# INF5150 Drop 1 from Group 2

Gorm Johnsen  Bjørnar Solhaug  Astri Ek Larsen
Ida Margrethe Heyerdahl  Sven-Jørgen Karlsen

October 14, 2005

## Contents

# 1 Introduction

From the exercise text:

> We are going to make a system to meet a group of people at a place that is optimally suited for those that will join.

We imaging that we are a third party vendor that offers a number of events identified by their type and time, e.g. “Go for a beer at 8 PM Saturday”, “Play bridge at 2 PM today”. It is possible to join such an event by sending an SMS message to our dedicated number. At appropriate time before the event, we will issue a message to the participant how he/she should use public transportation to reach a given meeting place.

We assume that the mobile phones sending the joining message can be positioned, and that we may use the Trafikanten system to find nearest bus stop, bus schedule and actual delays.

## 2 Assumptions

We have made the following assumptions:

- The abbreviation *BDS* stands for the class BlindDatingSystem.
- A user is uniquely identified by his or her mobile phone number. In other terms, the user communicates with the system with his or her mobile phone only.
- We are modelling the communication between the system and one single user at a time. The only exception to this is the system’s activity of publishing the events to all the users of the service.
- The system structure and interactions that are modelled in this assignment is limited to the requirements of the exercise text and our own extensions that we needed to add to the system in order to meet the requirements.
- System behaviour is modelled by means of sequence diagrams.
- We have not specified how the events are created. We imagine that they are defined by the system administrator and fed into the EventStore.
- We assume that our Trafikanten and our users (!) are capable of reading and handling geographical positions (represented by the Position class). It this is not the case, that is, human-readable addresses and descriptions of places are also needed, we could easily extend our system with a converter component to translate between the two representations of places (positions and meeting places).
- Undeletable execution occurrences (RSM problem): Please read short execution occurrences as invisible, they were impossible to get rid of (that is, deleting them swept off the messages they received) in some diagrams.

## 3 Generelt system

The messaging system we are going to specify, is an instance of the class BlindDatingSystem. The collaborations diagram of 3.1 shows the context of the system, its users (through their mobile phones), and the external Trafikanten system it depends on.

### 3.1 The System Context

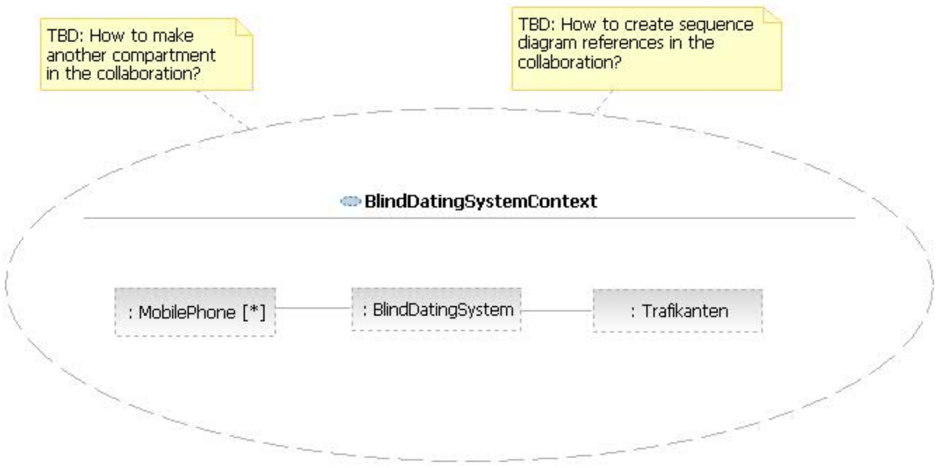

TBD: How to make another compartment in the collaboration?
TBD: How to create sequence diagram references in the collaboration?
BlindDatingSystemContext
: MobilePhone [*]
: BlindDatingSystem
: Trafikanten

Figure 1: The Context of the BlindDatingSystem

## 3.2 System Requirements Overview

The following Use Case Diagram gives an overview of the main requirements (and interactions of the system).

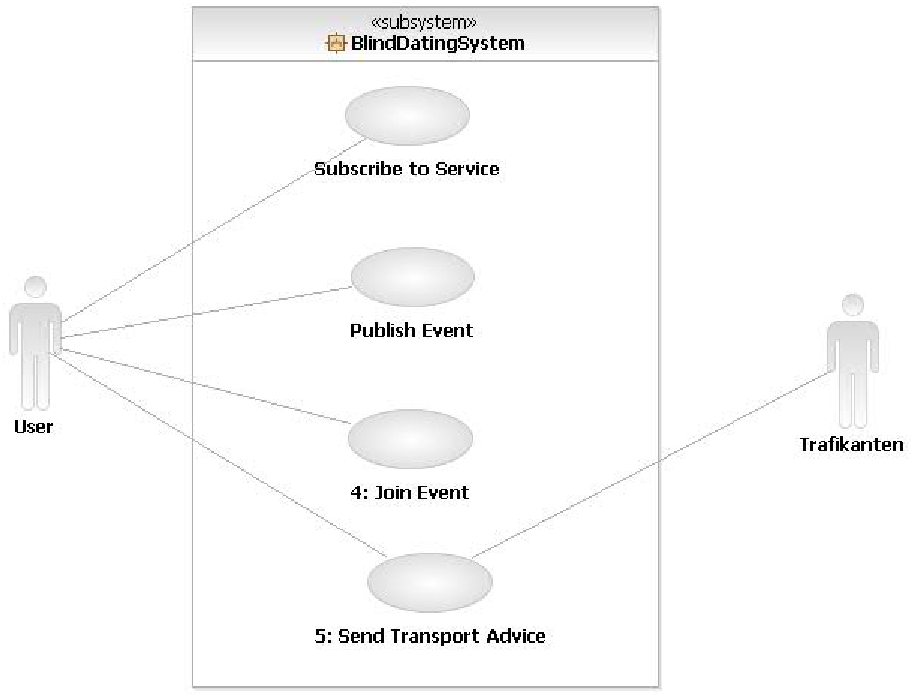


Figure 2: The Use Cases of the BlindDatingSystem

## 3.3 System Structure

The following composite structure diagram (CSD) shows the composition of the main system class.

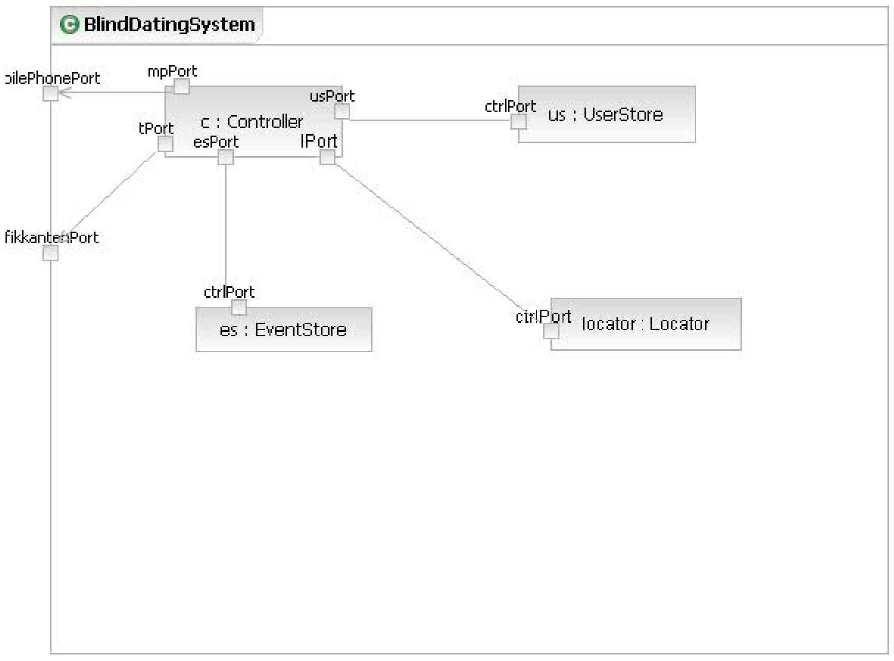


Figure 3: The Composite Structure of the BlindDatingSystem Class

### 3.3.1 The EventStore's Structure

The following CSD shows the composition of the EventStore part of the system class.

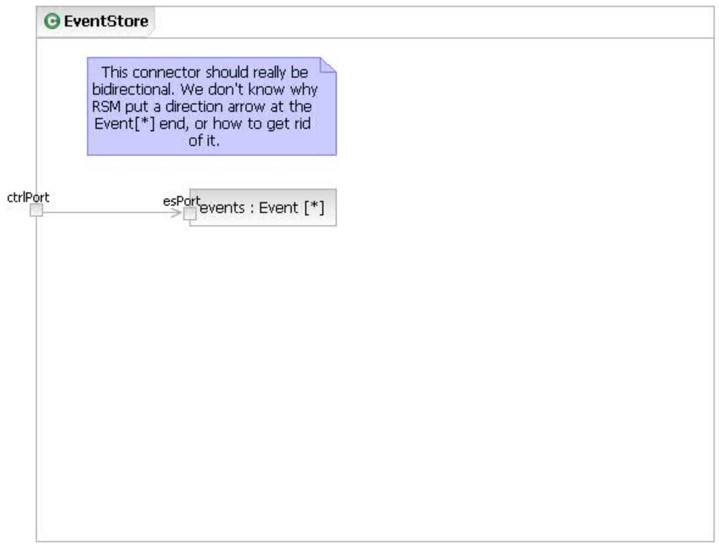

EventStore
This connector should really be bidirectional. We don't know why RSM put a direction arrow at the Event[*] end, or how to get rid of it.
ctrlPort
esPort
events : Event [*]

Figure 4: The Composite Structure of the EventStore Class

### 3.3.2 The UserStore's Structure

The following CSD shows the composition of the UserStore part of the system class.

