University of Oslo
Department of Informatics

STAIRS case study: The BuddySync System

Ragnhild Kobro Runde

Research Report 345
ISBN 82-7368-302-8
ISSN 0806-3036

January 2007

STAIRS case study: The BuddySync System

Ragnhild Kobro Runde

Abstract

This paper presents a case study evaluating the use of STAIRS when specifying a system for connecting service providers and people requesting those services. As part of the case study, we give an example of how STAIRS may be used in combination with development methodologies like e.g. RUP. We conclude that STAIRS seems a promising method for working with UML 2.x interactions, and indicate some possible directions for future research.

1 Introduction

STAIRS is a method for the compositional development of interactions in the setting of UML 2.x [OMG06]. The main motivation for STAIRS is

- understanding the meaning of the different advanced interaction operators in UML 2.x, and how these should be used in specifications using interactions, and
- explaining how initial specifications in the form of interactions may be refined into more complete descriptions of the system under development.

In order to achieve this, STAIRS provides a denotational trace semantics for the main parts of UML 2.x interactions, as well as a set of refinement notions.

Previous papers on STAIRS ([HHRS05a], [HHRS05b], [RHS05b], [RHS05a] and [RRS06]) have mainly been concerned with explaining STAIRS through formal definitions. Although the papers contain various examples, these have all been constructed for specific illustrative purposes. In [RHS06] we gave a tutorial introduction to STAIRS, explaining the practical relevance of STAIRS through pragmatical guidelines.

This paper presents an evaluation of STAIRS based on a thorough case study performed by ourselves. From an informal description of a system, a specification in the form of UML 2.x interactions is developed using our general knowledge of STAIRS and in particular the guidelines from [RHS06]. This paper presents mainly the resulting specifications, and should not be understood as providing a complete documentation of the development process.

In this paper we have chosen to present only the case study, and not STAIRS itself. For material on STAIRS we refer to the above mentioned papers, and in particular [RHS06]. The remainder of this paper is structured as follows: In Section 2 we present our evaluation criteria for STAIRS. In Section 3 we present the BuddySync system, the subject of this case study. Section 4 presents the development methodology used, and Sections 5–7 present the resulting UML 2.x interactions specifying the system. In Section 8 we evaluate STAIRS with

respect to the evaluation criteria, before giving some concluding remarks in Section 9.

2 Evaluation criteria

In this section we list our evaluation criteria for STAIRS. The criteria are inspired by [KS03], a framework for understanding quality in conceptual modelling. For evaluating language quality, a distinction is made between criteria for the concepts in the underlying basis, and criteria for the syntactic constructs used to represent these concepts visually. Many of the criteria in [KS03] are criteria that apply to UML in general, such as criteria regarding the size, solidity and position of the various elements in UML 2.x interactions. These criteria are not relevant here, where we evaluate only what is STAIRS specific. For an evaluation of UML in general, see e.g. [Kro05].

The criteria we have chosen for evaluating STAIRS are divided into three categories as follows:

- Criteria for the conceptual basis of the language:
  1. All relevant knowledge should be expressible.
  2. The concepts should be general, limiting the total number of concepts.
  3. The concepts should be composable, so that related requirements may be grouped together.
  4. Both precise and vague knowledge should be expressible.
  5. The concepts should be easily distinguished from each other.
  6. A concept should mean the same thing every time it is used.
  7. The concepts should allow flexibility in the level of detail.
  8. It should be possible to divide the models into natural parts, enabling work division.
  9. The most frequent kinds of requirements should be expressible in a compact form.
- Criteria for the visual representation of the language:
  10. The mapping from syntactic constructs to the underlying concepts must be unambiguous.
  11. The constructs should be easily distinguished from each other.
  12. A construct should represent the same concept in all contexts.
  13. The constructs should be composable, in order to support the grouping of related requirements.
  14. Constructs without information should be avoided.
- Criteria with respect to refinement:
  15. The refinement relations should be powerful enough to capture all refinement steps made in practice.

16. The refinement relations should be general, limiting the total number of relations.
17. The refinement relations should be easily distinguished from each other.
18. It should be possible to refine the different parts of a specification separately.

