messages proceeds in two steps: When a message is received by a component, it is first placed into a buffer pool, called *receive-buffer*. It will be "consumed" from the *receive-buffer* only when the behavior expression generated for the component according to Table 4 foresees the execution of a receive statement for a message of the specific type and parameter values. The message parameters mentioned above, and the additional parameter mentioned under point 4 below are used to determine whether a message in the *receive-buffer* is "receivable". If no receivable message is in the *receive-buffer*, the execution of the local behavior will wait until such a message arrives.
3. The flow messages used within an interrupt construct, abbreviated fim(x, i), have an additional Boolean parameter i that indicates whether an interrupt was successful,.
4. The execution of a weak while loop, say $C_1 *_w C_2$, within the distributed environment may lead to situations where the component, say c_1, which makes the decision of the looping condition, may already have performed several iterations of C_1 while another component, c_2, may have only started the first iteration (as shown in Figure 9(c) of [1]). When c_2 receives a flow message indicating the beginning of C_2, it is important that c_2 can determine whether C_2, should be started or whether more executions of C_1 should first be performed. Therefore we include an additional parameter, say n, in all flow messages that are part of the coordination within C_2, which contains the number of times

Table 4: Definition of the translation function T_c for component c

Structure of expression	Definition of T_c
C = <action>$^{(r)}$	T_c (C) = *if Alloc(r) = c then* "<action>" *else* ε Note: ε is the empty string and means that no actions need to be performed.
C = invoke <subcol>$^{(R)}$	T_c (C) = *if c in Alloc(R) then* " invoke <subcol>" *else* ε
$C = C_1 ;_s C_2$	T_c (C) = T_c (C_1) ";" SFM(C_1 , C_2) ";" RFM(C_1 , C_2) ";" T_c (C_2) *where* SFM(C_1 , C_2) = *if c in Alloc(TR(C_1)) then* "send fm(x) to all c' in (Alloc(SR(C_2)) – {c})" *and* RFM(C_1 , C_2) = *if c in Alloc(SR(C_2)) then* "receive fm(x) from all c' in (Alloc(TR(C_1)) – {c} " Note: The term "– {c}" avoids that flow messages are sent to the component itself.
$C = C_1 ;_w C_2$	T_c (C) = T_c (C_1) ";" T_c (C_2)
$C = C_1$ [] C_2	T_c(C) = DOcim$_c$(C_1, C_2) [] DOcim$_c$(C_2, C_1) *where* DOcim$_c$(C_1, C_2) = *if c in Alloc(PR(C_1)) then* "(T_c (C_1)" *if c is responsible for cim then* "|| send cim(y) to all c' in (Alloc(PR(C_2)) - Alloc(PR(C_1))))" ; else *if c in (Alloc(PR(C_2) - Alloc(PR(C_1))) then* "receive cim(y)" Note: The function DOcim$_c$(C_1, C_2) generates code for performing C_1, and looks after the transfer of choice indication messages from some component participating in C_1 to those components not participating in C_1, but in C_2.
$C = C_1 *_s C_2$	We assume that Alloc(SR(C_1)) = {r} and that Alloc(SR(C_2)) = {r} or C_2 = ε. T_c(C) = "(" T_c(C_1) ";" SFM(C_1 , C_1) ";" RFM(C_1 , C_1) ")* ; (" T_c(C_2) *if c = r then* "|| send cim(y) to all c' in (Alloc(PR(C_1)) - Alloc(PR(C_2)) – {r}) " *if c in (Alloc(PR(C_1)) - Alloc(PR(C_2)) – {r}) then* "|| receive cim(y) from r " ")"
$C = C_1 *_w C_2$	As above, except that the SFM and RFM constructs are absent
$C = C_1 \| C_2$	T_c (C) = T_c (C_1) || T_c (C_2)
$C = C_1 \|> C_2$ else C_3	We assume that C_2 has the form " <action>$^{(r)}$;$_s$ C_2' ". T_c (C) = NormalBeh ||* InterruptBeh . (see note below) These two parts communicate within each component using the following boolean local variables which are initially false: Interr : an interrupt occured (but it may have occurred too late) Interrupted : the normal behavior has been interrupted In addition, a local variable I-Enabled is used by the InterruptBeh part. The action "wait(x)"waits until the expression x becomes true. **NormalBeh** = *if c in Alloc(PR(C_1)) then* "(T_c (C_1) |> (wait(Interr); Interrupted := true;) else ε);" *if c in Alloc(TR(C_1)) then* "send fim(x, Interrupted) to all c' in ((Alloc(SR(C'$_2$)) U Alloc(SR(C_3)))–{c});" (* this is similar to SFM(C_1, C'$_2$ || C_3) *) *if c in (Alloc(SR(C'$_2$)) U Alloc(SR(C_3))) then* "((* similar to RFM(C_1, C'$_2$ || C_3) *) (for all c' in (Alloc(TR(C_1))–{c}) do (receive fim(x, i) from c'; if i then Interrupted := true;); if not Interrupted then DOcim$_c$ (C_3, C'$_2$);) ||* (wait(Interrupted); DOcim$_c$ (C'$_2$, C_3))) " *else* " (DOcim$_c$ (C'$_2$, C_3) [] DOcim$_c$ (C_3, C'$_2$)); " **InterruptBeh** = *if c = r then (* *if c in (Alloc(SR(C_1)) then* "I-Enabled := true; " *else* "for all c' in (Alloc(SR(C_1)) – {c}) do (receive iem(z) from c'; I-Enabled := true) || (wait(I-Enabled); <action> (* this may never happen *) ; Interr := true; send im(z) to all c' in (Alloc(PR(C_1)) - r) ;) " *else (* c not equal r *) (* *if c in Alloc(SR(C_1)) then* "send iem(z) to r; " *if c in Alloc(PR(C_1)) then* "(receive im(z) from r (*may not happen *); Interr := true;)"