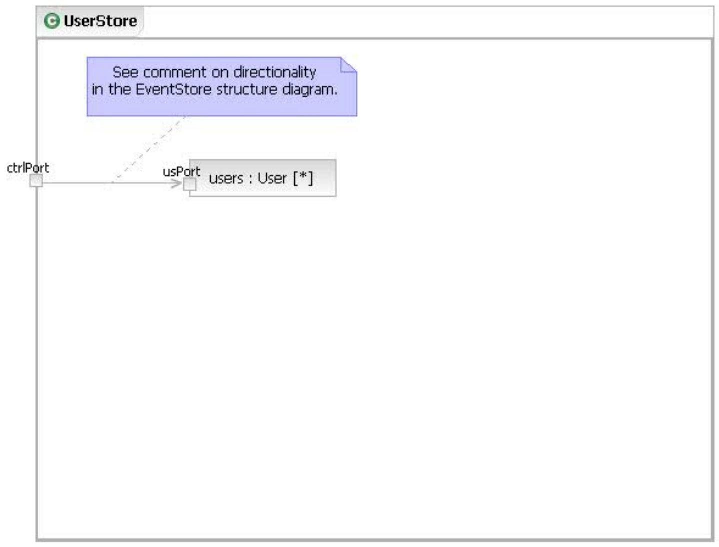

UserStore
See comment on directionality in the EventStore structure diagram.
ctrlPort
usPort
users : User [*]

Figure 5: The Composite Structure of the UserStore Class

## 3.4 Class Diagram

The following class diagram gives an overview of the classes in the system. The Controller part of the system class is responsible of organizing all business logic, and acts as a facade of the other parts. The Locator class is an abstraction of an optimal meeting place selector component, which we do not specify further, merely assume it is capable of finding suitable locations, given events and positions.

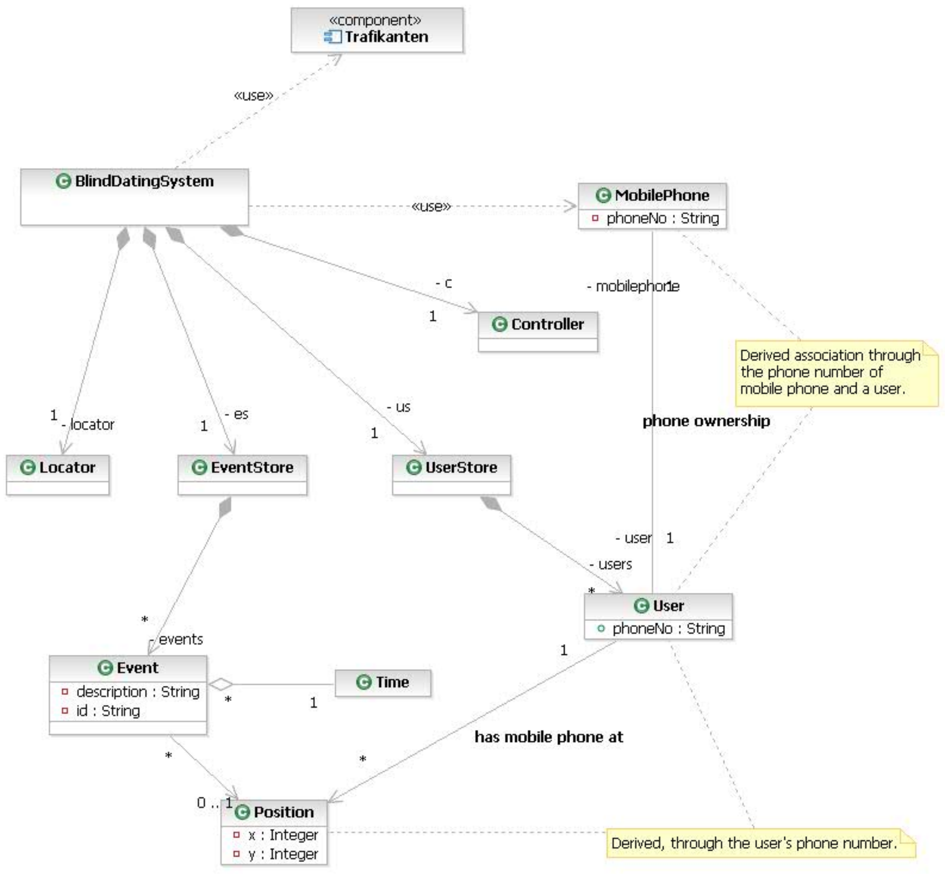


Figure 6: Detailed Overview of the Classes in the System

## 3.5 Interaction Overview

### 3.5.1 Activity Diagram View

The following activity diagram tries to look like an UML2 interaction overview SD diagram. Please interpret the action nodes with the suffix "ref" as UML2 interaction occurrences.

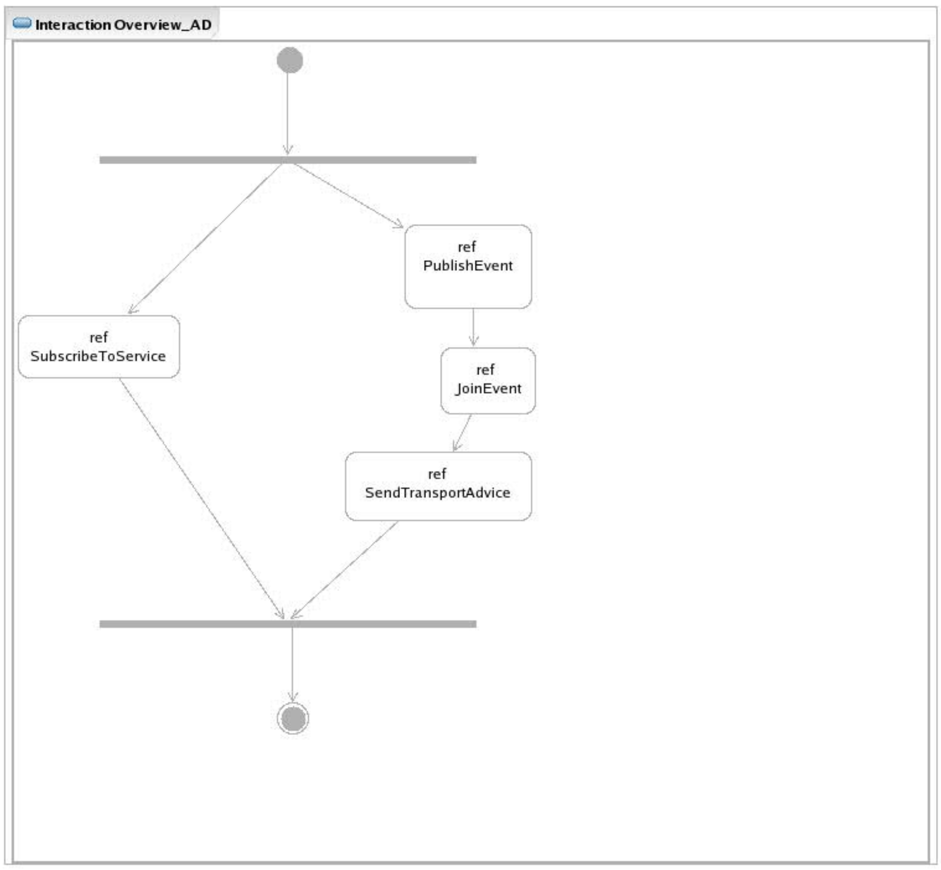


Figure 7: Overview of the main interactions of the System

### 3.5.2 Sequence Diagram View

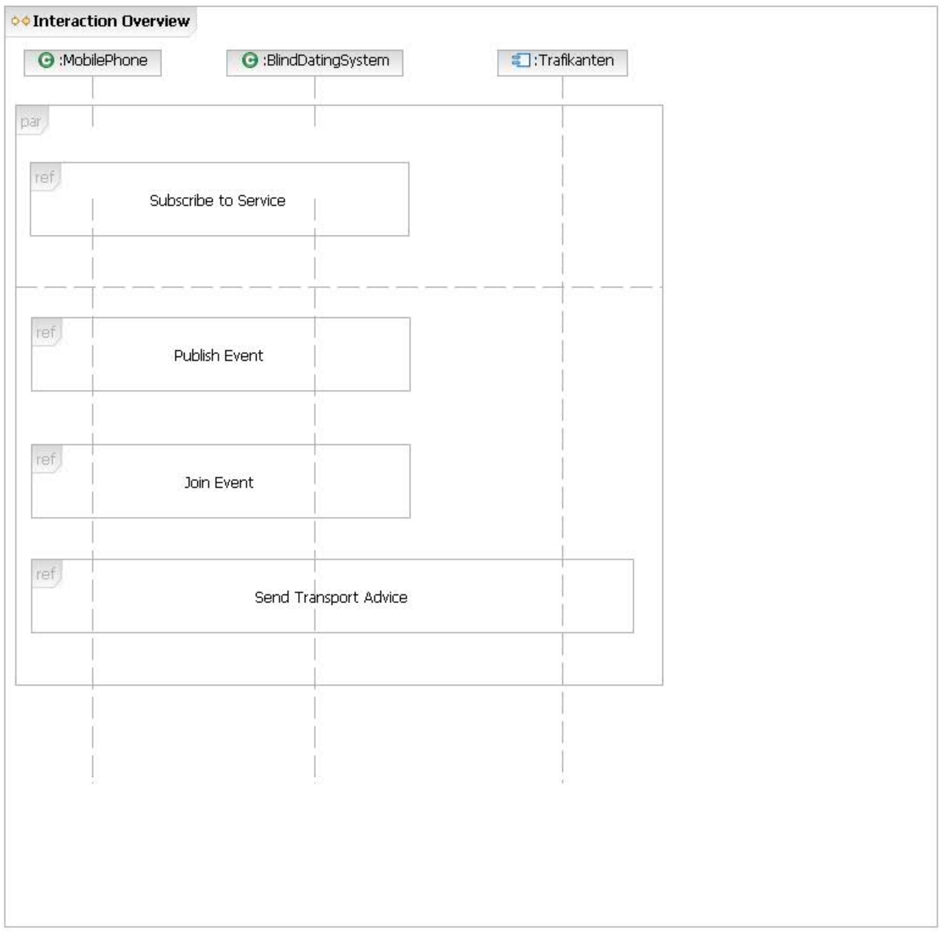


Figure 8: Overview of the main interactions of the System

## 3.6 Subscribe to Service

When a person wants to receive the services of the system, he/she has to subscribe to the service. This can be done by sending an sms to the system after which the system creates a new user with the phone number as userID. The system returns a welcome message to the mobile phone. One negative trace of this function could be that a person tries to subscribe to the service with the same phone number several times. However, this is solved by having a idemponent Subscribe To Service that just overwrites the user in the system with the same phone number.