As pointed our in [KS03], deficiencies in the language may be addressed by the methodology used. In our case, this means that we will also evaluate how the guidelines given in [RHS06] help with e.g. distinguishing the different concepts, constructs and refinement relations. For easy reference, these guidelines are included in Appendix A.

3 Initial description of the BuddySync system

In this section we describe the BuddySync system, based on the text for a mandatory assignment in the course INF-UIT at the University of Oslo autumn 2001. This has been chosen as the system for this case study for the following reasons:

- It is a different kind of system than those used as examples in our previous papers.
- Communication with different kinds of users is an essential part of the system.
- The assignment was created for using sequence diagrams (i.e. interactions), but not specifically tailored towards STAIRS (which did not exist at the time).
- Being a course assignment, the size of the system is manageable.

The BuddySync system is a system that helps connecting people that need a service with people that offer it. The idea is that users should be able to send an SMS to the system, either offering a service or requesting one. The system should then automatically match service requests and service offers. For some services, the system may also forward requests to potential service providers. Before requesting or offering a service, the user must be registered. Also, it should be possible to remove service requests and offers.

An example of a BuddySync service is taking a taxi, described as follows:

- A taxi offers its service as soon as it becomes available, stating its geographical position.
- A passenger requests a taxi. Either he is automatically assigned to one of the available taxis in his area, or he must wait for a taxi to become available.

Some initial example scenarios for this service is described by the sequence diagrams in Figures 1 and 2. In GetATaxi in Figure 1 the user requests a taxi (to position upos), and the taxi offers its service (at position tpos). The use of par indicates that either the user or taxi may initiate the interaction, or they may

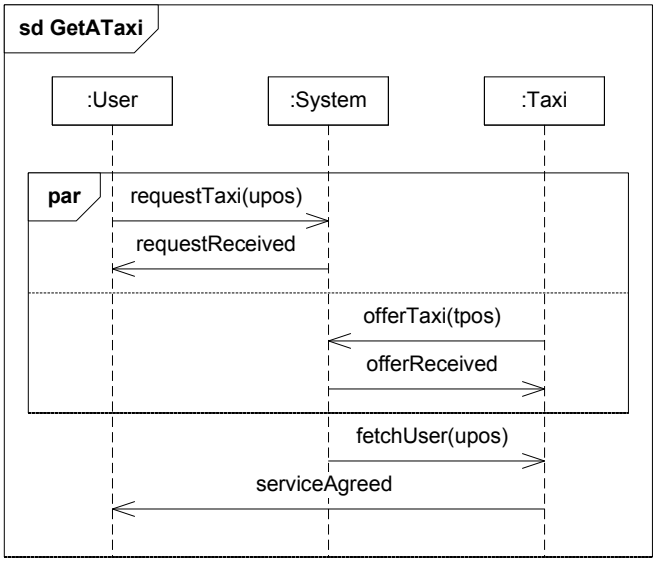

Figure 1: Example scenario for taxi service

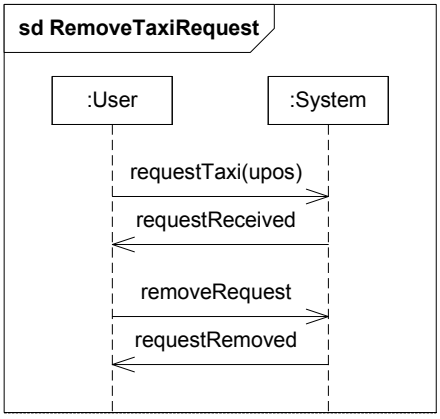

Figure 2: Example scenario for removing a taxi request

do so in parallel. After matching the request and the offer (implicitly comparing upos and tpos), the system tells the taxi to pick up the user at upos, and the taxi informs the user of its upcoming arrival.

In RemoveTaxiRequest in Figure 2, the user first requests a taxi, but then he withdraws the request. This may happen for instance if it takes too long before a taxi becomes available, and the user decides to go by the bus instead.

4 Development methodology

STAIRS is not intended to be a complete methodology for system development, but should rather be seen as a supplement to existing methodologies such as e.g. RUP [Kru04]. The purpose of this case study is to create a specification in the form of UML 2.x interactions, in order to perform an evaluation of STAIRS.