that C_1 has been executed. Also, each component involved in the weak while loop will have a local variable, say N, counting the number of times that C_1 has been executed locally. A component involved in the weak while loop will therefore accept from the *receive-buffer* a flow message indicating the beginning of C_2 only if its parameter n is equal to its local variable N. - Note: In general, a component executing its local behavior will need a stack of counter variables in case that weak while loops are embedded in one another; the top of the stack would correspond to the inner-most loop that is active.

We note that we use in the behavior expression for the interruption construct the parallel operator written "||*" which has the meaning that the two sub-behavior expressions are performed in parallel, but the whole construct terminates as soon as the first sub-behavior terminates. For instance, the meaning of the expression "NormalBeh ||* InterruptBeh" is shown by the Activity Diagram of Figure 4.

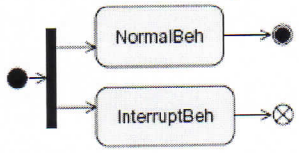

Fig. 4. Activity Diagram representing the meaning of "NormalBeh ||* InterruptBeh"

4 Derivation of component behaviors: examples

4.1 A simple example

We consider here the behavior shown in Figure 5(a) where the behaviors C1 and C2 are defined in the form of sequence diagrams, taken from [1]. Either C2 is executed directly, or C1 is executed followed by C2. A straightforward implementation of this choice may lead to the race condition shown in Figure 5(c).

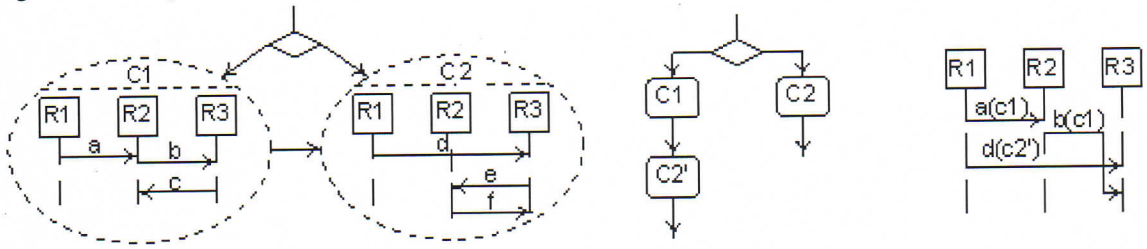

Fig. 5. (a) Choice implying choice propagation with races; (b) alternative view; (c) possible race condition (figure taken from [1])