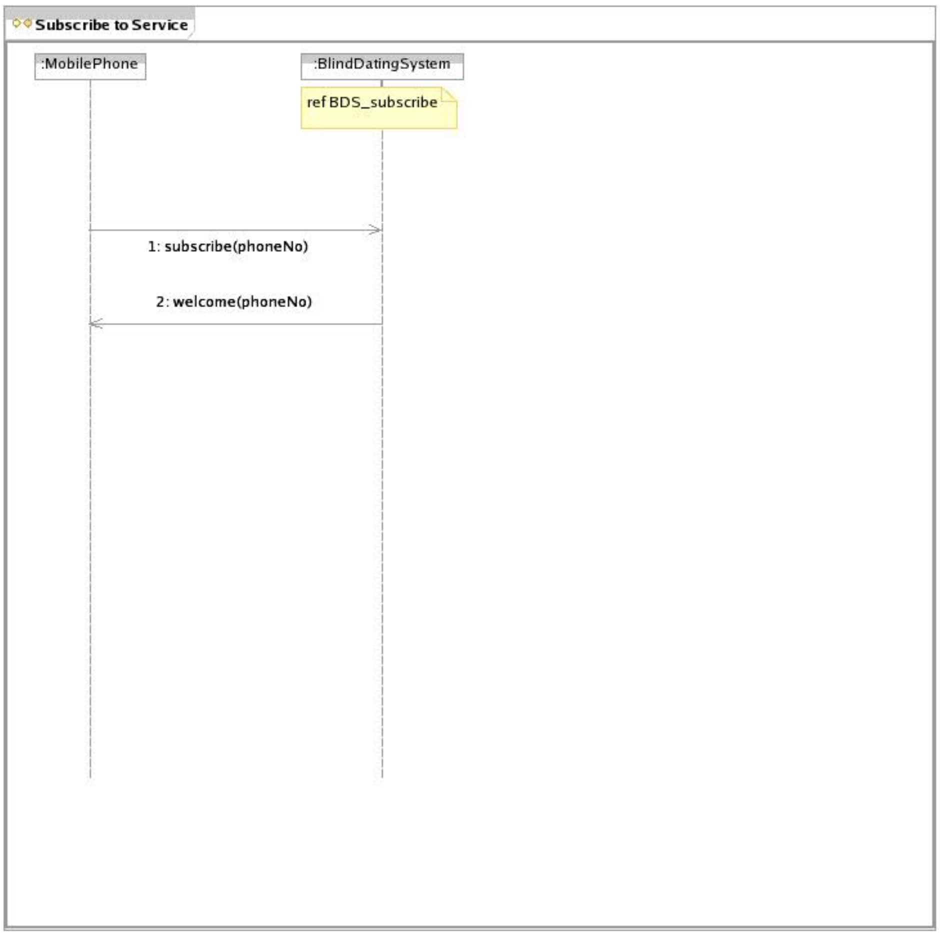


Figure 9: The “Subscribe to Service” interaction

### 3.6.1 BDS_Subscribe

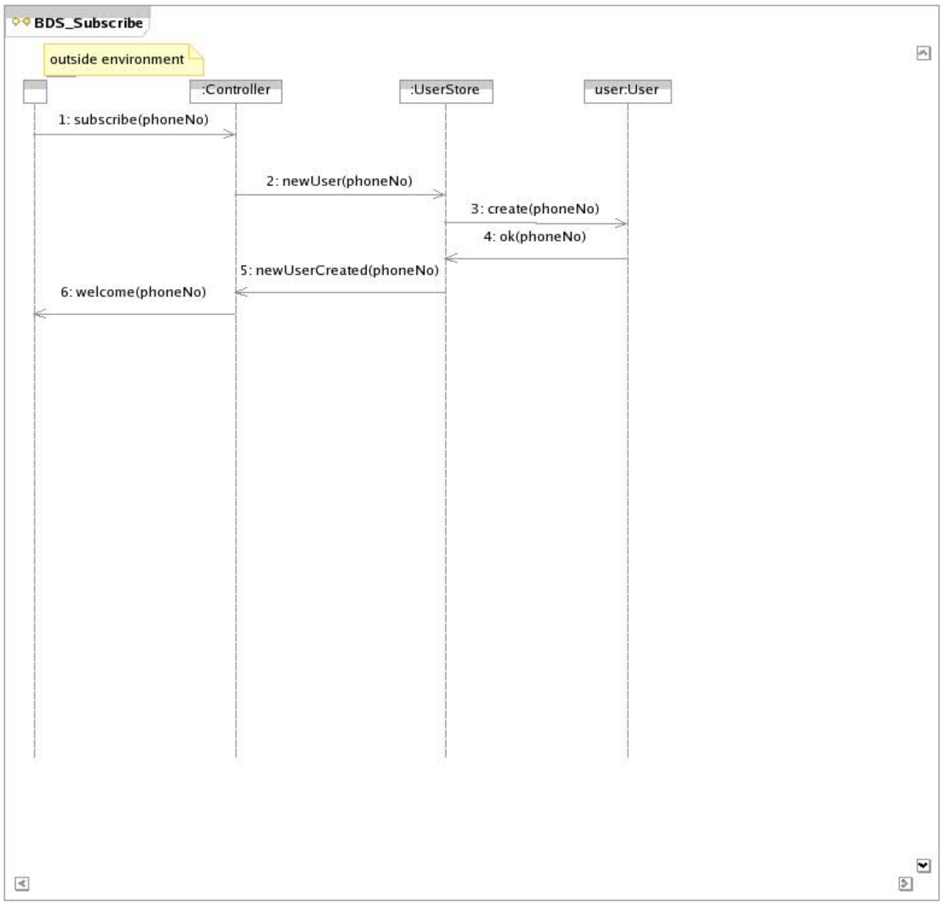


Figure 10: "Subscribe to service" decomposed w.r.t. the BDS lifeline

## 3.7 Publish Event

Publish Event models how the system broadcasts to the users the set of events that is available to the users. This behaviour is triggered by the Event Store instance.