As this is not a real project, there are no customers and no budget involved. Also, there will be no actual implementation of the system. Hence, many parts of RUP are not relevant for this case study although they would have been in a real project for the same system.

Of the four main phases of RUP, the activities performed in this case study fit in as parts of the elaboration phase, which is the phase where the majority of the functional and non-functional requirements are specified. Each of the main phases of RUP consists of a sequence of iterations. RUP also defines nine disciplines that cut across the iterations. The disciplines relevant in the setting of this case study is the requirements discipline, where the goal is establishing an agreement with the customers as to what the system should do, and the analysis and design discipline, where the requirements are translated into a specification of how the system should be implemented.

From the informal requirements in Section 3, we plan three iterations each adding a few more features to the system (specification):

- **Iteration 1:** Requesting and offering services. This is the main functionality of the system, and a natural starting point. For this first iteration, we assume that all users are registered, and leave the actual checking of this to iteration 3.
- **Iteration 2:** Removing service requests and offers.
- **Iteration 3:** Subscribing and unsubscribing. First of all, this iteration should add (un-)subscription mechanisms to the system. Also, subscription checking should be added to the handling of service requests/offers from iteration 1.

Each of the above iterations should consist of the following activities:

- Specify the required functionality from the perspective of a system user.
- Specify how the system should implement the required functionality.
- Check the resulting interactions with respect to the guidelines given in [RHS06] for creating interactions (included here in Appendix A.1).
- Whenever there is more than one interaction specifying the same functionality, check that they are in a refinement relationship according to the guidelines in [RHS06] (Appendix A.2).

In a specification process such as the one described here, there will usually be some degree of trial and error, where e.g. alternative designs are explored and an increased understanding of the domain and the system to built results in modifications of diagrams made earlier in the process. These parts of the case study are not reported here, as we are mainly interested in the diagrams that are correct descriptions of the BuddySync system and the relationships between these diagrams.

5 Iteration 1: RequestService and OfferService

A general overview of the behaviours of the BuddySync system is given in Figure 3. All of the referenced interactions describe functionality that is required

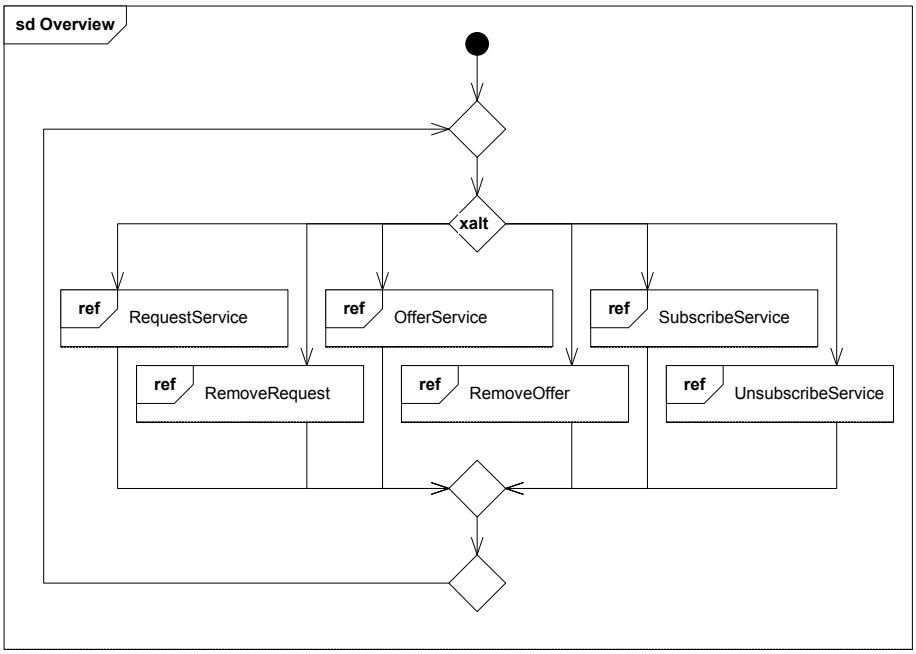

Figure 3: Overview of the BuddySync System

of the system. The decision with respect to which of the referenced interactions will be performed in each of the iterations depends on the given user input. Hence, the operator xalt is used according to the guidelines in Appendix A.1.1. For simplicity, we assume that the system handles only one user input at the time. For handling several users at the same time, the system should be able to perform arbitrary many instances of Figure 3 in parallel.