The behavior shown in Figure 5(b) can be expressed by the behavior definition:

$C = (C1 \; [] \; \textit{nothing} \;)\; ;_w \; C2.$

From the sequence diagrams defining the sub-collaborations in the figure above, we deduce that the starting role for both sub-activities is R1, the terminating role of C1 is R2 and the terminating role of C2 is R3. All roles R1, R2 and R3 participate in both sub-activities. The choice between the sub-activities C1 and *nothing* can be performed as a local choice by role R1.

Let us assume that the three roles R1, R2 and R3 are allocated to three components Cp1, Cp2 and Cp3, respectively. According to the rules of Table 3, the only coordination messages to be introduced are *choice indication* messages (cim) sent to Cp2 and Cp3 when sub-activity *nothing* is chosen. (The reason is that components Cp2 and Cp3 participate in the C1 alternative of the choice, but not in the *nothing* alternative). These messages must be sent by Cp1, the only component that knows about the choice of *nothing*.

We therefore obtain the following behaviors for the three components (where send(x) and receive(x) stand for the sending and reception of message x, respectively):

$T_{Cp1}(C) = (\; T_{Cp1}(C1) \; []$ send cim(y) to Cp2 and Cp3 $) \; ; \; T_{Cp1}(C2)$

$T_{Cp2}(C) = (\; T_{Cp2}(C1) \; []$ receive cim(y) from Cp1 $) \; ; \; T_{Cp2}(C2)$

$T_{Cp3}(C) = (\; T_{Cp3}(C1) \; []$ receive cim(y) from Cp1 $) \; ; \; T_{Cp3}(C2)$

where the component behaviors for the given sub-activities C1 and C2 are defined according to Figure 5(a) as follows:

T_{Cp1} (C1) = send a
T_{Cp2} (C1) = receive a ; send b ; receive c
T_{Cp3} (C1) = receive b ; send c
T_{Cp1} (C2) = send d
T_{Cp2} (C2) = receive e ; send f
T_{Cp3} (C2) = receive d ; send e ; receive f

4.2 The telemedicine example

Referring to the example discussed in Section 2.3, let us assume that the roles P (patient), R (receptionist) and D (doctor) are to be implemented on three different components, also called P, R and D, respectively. This means that Alloc(P)=P, Alloc(R)=R and Alloc(D)=D. In the following we explain how the algorithm described in Section 3 can be used to derive the behavior of these components such that they realize the correct coordination of activities among these three components. We start out with the definition of the collaboration behavior given in Section 2.3, as follows:

<w>{P, ${}_sR_t$} = <wait>{P_t, ${}_sR$} $*_w$ ε
<act>{P_t, ${}_sR_t$, D_t} = <assign>{${}_sR_t$, D} $;_w$ <consult>{P_t, ${}_sD_t$}
<telemed> = <registr>{${}_sP_t$, R} $;_w$ (<w>{P, ${}_sR_t$ } |> <h-up>{${}_sP_t$} else <act>{ P_t, ${}_sR_t$, D_t })

Let us first determine the behavior for the sub-activities <w> and <act> at each of the three components:

T_P (<w>) = T_P (<wait>) * ; receive cim(y) from R
T_P (<act>) = T_P (<consult>) (* P is not involved in <assign> *)
T_R (<w>) = T_R (<wait>) * ; send cim(y) to P
T_R (<act>) = T_R (<assign>) (* R is not involved in <consult> *)
T_D (<w>) = ε
T_D (<act>) = T_D (<assign>) ; T_D (<consult>)

For the sub-collaboration <act> for example, the rule of Table 4 indicates that T_P (<act>) = T_P (<assign>) ; T_P (<consult>), however, T_P (<assign>) is empty since the component P (and the role P) is not involved in the <assign> sub-collaboration. For the behavior of component P for the sub-collaboration <w>, we have to evaluate the rule of Table 4 for C = C_1 $*_w$ C_2 where C_1 = <wait>{P_t, ${}_sR$} and C_2 = ε . Since *" P in Alloc(SR(C_1))"* is false and *" P in (Alloc(PR(C_1)) - Alloc(PR(C_2)))"* is true, we get the expression given above.