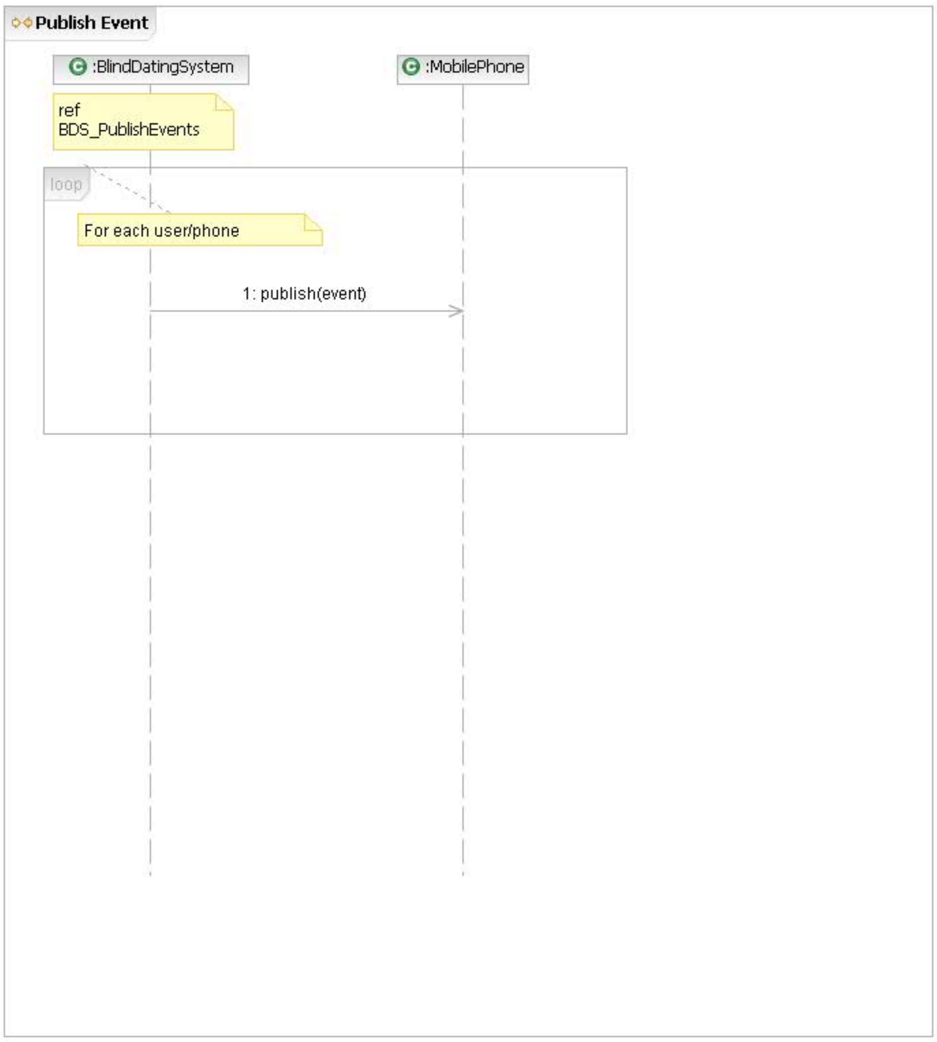


Figure 11: The "Publish Event" interaction

### 3.7.1 BDS_PublishEvent

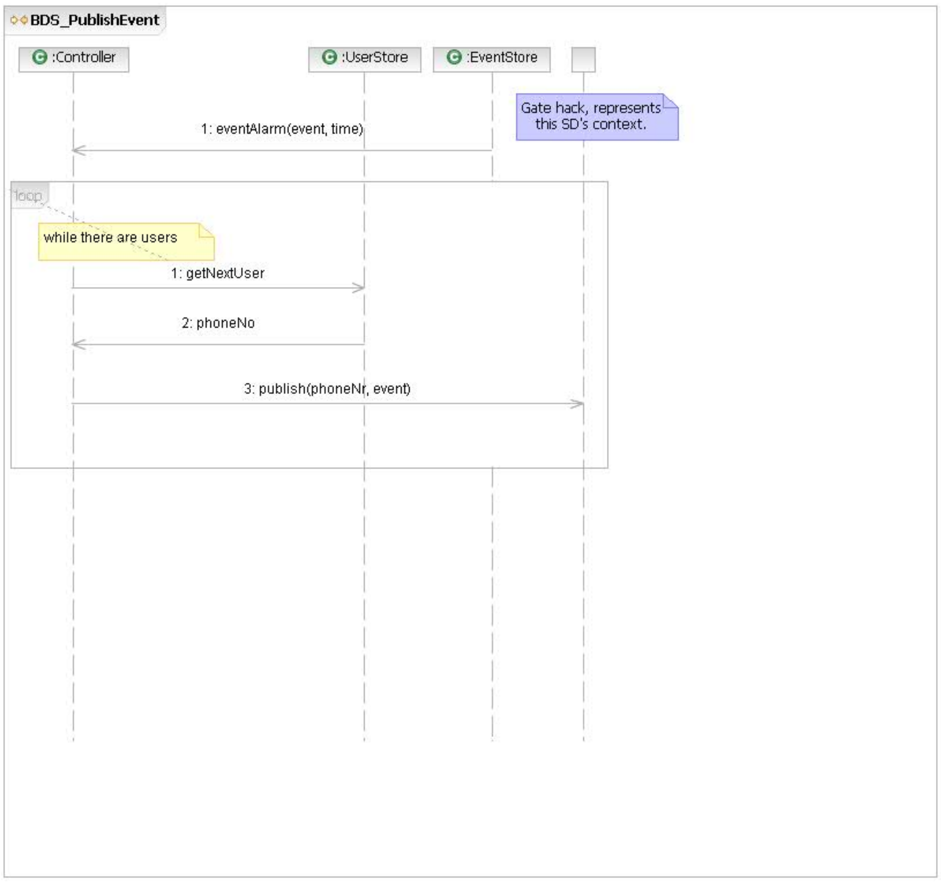


Figure 12: The “Publish Event” decomposition of the system’s lifeline

## 3.8 Join Event

Join Event models how a given user interacts with the system in order to join an event that previously has been published by the system. If the user has already signed up for this particular event, a message with this information is returned by the system. If not, the system replies with a confirm message.

The system should not return a confirmation in case the user has already signed up for the event. Hence the negative combined fragment.

In case the system receives a join message with which an unknown phone number is associated, the system returns a message in which the owner of the mobile phone is asked to register.

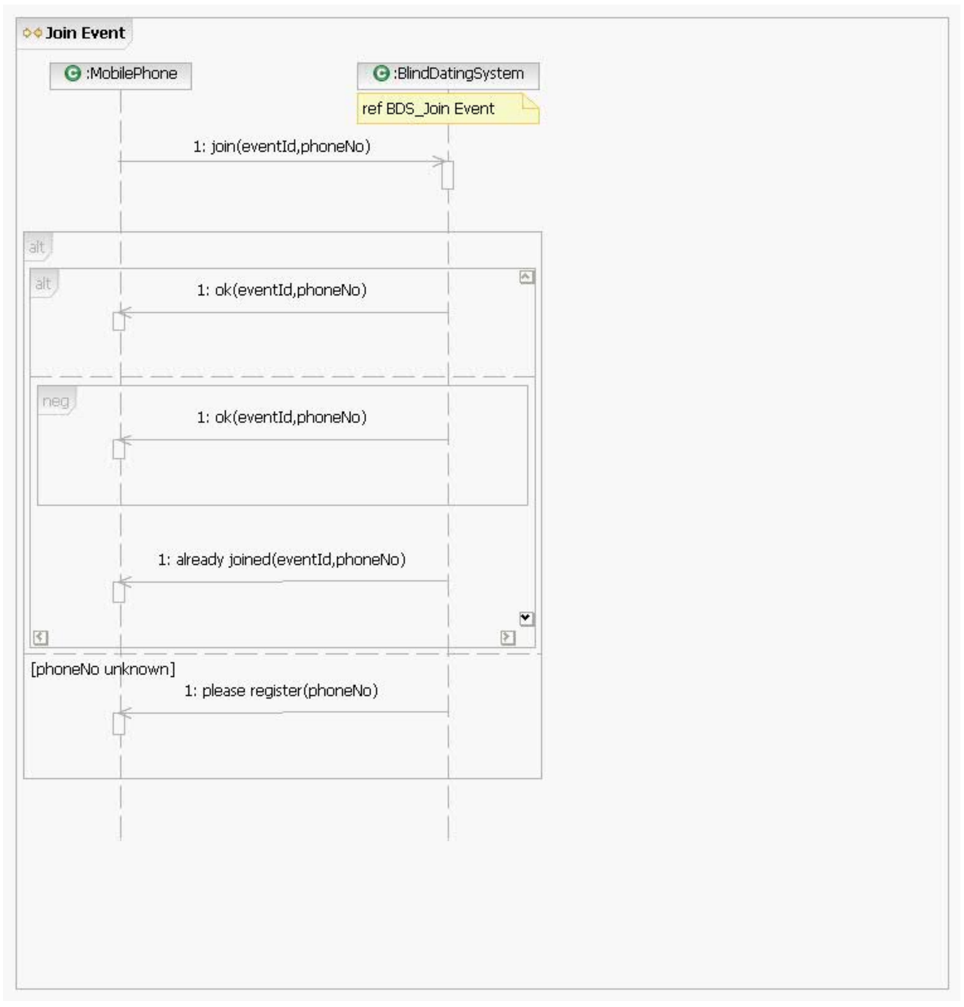


Figure 13: The "Join Event" interaction

### 3.8.1 BDS_JoinEvent

BDS_Join Event shows the decomposition of the lifeline BlindDatingSystem of Join Event.

