Deriving component designs from global requirements

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. 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. In contrast to earlier work, strong and weak sequencing is distinguished and the primitive sub-activities included in the global behavior descriptions may be collaborations involving several components.

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. 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).

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 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 [14]. 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 [16].

These difficulties are increased by the use of **weak** sequencing operators in the description of the global system behavior. We note that **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. In contrast, **weak sequencing** between A1 and A2 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. 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.

The paper is structured as follows. In Section 2, we consider the temporal ordering of activities in a global requirements model, present the ordering operators that we assume in this paper and introduce a simple example. The main body of the paper is Section 3. After a review of past work on the transformation from global requirements to component behaviors, we describe in Section 3.1 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*. Then in Section 3.2, we present an algorithm that does this transformation automatically. The application of this algorithm to the example of Section 2 is discussed in Section 4. Finally, Section 5 provides our conclusions.

2 Describing composed collaborations and work flow 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 activities 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. 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 below.

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. In contrast, as mentioned in the Introduction, we assume that the basic activities in the description of the overall behavior may be collaborations involving several components.

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. In this paper, we use essentially the same behavior expressions to describe both of these behaviors. However, the distinction between weak and strong sequencing disappears when one deals with the behavior of a single component. The operators used for describing the temporal properties of these behavioral models are listed in Table 1. They 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 to flow control constructs that have a single entry point and a single exit point.

We write "<name>(R) = C" to indicate that the behavior of a collaboration, called <name>, which involves the set of roles R, is given by the expression C. The expression is composed out of primitive actions, the invocation of collaborations and certain sequencing operators as shown 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.

It is 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.

As an example we consider 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 (collaboration <registr> below). 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 (collaboration <w> below). If the patient does not want to wait any longer, he/she may hang up (action <h-up> below) and call back later.

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 sub-col.	<subcol>$^{(R)}$	Execution of a collaboration with name <subcol> involving the set R 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 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 a single role).
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 (or occurs when C_1 is already terminated) then C_3 will occur after C_1 (this is the other choice alternative).

This behavior can be described using the operators defined in Table 1 as follows:

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

where <w>$^{\{P, R\}}$ = <wait>$^{\{P, R\}}$ *$_w$ ε and

<act>$^{\{P, R, D\}}$ = <assign>$^{\{R, D\}}$;$_w$ <consult>$^{\{P, D\}}$

The roles involved in each activity are indicated by the upper indices (P stands for patient, R for receptionist, and D for doctor). This definition of the <telemed> workflow indicates that the registration of the patient is followed by a waiting period <w> that may be empty (ε); this waiting period may be interrupted when the patient hangs up the telephone. The waiting period, if not interrupted, is followed by the <act> sub-collaboration which consists of the weak sequential execution of the assignment of the patient to the doctor followed by the consultation. We note that the detailed interactions involved in each of these activities (or collaborations) are not

specified, and we do not need to know them for what we discuss in this paper. The behavior can also be represented by the UML Activity Diagrams shown in Figure 1.

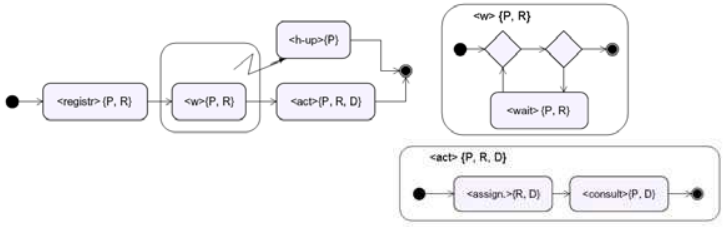

Fig. 1. Dynamic behavior of the Telemedicine collaboration, including two sub-collaborations

3 Deriving component-based designs

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 [20] 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.

3.1 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].

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.

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.