Now let us determine the behaviors of the three components for the <telemed> activity. Applying the rules of Table 4, we obtain the following behaviors for all components c = P, R or D:

T_c (<telemed>) = T_c (<registr>) ; T_c (<w> |> <h-up>; ε else <act>)
= T_c (<registr>) ; (NormalBeh $_c$ ||* InterruptBeh $_c$)

where T_D (<registr>) = ε and the behaviors NormalBeh $_c$ and InterruptBeh $_c$ are defined as follows:

NormalBeh $_P$ = (T_P (<w>) |> (wait(Interr); Interrupted := true;) else ε);
(receive cim(y) from R [] T_P (<act>))
InterruptBeh $_P$ = receive iem(z) from R; <h-up>; Interr := true; send im(z) to R
NormalBeh $_R$ = (T_R (<w>) |> (wait(Interr); Interrupted := true;) else ε);
(receive fim(x, i) from P; if i then Interrupted := true; if not Interrupted
then T_R (<act>)) || (wait(Interrupted); send cim(y) to D and P)
InterruptBeh $_R$ = send iem(z) to P; receive im(z) from P; Interr := true
NormalBeh $_D$ = T_D (<act>) [] receive cim(y) from R
InterruptBeh $_D$ = ε

Let us look at this derivation in more detail. First we note that the <telemed> collaboration is the weak sequential execution of the registration <registr>{${}_sP_t$, R}followed by an interruption behavior C = C_1 |> C_2 else C_3 , where C_1 = <w>{P, ${}_sR_t$ }, C_2 = <h-up>{${}_sP_t$}, and C_3 = <act>{P_t, ${}_sR_t$, D_t}. We see that C_2 consists only of the interrupt, namely <h-up>; therefore C'$_2$ = ε, and the role performing the interrupt is r = P. The rule forr weak sequencing in Table 4 leads to the first expression for T_c (<telemed>) given above. The rule for the interruption behavior in Table 4 leads to the second expression for T_c (<telemed>) above and the different forms of NormalBeh $_c$ and InterruptBeh $_c$ for the three components listed above.

Let us look at the derivation of the expression for NormalBeh $_c$ in the case of the component c = P. We have that *"P in Alloc(PR(C_1))"* is true, *"P in Alloc(TR(C_1))"* is false, *"P in (Alloc(SR(C'_2)) U Alloc(SR(C_3)))"* is also false since Alloc(SR(C'_2))= { }. We therefore have to determine "DOcim$_P$ (C'_2, C_3) [] DOcim$_P$ (C_3, C'_2)", which means "DOcim$_P$ (ε, <act>{P_t, $_sR_t$, D_t}) [] DOcim$_P$ (<act>{P_t, $_sR_t$, D_t}, ε)". The definition of the DOcim function in Table 4 for choice rule indicates for "DOcim$_P$ (ε, <act>{P_t, $_sR_t$, D_t}) " that the "else" part applies (since the first argument is ε which includes no participating roles); this leads to the generated code "receive cim(y) from R " since P is in Alloc(PR(<act>{P_t, $_sR_t$, D_t})) and if we select R as the component responsible to send the choice indication message to the other components when the interrupt leads to the execution of C'_2 (which is empty) instead of the <act> sub-collaboration. For "DOcim$_P$ (<act>{P_t, $_sR_t$, D_t}, ε)" we obtain "T$_P$ (<act>)" since P is in Alloc(PR(<act>{P_t, $_sR_t$, D_t})). Putting all these things together leads to the expression for NormalBeh $_P$ given above.
For the expression for InterrBeh $_c$ in the case of the component c = P_2, we have the case c = r, and the condition of the next IF statement is false. Therefore we obtain the generated code defined by the "else" part of the form " (receive iem(z) from R ; I-Enabled := true) || (wait(I-Enabled); <h-up>; Interr := true; send im(z) to R;) ". However, this can be simplified to " receive iem(z) from R ; <h-up>; Interr := true; send im(z) to R; " as given above.