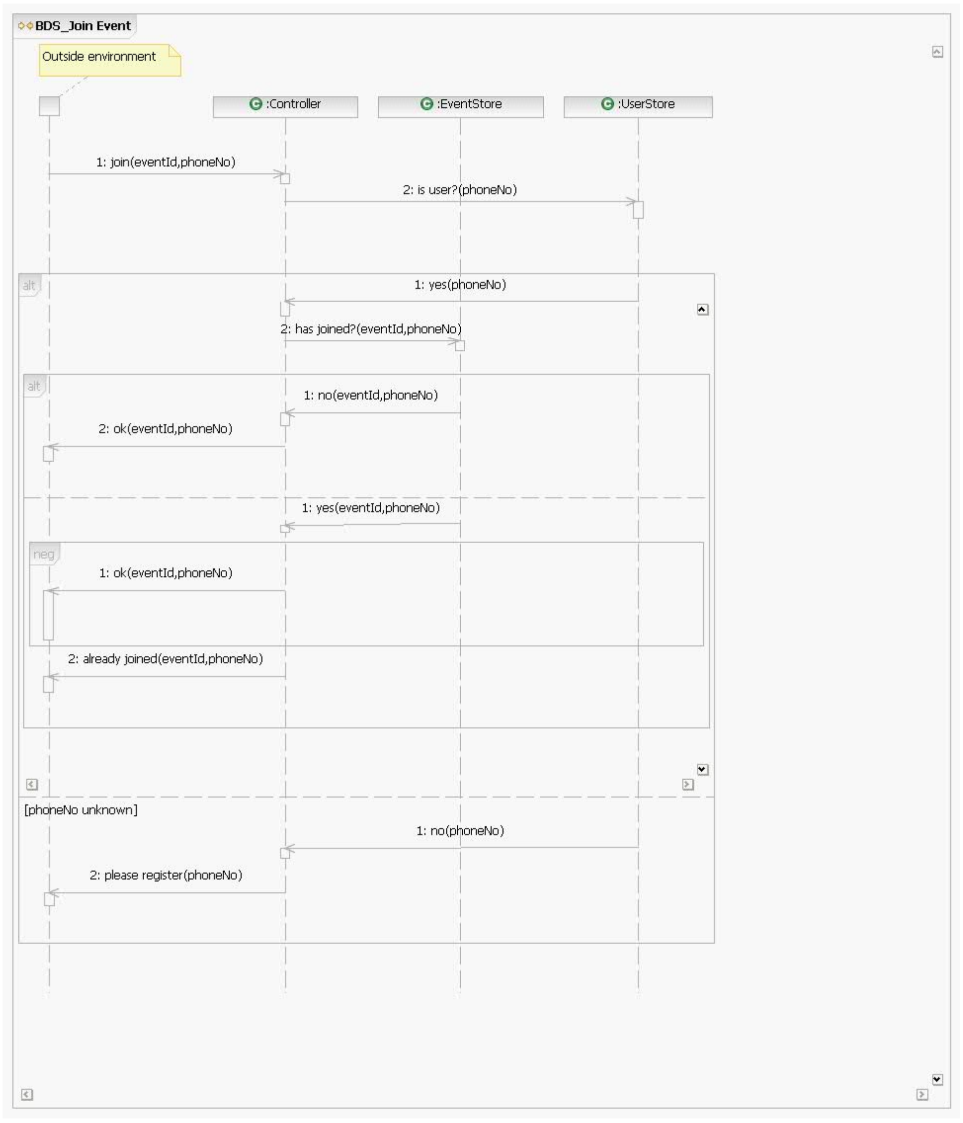


Figure 14: The "Join Event" decomposition of the BDS lifeline

## 3.9 Send Transport Advice

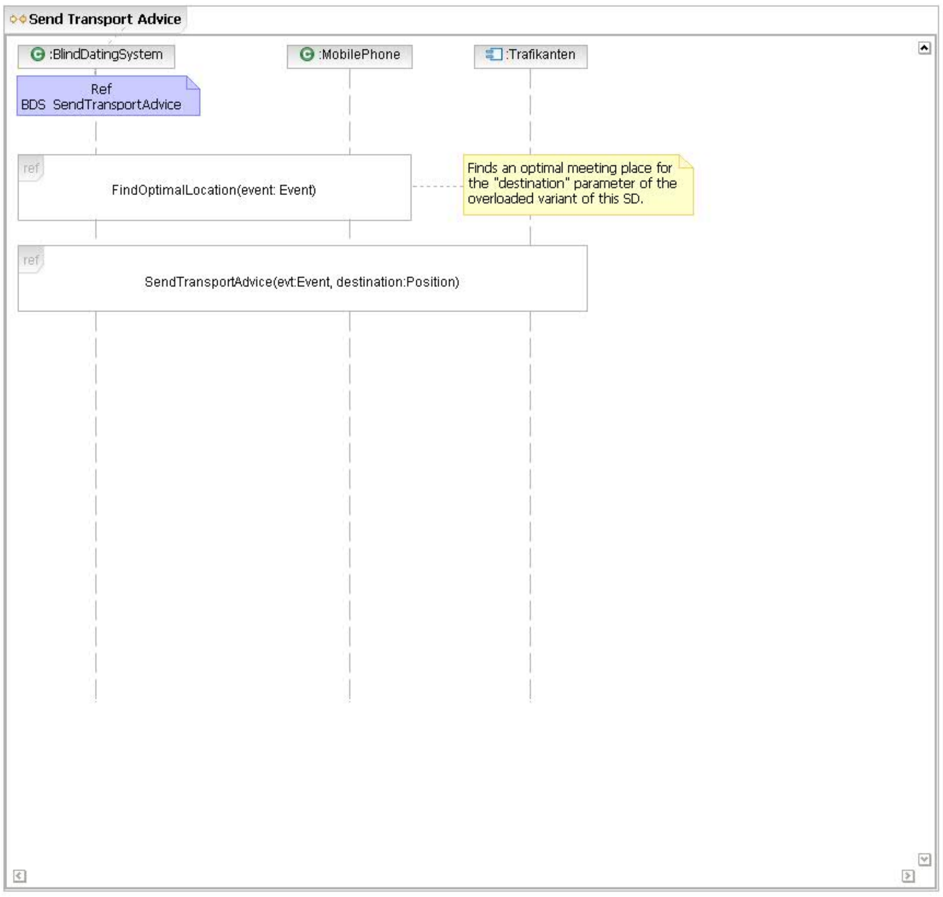

Send Transport Advice
:BlindDatingSystem
:MobilePhone
:Trafikanten
Ref
BDS SendTransportAdvice
ref
FindOptimalLocation(event: Event)
Finds an optimal meeting place for the "destination" parameter of the overloaded variant of this SD.
ref
SendTransportAdvice(evt:Event, destination:Position)

Figure 15: The “Send Transport Advice” interaction

### 3.9.1 FindOptimalLocation

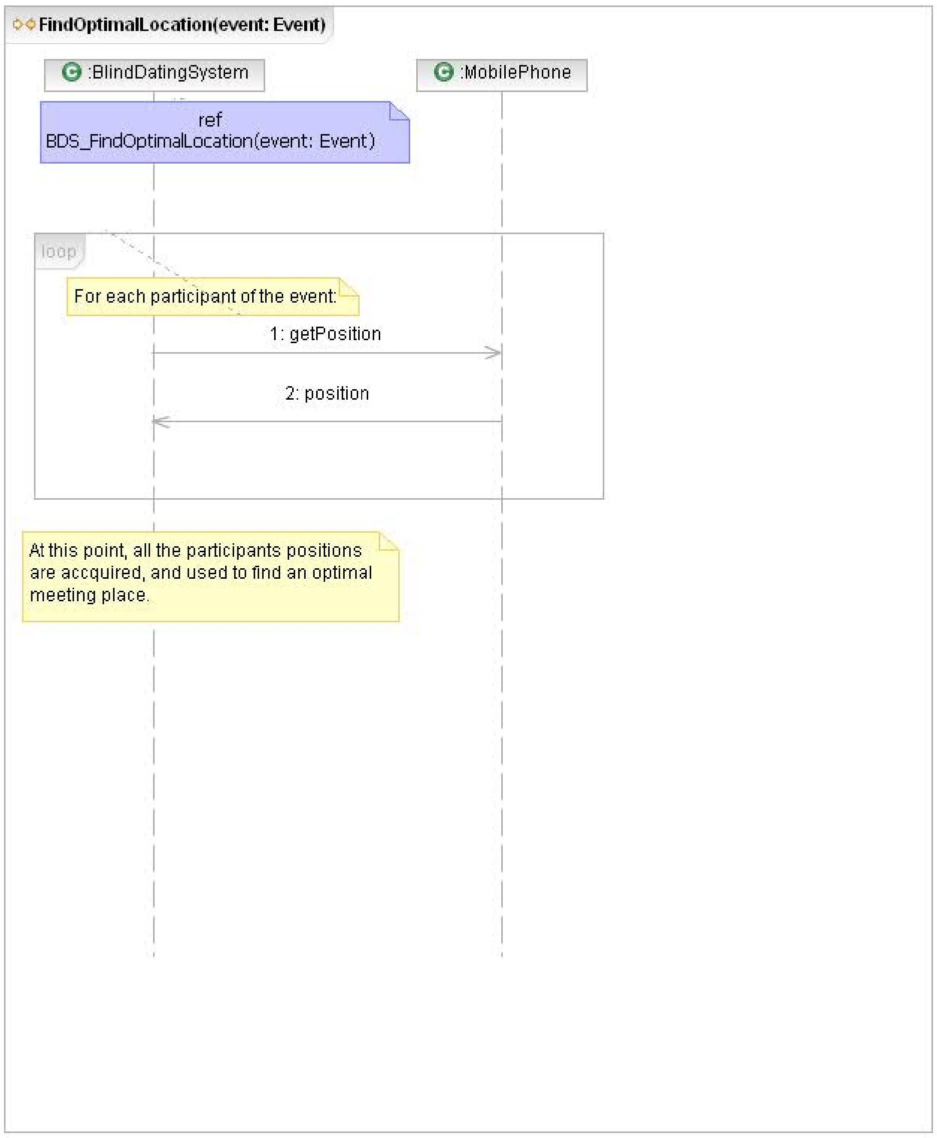

FindOptimalLocation(event: Event)
:BlindDatingSystem
:MobilePhone
ref
BDS_FindOptimalLocation(event: Event)
loop
For each participant of the event:
1: getPosition
2: position
At this point, all the participants positions are accquired, and used to find an optimal meeting place.