As described in Section 4, we start by specifying the main functionality of the system, that is how it supports the requesting and offering of services. As the system should support a wide range of services, and the exact selection of services may change over time, it is important that the specifications should be generic with respect to service. For this first iteration, we assume that all users are registered.

5.1 User requirements: RequestService

The composite diagram in Figure 4 gives the initial context for the BuddySync System. As illustrated, the system interacts with two different users, a requester and a provider, but these two never interact.

Figure 5 gives the initial specification of how a user interacts with the system when requesting a service. First, the user sends a request to the system, stating the requested service (e.g. taking a taxi) and detailed requirements (e.g. the time and place). These details are not important for our case study, so we have put them together in the parameter called service. As a reply to the request, the user should get either an agree-message indicating that a matching service provider has been found, or a message that the request is registered (waiting for a matching provider). Checking with the guidelines in Appendix A.1.1, we specify these alternatives using xalt as there is obviously some underlying

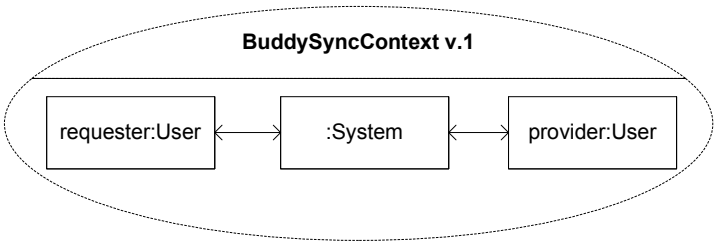

Figure 4: Composite structure for the BuddySync System

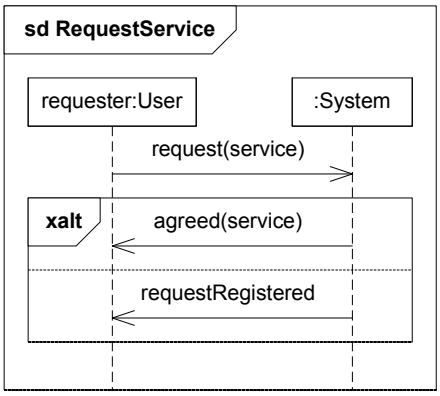

Figure 5: Request service — user view

condition in the system directing which one of these reply messages will be sent.

In Figure 6 we have added interaction with the provider (another user) to the specification in Figure 5. For readability and maintainability the two alternative system responses have been separated into the diagrams in Figures 7 and 8. If a match is found, the provider is notified via a perform-message, while in the case where the request is simply registered, the system may optionally also notify a potential provider of the requested service. (If the provider then chooses to offer the service, this should be treated as an ordinary service offer according to the specifications in Section 5.2.) The opt-construct is a shorthand for an alt between the given operand and the empty diagram (skip). Checking again with the guidelines in Appendix A.1.1, using alt (and not xalt) is correct here, as this is an instance of underspecification where we do not require both alternatives to present in an implementation.

The requester and the provider may be seen as two different interfaces of the system, of which Figure 5 only includes the requester while Figure 6 presents both interfaces. Comparing these two figures, it is straightforward to see that from the requester's perspective, the behaviour is the same. The notion of interfaces is not included in STAIRS. However, from the perspective of the requester, Figure 6 may also be understood as a detailing refinement of Figure 5 as it gives more details about how the system handles the request. In the terminology of Appendix A.2.3, this would then correspond to a decomposition of the system, with the following lifeline mapping:

$$ID[\text{provider:User} \mapsto \text{:System}]$$

i.e. the identity mapping ID updated so that provider:User maps to :System.