By substituting the behaviors of the sub-activities <w> and <act> given above, we obtain three behavior expressions for the three system components P, R and D. These expressions include the local behaviors of the primitive collaborations <wait>, <assign> and <consult> and are independent of their particular nature; the expressions only depend of the sets of starting, terminating and participating roles given above. We note that these behaviors can also be represented by UML Activity Diagrams; for instance, Figure 6 shows the behavior for component P. We note that P is not a starting role of the <consult> sub-collaboration; therefore its behavior for this sub-collaboration will begin with the reception of some message. The choice between Tp<consult> and "receive cim(y)" at the component P will therefore depend on which message will be received by the component.

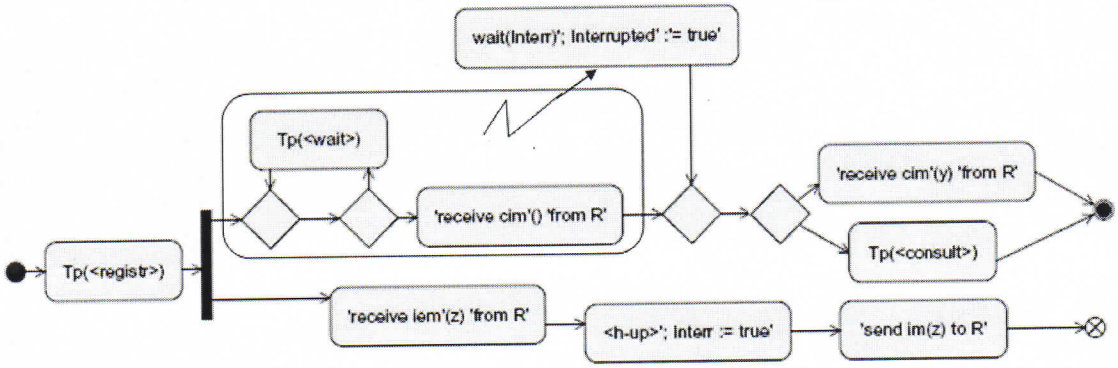

Fig. 6. Telemedicine example: Dynamic behavior of the P-component

Let us now assume that the basic activities consist of the following message exchanges:

<registration> = r1 from P to R; r2 from R to P.
<wait> = w1 from R to P.
<assign> = a1 from R to D; a2 from D to R.
<consult> = c1 from D to P; c2 from P to D.

Then the above definitions of the component behaviors give rise (among others) to the execution scenarios shown in Figure 7.