Figure 16: The “Find Optimal Location” interaction

### 3.9.2 Send Transport Advice(Event, Location)

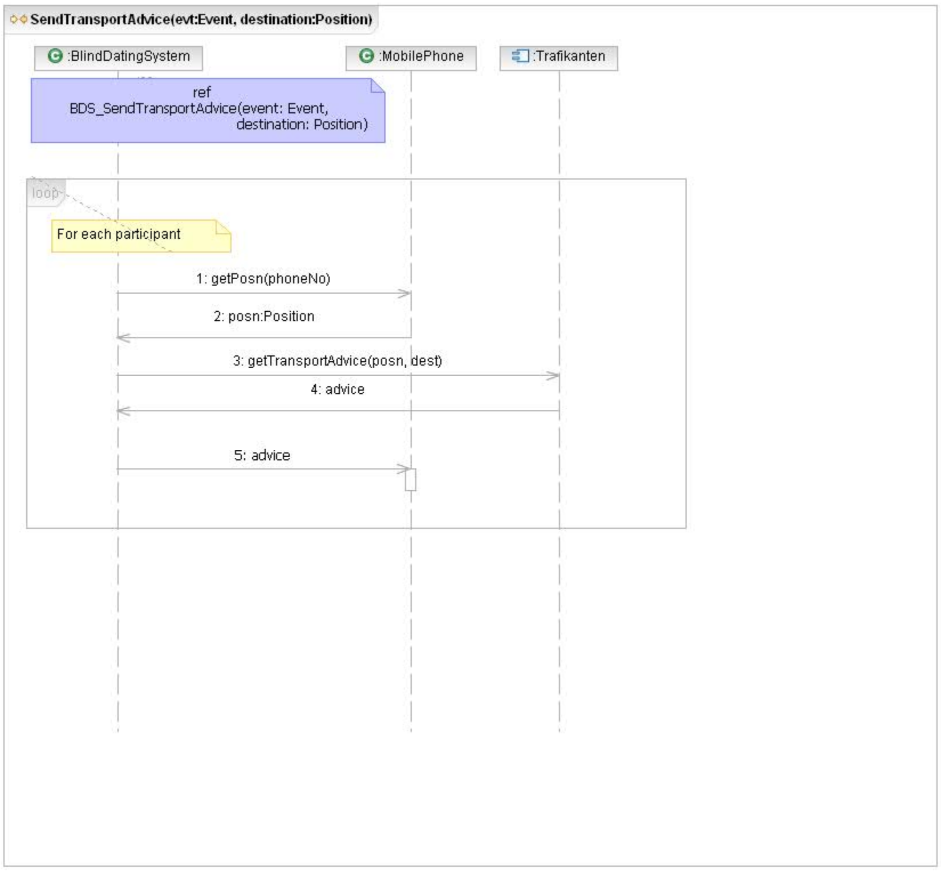

SendTransportAdvice(evt:Event, destination:Position)
:BlindDatingSystem
:MobilePhone
:Trafikanten
ref
BDS_SendTransportAdvice(event: Event, destination: Position)
loop
For each participant
1: getPosn(phoneNo)
2: posn:Position
3: getTransportAdvice(posn, dest)
4: advice
5: advice

Figure 17: The "Send Transport Advice" with parameters interaction

### 3.9.3 BDS_SendTransportAdvice

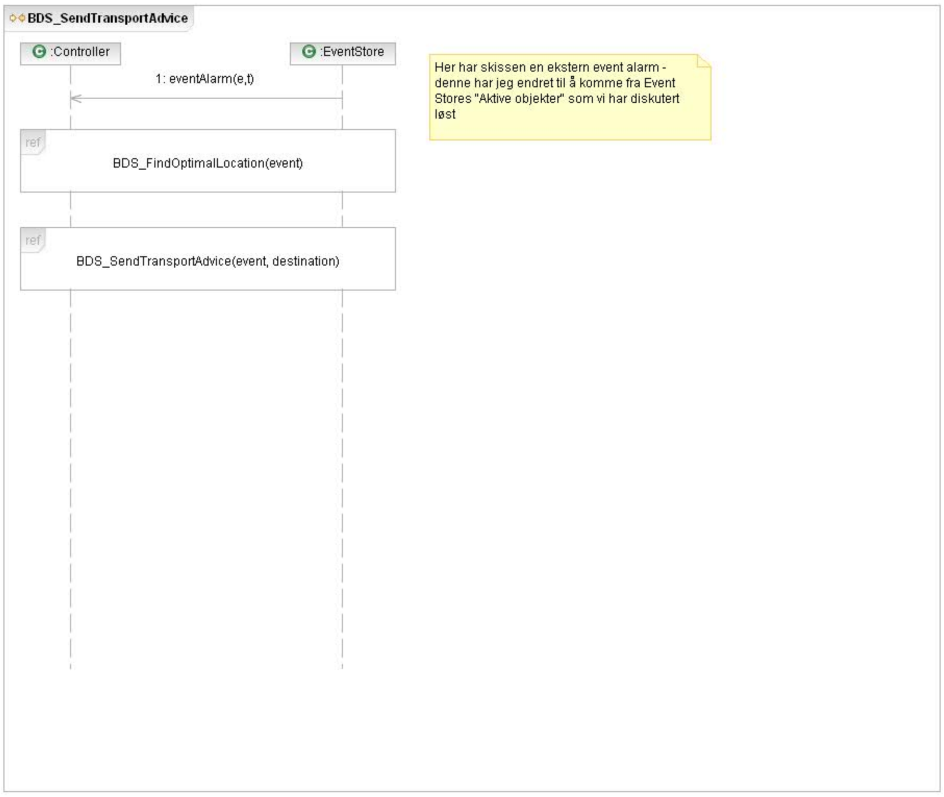
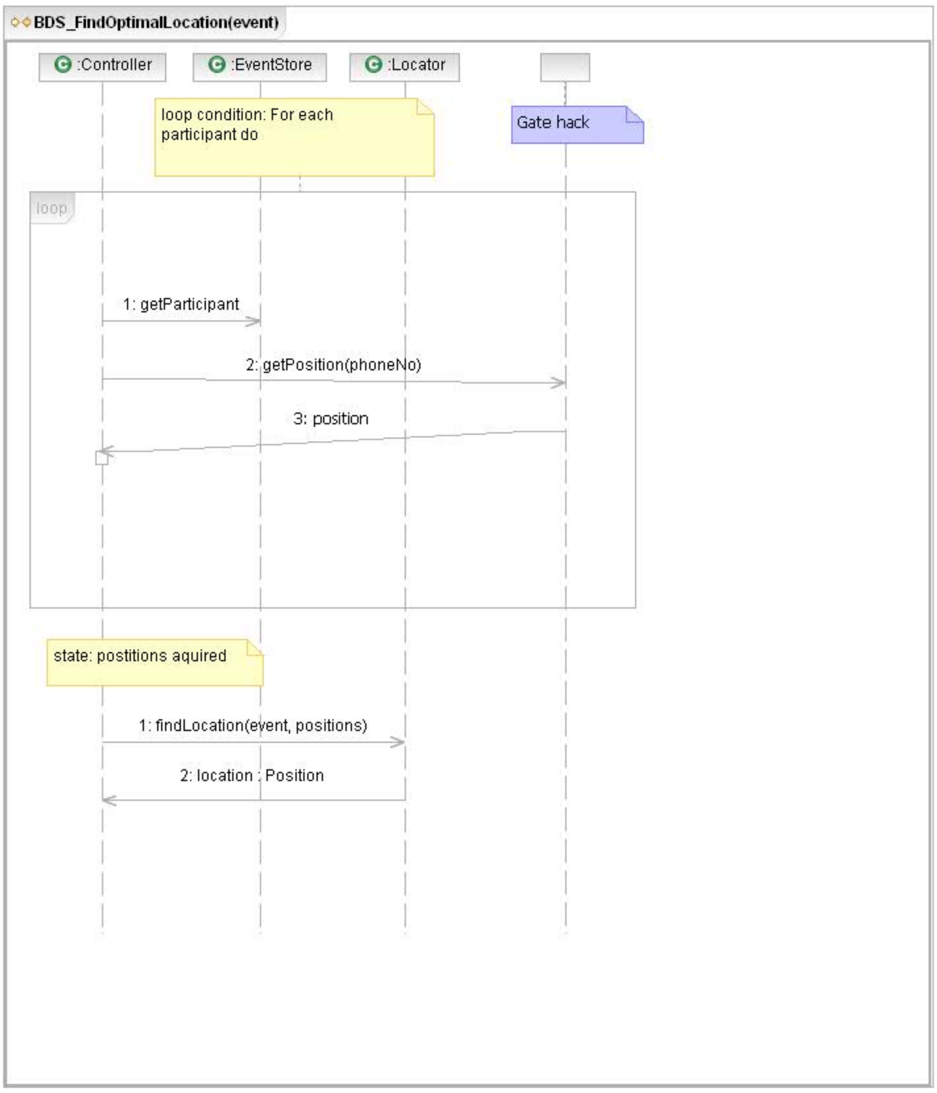

BDS_SendTransportAdvice
:Controller
:EventStore
1: eventAlarm(e,t)
Her har skissen en ekstern event alarm - denne har jeg endret til å komme fra Event Stores "Aktive objekter" som vi har diskutert løst
ref
BDS_FindOptimalLocation(event)
ref
BDS_SendTransportAdvice(event, destination)

Figure 18: The "Send Transport Advice" decomposition of the BDS lifeline

### 3.9.4 BDS_FindOptimalLocation(event: Event)



BDS_FindOptimalLocation(event)
:Controller
:EventStore
:Locator
loop condition: For each participant do
Gate hack
loop
1: getParticipant
2: getPosition(phoneNo)
3: position
state: postitions aquired
1: findLocation(event, positions)
2: location : Position

Figure 19: The "Find Optimal Location(e: Event) " decomposition of the BDS lifeline