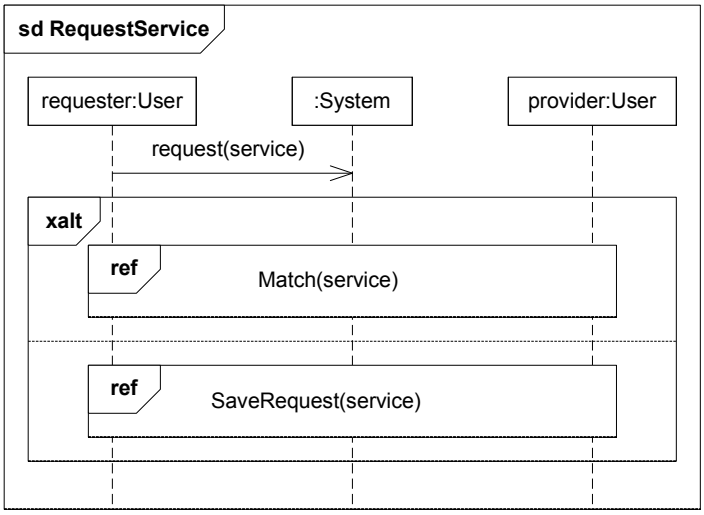

Figure 6: Request service — user view with provider

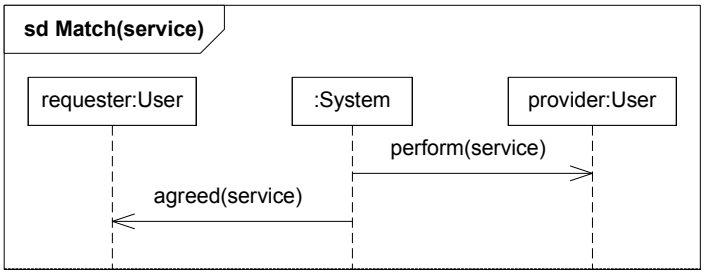

Figure 7: Matching service requester and provider

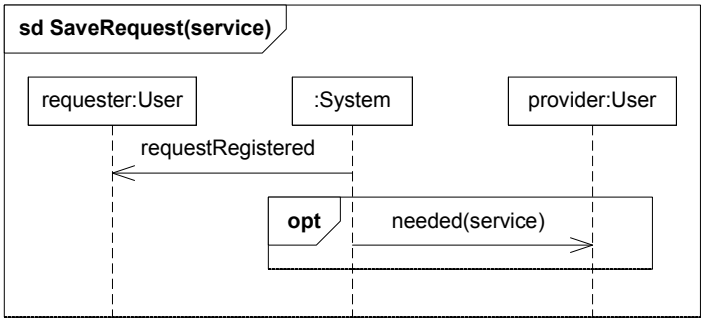

Figure 8: Saving request

5.2 User requirement: OfferService

We now move on to specify how a user may offer his service via the BuddySync system. The initial specification of how the user interacts with the system is given in Figure 9. First, the user sends an offer to the system, giving the details of the provided service as parameter. As a reply, the user should get either a perform-message indicating that the system has found a matching service request, or a message that the offer has been registered in the system. Again,