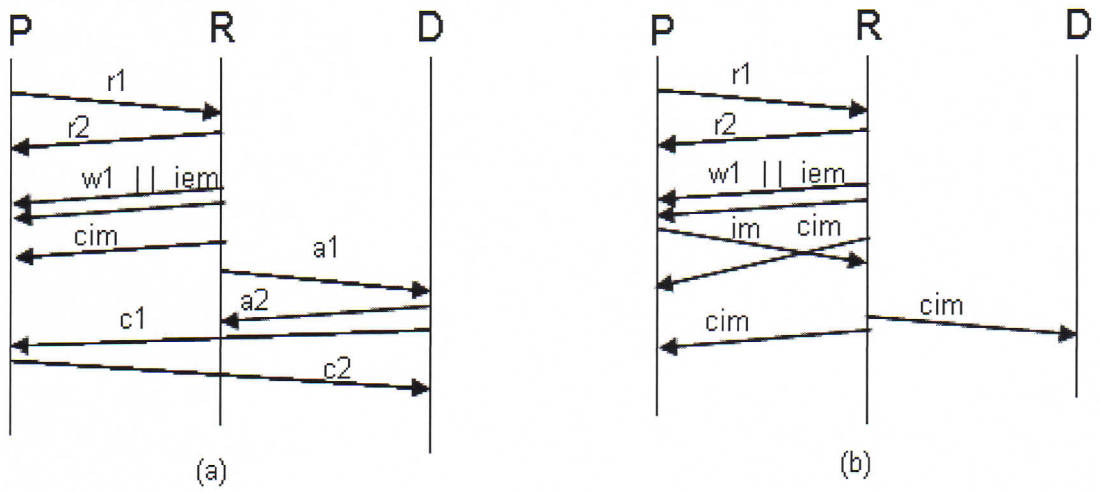

Fig. 7. Possible execution scenarios for the telemedicine application: (a) normal behavior, (b) the user quits the waiting loop

5 Discussion

5.1 Correctness of the derivation algorithm

It is clearly desirable to prove the correctness of the derivation algorithm of Section 3.3. Correctness means that for any global behavior specification C and any architectural choice defined by the Alloc() function, the component behaviors obtained by the algorithm, when executed in a distributed environment using reliable message transmission (with possibly out-of-sequence delivery), will exhibit the following properties:

1. General property (a) – **no unspecified receptions**: Any message received in the input buffer of a given component will eventually be consumed by the component.
2. General property (b) – **absence of deadlocks**: Any message required for the progress of the behavior of a given component will eventually be received in the input buffer of that component.
3. Service-specific property (a) – **safeness**: any possible sequence of execution of local actions by the different components satisfies the temporal constraints defined by the global behavior specification C.
4. Service-specific property (b) – **progress**: all sequences of execution of local actions that are possible according to the global behavior specification C can be realized by the component behaviors obtained by the algorithm.

Giving a formal proof of these properties is outside the scope of the paper. However, from the visual inspection of the rules in Table 4, one may get some confidence that the general properties are satisfied since the sending and receiving operations to be performed by the components in respect to a particular type of coordination message in a particular context are always defined within the same rule of Table 4. A formal proof of these properties requires first the formal definition of the semantics of the termporal operators defined in Table 1, and the formal meaning of what it means that the distributed execution satisfies all temporal constraints defined by the global behavior specification. A particular formalism towards this end is explored in [20] for a related purpose.

5.2 Considering data flow and more general control structures

In this section we discuss possible generalizations of the component derivation approach described in this paper. First we note that we only considered messages between the different system components for the temporal ordering of the activities performed by these different components. However, these activities also imply the exchange of data between the different components where the activities take place. Activity

Diagrams are able to associate the flow relationships between different activities with data or object flow. Such data flow could easily be associated with the flow coordination messages introduced in our formalism. For instance, the starting activity of C2 in a strongly sequenced requirements model C1 $;_s$ C2 would have to wait for flow messages from all terminating activities in C1; these messages may also include the data that is required to perform the first action in C2 (see also [4]).

Secondly, we note that the sequencing operators introduced in Section 2.2 do not allow to model arbitrary partial orders, as can be described by Activity Diagrams. For instance, the sequencing defined by the diagram in Figure 7 (where activity A4 has to wait for data from A1 and A3, while A2 only waits for A1) would be difficult to model using the sequencing operators of Table 1. In BPEL, the order dependency of the activity A4 on the completion of activity A1 could be modeled by a link. Such a data flow dependency could be implemented in the distributed design of the component behaviors by an additional flow message from a component involved in A1 to the starting component of A4. If the activities shown in the Activity Diagram are each localized at a particular component, the synchronization between the activities at these different components can be easily obtained by such flow messages. We note that work on deriving component behavior from global service descriptions in the form of Petri nets [12, 13] are relevant here, since Petri nets can model most aspects of Activity Diagrams. However, the situation becomes more complicated when the activities are in fact collaborations and when choices and loops are involved in the behavior of the Activity Diagram.