### 3.9.5 BDS_SendTransportAdvice(event: Event, destination: Location)

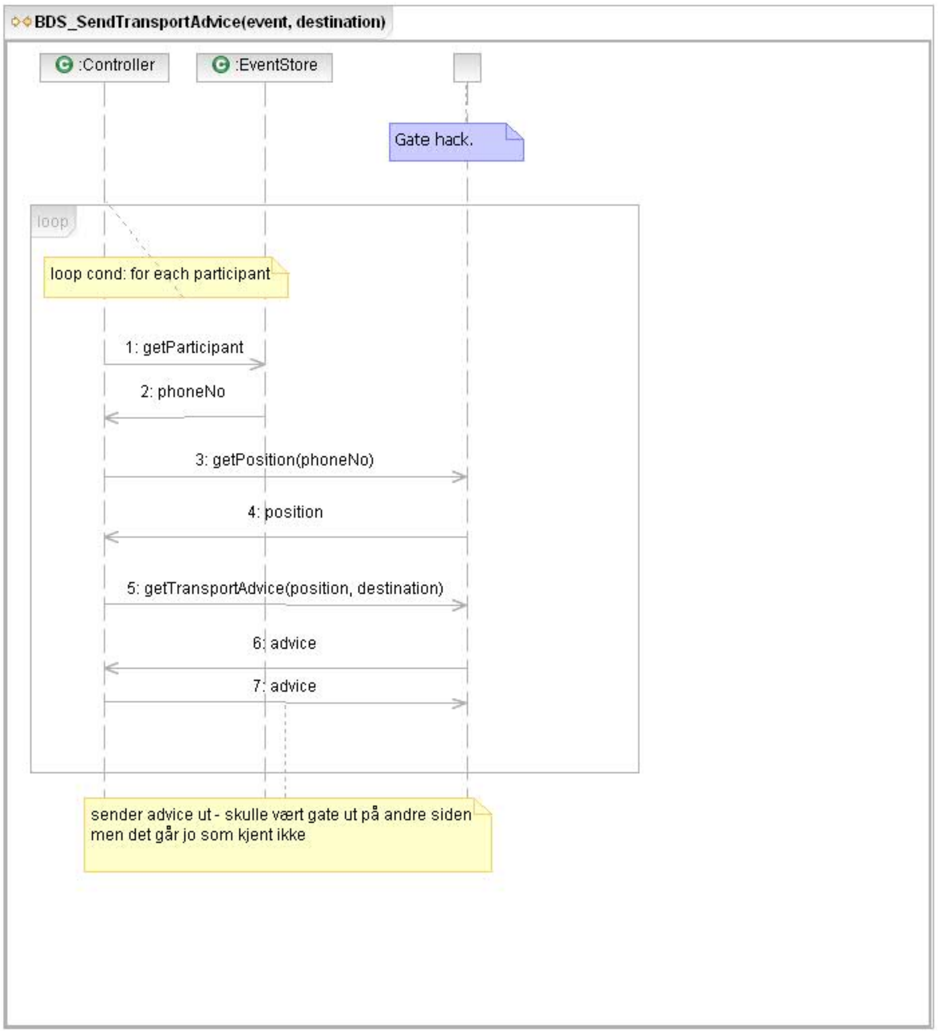

BDS_SendTransportAdvice(event, destination)
:Controller
:EventStore
Gate hack.
loop
loop cond: for each participant
1: getParticipant
2: phoneNo
3: getPosition(phoneNo)
4: position
5: getTransportAdvice(position, destination)
6: advice
7: advice
sender advice ut - skulle vært gate ut på andre siden men det går jo som kjent ikke

Figure 20: The "Send Transport Advice(e: Event, dst: Location) " decomposition of the BDS lifeline

# 4 Alternative functionalities

We will in this section change the general system of the previous section by introducing basically two new features as a replacement of earlier features.

The first novelty is to give the location as a part of the original event description, whereas the second novelty is to fix the time of the event to be 30 minutes after the event has been published. We will in the next section show that these changes represent a refinement of the system.

The two new features describe two exclusive refinement alternatives. In other terms, an implementation should not implement the two simultaneously. We will, however, provide a specification that captures both changes and prove that this specification is a refinement of the original one.

## 4.1 Interaction overview

### 4.1.1 Activity diagram view

The activity diagram for the refined system shows the two new functionalities. Notice that the behaviour of the original system as specified by Subscribe to Service is left unchanged, while the remaining behaviour of the original system is categorized as negative.

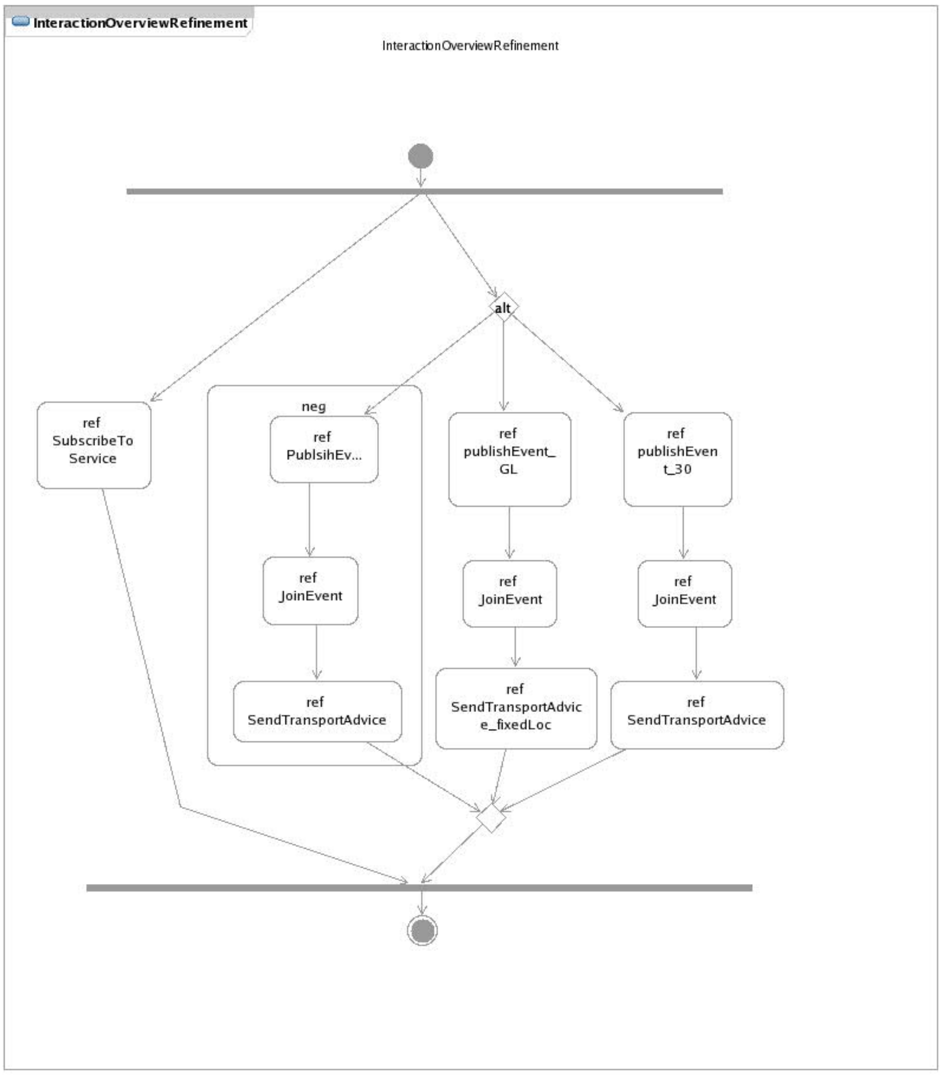


Figure 21: Interaction overview with refinements

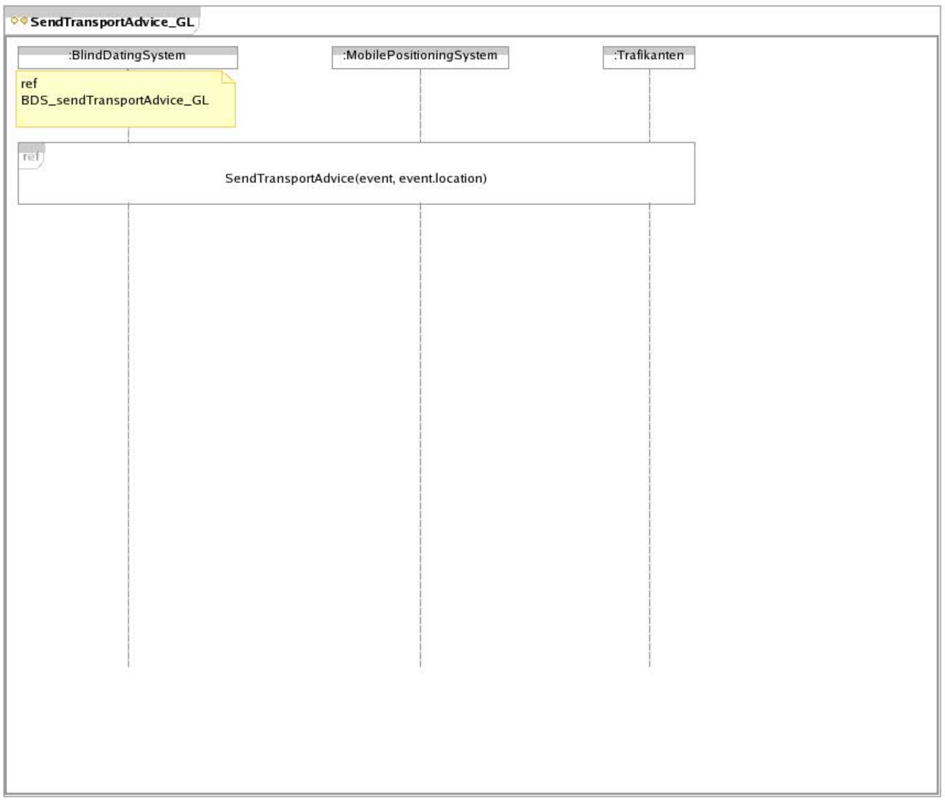


Figure 22: The "Send Transport Advice" decomposition of the BDS lifeline

### 4.2 Refinement 1 - Location given in the original event

When an event is created in the original system, the location of the event is decided from the best location with respect to the users who want to join the event. In this refinement, the location of the event is given in the original event description. Since this feature is the only difference between this refinement and the original Publish Event functionality, the sequence diagrams of PublishEvent_GL will be identical with the exception of an additional attribute (the location). These diagrams are therefore omitted here.

The sequence diagram for the transport advice when the location is already given, SendTransportAdvice_GL, differs from the original one and is described in the sequence diagram below.