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

Operator	**Starting roles (SR)**	**Terminating roles (TR)**	**Participating roles (PR)**
<action>$^{(r)}$	{r}	{r}	{r}
<subcol>$^{(R)}$	SR(<name>)	TR(<name>)	PR(<name>) = R
C_1 ;$_s$ C_2	SR(C_1)	TR(C_2)	PR(C_1) U PR(C_2)
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)
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)
C_1 *$_s$ C_2	SR(C_1) = SR(C_2)= {r}	TR(C_2); SR(C_1) if C_2= ε	PR(C_1) U PR(C_2)
C_1 *$_w$ C_2	*as above*	TR(C_2) U (TR(C_1) - PR(C_2))	PR(C_1) U PR(C_2)
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)
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)

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.

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 (for details, see Table 3):

- *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.

3.2 Algorithm for deriving component behaviors from global behavior

In the following, we define an algorithm that realizes the derivation method explained in Section 3.1. 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 3 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 and decisions to be performed during 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 3.1, the parameters of the coordination messages and the buffering of received messages before their consumption are important elements for the correct operation of the distribution system derived by the algorithm of Table 3. 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 3. 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 3 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 deciding the looping conditions, say c_1, 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 that C_1 has been executed. For more details, see [17].

Table 3: Definition of the translation function T_c for component c

Operator	**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 Alloc(SR(C_1)) = {r}, and Alloc(SR(C_2)) = {r} or $C_2 = \varepsilon$. $T_c(C)$ ="(" $T_c(C_1)$ ";" 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 PR" *if c in PR then* "|| receive cim(y) from r" ")" *where* PR = Alloc(PR(C_1)) - Alloc(PR(C_2)) – {r}
$C = C_1 *_w C_2$	As above, except that the SFM and RFM constructs are absent
$C = C_1 \parallel C_2$	$T_c(C) = T_c(C_1) \parallel T_c(C_2)$
$C = C_1 \mid> C_2$ 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 SR" *if c in SR then* "(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)); " *where* SR = (Alloc(SR(C'_2)) U Alloc(SR(C_3))) –{c} **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); 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;)"

The expression "C1 ||* C2" has the meaning that the two sub-expressions C1 and C2 are executed in parallel, but the whole construct terminates as soon as C1 terminates.

4 Application to the Telemedicine example

Referring to the example discussed in Section 2, 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. 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 assume that the starting and terminating roles of the basic activities are as follows: SR(<registration>) = TR(<registration>) = {P}; SR(<wait>) = {R}; TR(<wait>) = {P}; SR(<h-up>) = TR(<h-up>) = {P}; SR(<assign>) = TR(<assign>) = {R}; SR(<consult>) = {D}; TR(<consult>) = {P, D}. Using Table 2, this leads to the starting and terminating roles of the sub-activities <w> and <act> as follows: SR(<w>) = TR(<w>) = {R}; SR(<act>) = {R}; TR(<act>) = {P, D, R};

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>)

Now let us determine the behaviors of the three components for the <telemed> activity. Applying the rules of Table 3, 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

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. Further details are given in [17]. If the behaviors of the primitive collaborations are also given, for instance in the form of simple Sequence Diagrams, a complete

description of each component behavior can be obtained by substitution. An example is discussed in [17] in more detail, including possible execution scenarios.

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 basic 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 a more detailed description of the basic activities, the distributed system design is developed which defines the behavior of each of the system components including the messages required for realizing the collaborations and for ensuring the global coordination of all activities among 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 plan to prove the correctness of the algorithm, as discussed in [17].

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 more general order relationships 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 and Toqeer Israr 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, G. J. Holzmann and D. Peled, "An analyzer for Message Sequence Charts", Software - Concepts and Tools, 17(2), 70–77, 1996.
[17] G.v. Bochmann, "Deriving component designs from global service and workflow specifications", submitted for publication.
[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] M. Carbone, K. Honda and N. Yoshida, "Structured communication-centred programming for Web Services", ESOP'2007.