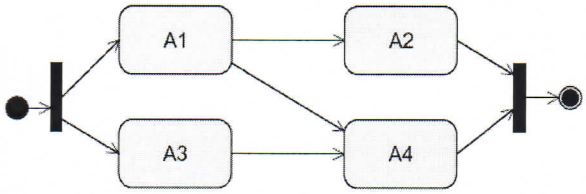

Fig. 5. Some simple Activity Diagram

5.3 Considering multiple sessions

We have considered in this paper a single occurrence of a collaboration and have assumed that all participating parties are statically known. However, in most practical applications, a given server may be involved in a large number of concurrent occurrences of the same collaboration, but initiated by different partners (clients). These different occurrences are usually distinguished by session or transaction identifiers. If each of these sessions only involve two partners – a client and a server – then the coordination of the activities in each session may proceed as explained above, since each client is only involved in a single session, and the server may perform the activities of the different sessions concurrently. Note, however, that there may be dependencies among the different session related to shared resources in the server.

However, when more than two parties are involved, the additional partners are often selected dynamically, possibly using some kind of service directory in order to find a service provider that satisfies the particular non-functional requirements for the partners already involved in the collaboration. In this context, the management of the sessions among the different partners becomes more complex because of the following reasons: (a) Each partner may be involved in concurrent sessions, possibly playing different roles in each of these sessions. (b) A unique session identifier may not be convenient for all partners. (c) Partners must be selected dynamically. (d) The choice indication message introduced in Section 3 must be eliminated if the destination is currently not involved in the collaboration. For example, if we consider the telemedicine example discussed above in the case of multiple patients and multiple doctors, there is no use of sending the choice indication message (cim) shown in Figure 7, since the doctor has not been involved in that session and will not be in the future.

6 Conclusions

We assume that the system design for a distributed system consisting of several separate components can be developed in the following steps:

1. Construction of a requirements model including the specification of the global behavior of the system in terms of certain activities and their temporal ordering.
2. Through architectural and non-functional requirements, a certain number of separate system components are identified; each of the activities identified at the requirements level is either allocated to one of these components, or performed as a collaboration among several components.
3. Based on the global behavior of the requirements, the identified components and the more detailed behavior of local activities and collaborations, the design of the distributed system is developed. This design defines the behavior of each of the system components including the messages required for realizing the collaborations and for ensuring the global coordination of the different activities between the different system components.

We have shown in this paper how the third step can be automated, assuming that the global behavior is given in a suitable modeling language. The modeling language supported by our design derivation algorithm described in Section 3 supports most of the concepts found in UML Activity Diagrams. This includes stepwise refinement where the behavior of a given activity is further detailed in terms of sub-activities and their ordering constraints, described as a separate activity diagram. In addition, a distinction between weak and strong sequencing can be made in the requirements model.

We believe that this approach to the automatic derivation of distributed system designs is useful in many fields of application, including distributed workflow management systems, service composition for communication services, e-commerce applications, or Web Services.

We plan to work on the implementation of the here proposed derivation algorithm in a tool environment and on the extension of the algorithm to support such partial order relationships as shown in Figure 5, including data flow.

Acknowledgements: I would like to thank Rolv Braek and Humberto Nicolás Castejón for many interesting discussions on the problems and issues related to this paper, and Fedwa Laamarti, Toqeer Israr, Reinhard Gotzhein, Mohammed Erradi and Jianxun Liu for suggesting improvements to an earlier version of this paper.

References

[1] H. Castejón , G.v. Bochmann, R. Bræk, Using Collaborations in the Development of Distributed Services, submitted for publication.

[2] H. Castejón, R. Bræk, G.v. Bochmann, Realizability of Collaboration-based Service Specifications, Proceedings of the 14th Asia-Pacific Soft. Eng. Conf. (APSEC'07), IEEE Computer Society Press, pp. 73-80, 2007.

[3] G. v. Bochmann and R. Gotzhein, Deriving protocol specifications from service specifications, Proc. ACM SIGCOMM Symposium, 1986, pp. 148-156.