### 4.3 Refinement 2 - Time of event 30 minutes after publishment

This new functionality fixes the time of the event to be 30 minutes after the publishment of the event. The sequence diagrams show how the system

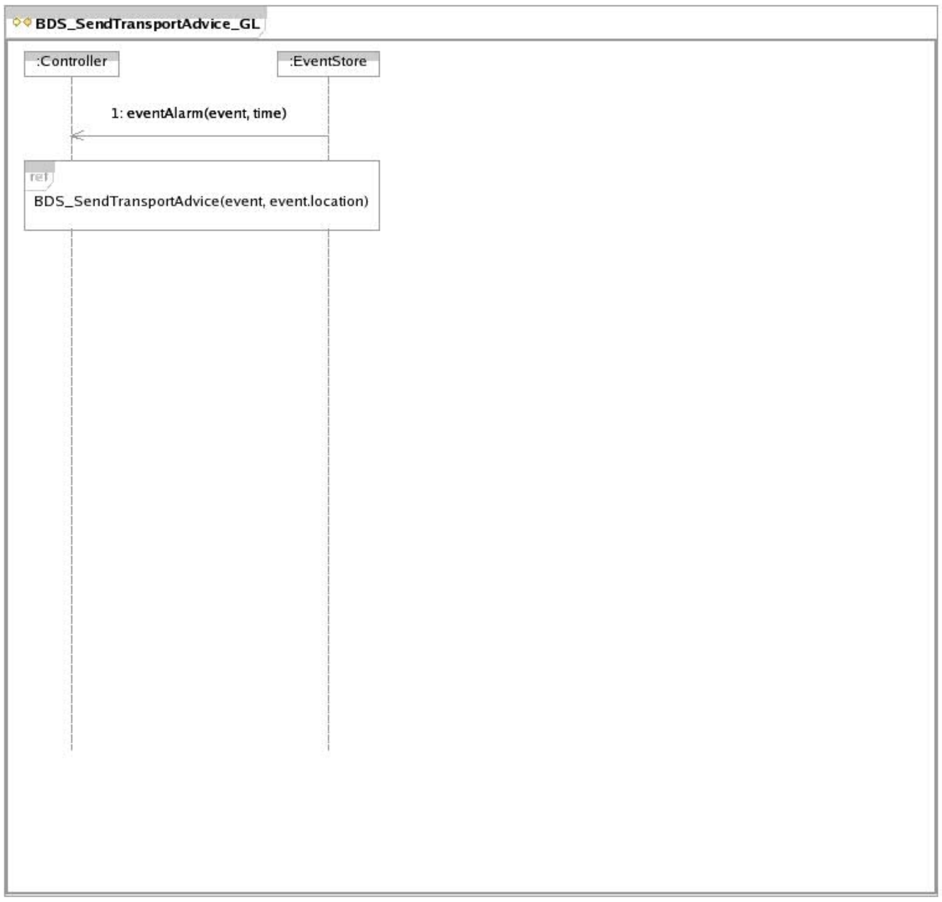

BDS_SendTransportAdvice_GL
:Controller
:EventStore
1: eventAlarm(event, time)
ref
BDS_SendTransportAdvice(event, event.location)

Figure 23: The “Send Transport Advice(e: Event, dst: Location) ” decomposition of the BDS lifeline

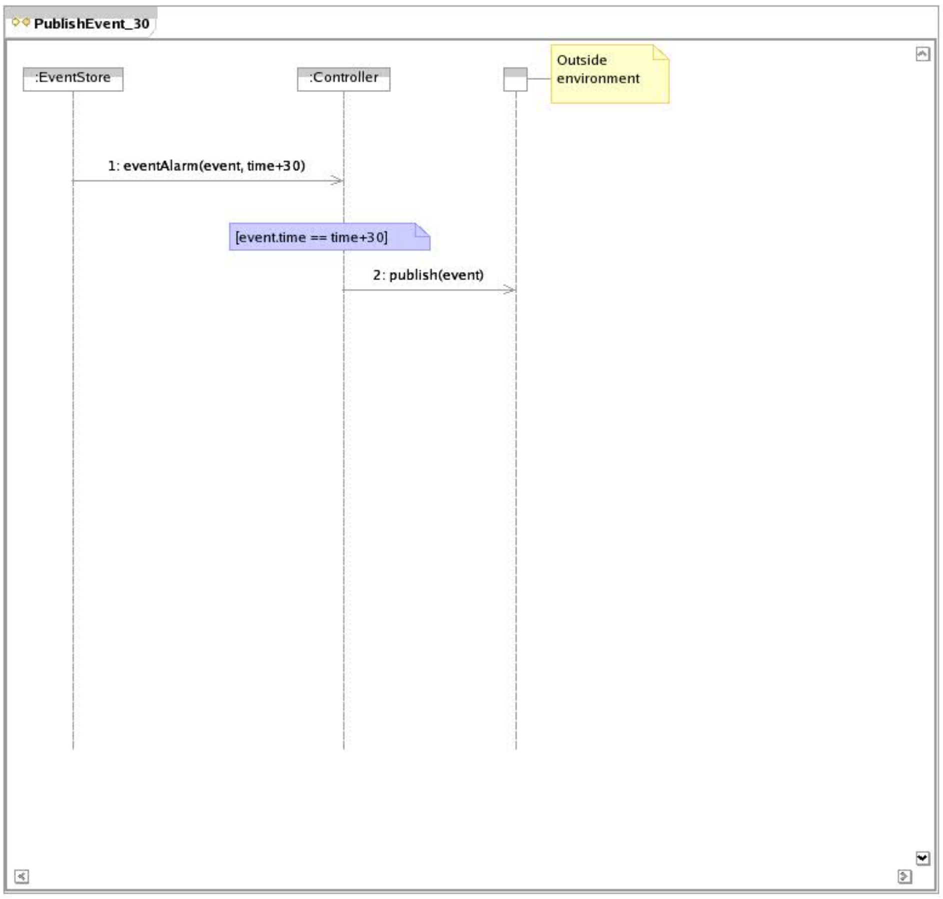


Figure 24: Publish Event with Time 30 min from Published Sequence Diagram

gets the current time and fixes the event time accordingly.

The system behaviour as specified by Join Event and Send Transport Advice will be unchanged in this refinement, and so the sequence diagrams of the previous section apply.

## 5 Refinement Arguments

We have modeled two different system, the first captured by the interaction overview of Figure 3.5.2, the second captured by the interaction overview of Figure 4.1.1.

The fist system specification contains the sequence diagrams Subscribe to Services, Publish Event, Join Event and Send Transport Advice. These diagrams will for simplicity be referred to as $SS$, $PE$, $JE$ and $STA$, respectively. $SS$ is processed in parallel with the sequential processing of the latter three. Let $S_1$ denote ($PE$ seq $JE$ seq $STA$). Our first system may now be denoted by $BDS_1 = (SS \text{ par } S_1)$.

The second system specification contains the sequence diagram $NS_1$ in which the traces of $S_1$ has been made negative. The specification furthermore contains the sequence diagrams $SS$, Publish Event_GL = $PE_{GL}$, Publish Event_30 = $PE_{30}$, $JE$, $STA$ and Send Transport Advice_GL = $STA_{GL}$. Let $S_2$ denote ($PE_{GL}$ seq $JE$ seq $STA_{GL}$) and $S_3$ denote ($PE_{30}$ seq $JE$ seq $STA$). Our second system may now be denoted by $BDS_2 = (SS \text{ par } (NS_1 \text{ alt } S_2 \text{ alt } S_3))$.

We will now prove that $BDS_2$ is a property refinement of $BDS_1$, i.e. that $BDS_1 \rightsquigarrow BDS_2$.

Notice first that by reflexivity of the refinement relation, $SE \rightsquigarrow SE$. The refinement relation is furthermore monotonic with respect to the par-operator, which means that in order to prove that $BDS_1 \rightsquigarrow BDS_2$, it suffices to prove that $S_1 \rightsquigarrow (NS_1 \text{ alt } S_2 \text{ alt } S_3)$

In order to do this we need to capture the positive and the negative traces of the sequence diagrams. Let $d$ be any sequence diagram. $pos(d)$ is then the set of positive traces of $d$, while $neg(d)$ is the set of negative traces of $d$.

The semantics of $S_1$, denoted $[\![S_1]\!]$, is the singleton set of interaction obligations $\{(p_1, n_1)\}$, where $p_1 = pos(S_1)$ and $n_1 = neg(S_1)$.

The semantics of $(NS_1 \text{ alt } S_2 \text{ alt } S_3)$ is also a singleton set of interaction obligations $\{(p_2, n_2)\}$, where $p_2 = pos(NS_1) \cup pos(S_2) \cup pos(S_3)$ and $n_2 = neg(NS_1) \cup neg(S_2) \cup neg(S_3)$.

By definition, the interaction obligation $(p_2, n_2)$ refines the interaction obligation $(p_1, n_1)$ iff $n_1 \subseteq n_2$ and $p_1 \subseteq p_2 \cup n_2$.

Since $neg(S_1) \subseteq neg(NS_1)$, $neg(S_1) = n_1$ and $neg(NS_1) \subseteq n_2$, we have $n_1 \subseteq n_2$. Since all the positive traces of $N_1$ has been made negative in $NS_1$, we have $p_1 \subseteq neg(NS_1)$. Clearly $neg(NS_1) \subseteq p_1 \cup n_2$, and so $p_1 \subseteq p_2 \cup n_2$.

The refinements of the general dating system are all property refinements. By adding the traces of $S_2$ and $S_3$, we did a supplementing as inconclusive behaviour of $S_1$ was made either positive or negative. By making all the positive traces of $S_1$ negative, we did a narrowing.

Notice finally that since $pos(NS_1) = \{\langle\rangle\}$, i.e. the set of positive traces of $NS_1$ contains the empty trace only, our refinements give that $p_2 \subseteq pos(SE)$. This is a flaw in the specification of $BDS_2$ that we have chosen not to deal with in this assignment.