[4] R. Gotzhein and G. v. Bochmann, Deriving protocol specifications from service specifications including parameters, ACM Transactions on Computer Systems, Vol.8, No.4, 1990, pp.255-283.

[5] F. Khendek, G. v. Bochmann and C. Kant, New results on deriving protocol specifications from services specifications, Proc. SIGCOMM'89, July 1989, in Computer Communications Review Vol.19 no.4, pp. 136-145.

[6] C. Kant, T. Higashino and G. v. Bochmann, Deriving protocol specifications from service specifications written in LOTOS, Distributed Computing, Vol. 10, No. 1, 1996, pp.29-47.

[7] H. Ben-Abdallah and S. Leue, "Syntactic detection of process divergence and non-local choice in Message Sequence Charts", *Proc. 2nd* Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97), 1997

[8] M. G. Gouda and Y.-T. Yu, Synthesis of communicating Finite State Machines with guaranteed progress, IEEE Trans on Communications, vol. Com-32, No. 7, July 1984, pp. 779-788.

[9] Mooij, Arjan J., Goga, Nicolae, & Romijn, Judi. 2005. Non-local Choice and Beyond: Intricacies of MSC Choice Nodes. *Pages 273–288 of: FASE.*

[10] Mooij, Arjan, Romijn, Judi, & Wesselink, Wieger. 2006. Realizability criteria for compositional MSC. *In: Proc. of 11th Intl. Conf. on Algebraic Methodology and Software Technology (AMAST'06).* LNCS, vol. 4019. Springer.

[11] A. Nakata, T. Higashino and K. Taniguchi, Protocol synthesis from context-free processes using event structures, in Proc. of 5th Int'l Conf. on Real-Time Computing Systems and Applications (RTCSA'98), Hiroshima, Japan, IEEE Computer Society Press, pp.173-180, Oct. 1998.
[12] H. Kahlouche and J. J. Girardot, "A Stepwise Requirement Based Approach for Synthesizing Protocol Specifications in an Interpreted Petri Net Model," Proc. INFOCOM'96, pp. 1165–1173, 1996.
[13] H. Yamaguchi, K. El-Fakih, G. v. Bochmann and T. Higashino, Protocol synthesis and re-synthesis with optimal allocation of resources based on extended Petri nets, Distributed Computing, Vol. 16, 1 (March 2003), pp. 21-36.
[14] Alur, Rajeev, Etessami, Kousha, & Yannakakis, Mihalis. 2000. Inference of message sequence charts. *Pages 304–313 of: 22nd International Conference on Software Engineering (ICSE'00).*
[15] F. Khendek and X. J. Zhang, "From MSC to SDL: Overview and an application to the autonomous shuttle transport system", *Proc. 2003 Dagstuhl Workshop on Scenarios: Models, Transformations and Tools*, LNCS, vol. 3466, 2005.
[16] R. Alur, K. Etessami and M. Yannakakis, "Inference of Message Sequence Charts", Proc. 22nd Intl. Conf. on Soft. Eng. (ICSE'00), 2000.
[17] R. Alur, G. J. Holzmann and D. Peled, "An analyzer for Message Sequence Charts", Software - Concepts and Tools, 17(2), 70–77, 1996.
[18] M.G. Nanda, S. Chandra and V. Sankar, "Decentralizing execution of composite Web Services", Proc. OOPSLA'04 (ACM), Vancouver, Canada, 2004.
[19] W. Tan and Y. Fan, "Dynamic workflow model fragmentation for distributed execution", Computers in Industry, Vol. 58, pp. 381-391 (2007).
[20] H.N. Castejon et al., "Investigating the realizability of collaboration-based service specifications" – Annex to PhD thesis, Dept. of Telematics, Norwegian University of Science and Technology, Trondheim, Norway, 2008.
[21] L. Lamport, Time, clocks and the ordering of events in a distributed system, Comm. ACM 21, 7 (July 1978), pp. 558-565.
[22] M. Carbone, K. Honda and N. Yoshida, "Structured communication-centred programming for Web Services", ESOP'2007.