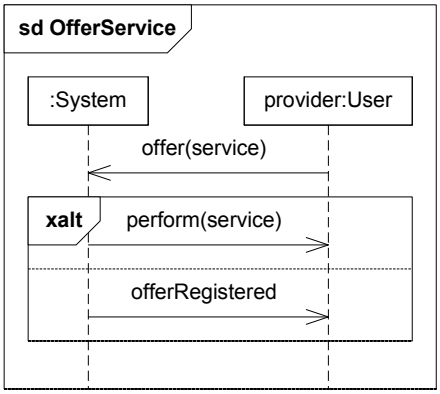

Figure 9: Offer service — user view

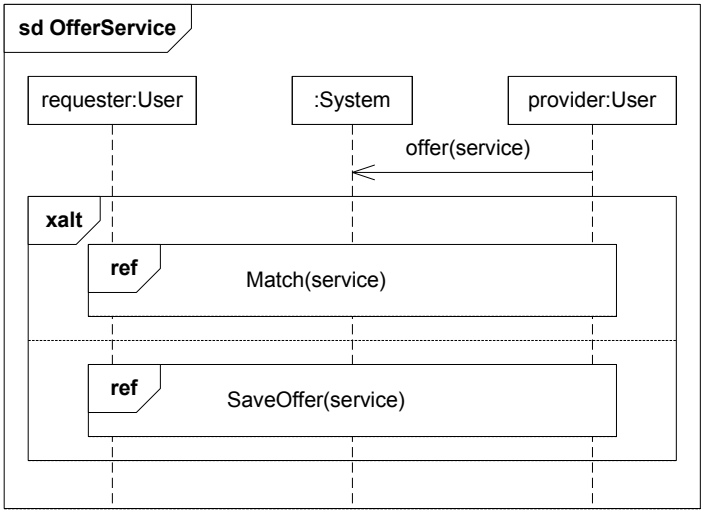

Figure 10: Offer service — user view with requester

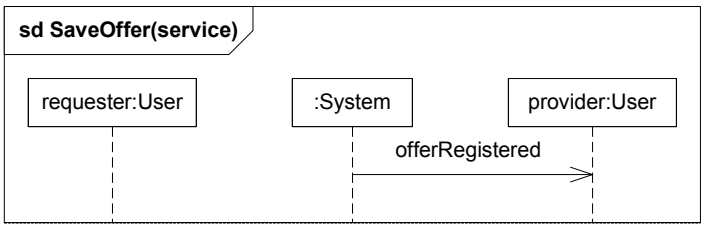

Figure 11: Saving offer

these alternatives are specified using xalt due to the underlying condition.

In Figure 10 interaction with the requester (another user) is added to the specification in Figure 9. If a match is found, the requester is notified via an agreed-message (in fact, here we reuse the specification of Match(service) in Figure 7), while in the case that the offer is simply saved, no interaction with the requester takes place as specified in Figure 11.

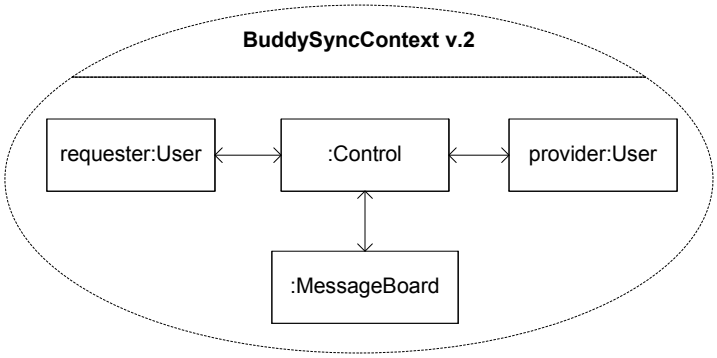

Figure 12: Revised composite structure diagram

Comparing Figures 10 and 9, it is again straightforward to see that from the provider's perspective the behaviour is the same, meaning that this may be understood as a detailing refinement with lifeline mapping:

$$ID[\text{requester:User} \mapsto \text{:System}]$$

5.3 System specification: RequestService

After having specified the functionality from the perspective of the user(s), we now turn to specifying how the system should implement this functionality. The revised composite structure diagram in Figure 12 illustrates how the system consist of two parts, a control and a messageboard. Only the control interacts with the users of the system. The messageboard should maintain a list of pending offers and a list of pending requests (not shown in the diagram).

In Figure 13, we take the specification of RequestService in Figure 6 and expand the system into control and messageboard.[1] Similarly, Figure 16 is an update of SaveRequest(service) in Figure 8. For the first xalt-operand in Figure 6, however, the reference to Match(service) is in Figure 13 replaced with a reference to MatchRequest(service) specified in Figure 14 (which again references the updated version of Match(service) in Figure 15).

Constraints are added to the beginning of MatchRequest (Figure 14) and SaveRequest (Figure 16), specifying the required conditions for performing each of them. The constraint s ≈ service takes into account that the match between a service request and a service offer does not need to be exact, as long as it is sufficiently similar. For instance, from its own position a taxi will usually have to drive for a few minutes before picking up its customer. In the context of Figure 13, the effect of the given constraints is exactly the same as if they instead had been added in the form of guards to the xalt-construct. In general, adding guards is a valid narrowing refinement according to Appendix A.2.2.

However, adding guards is not the only change introduced by the specifications in this section. With respect to Match(service) in Figure 7, MatchRequest(service) in Figure 14 also adds an internal message between the control and the messageboard, and an assignment on messageboard. As the

[1] In UML 2.x, such a decomposition is usually given by using the ref construct in the header of the system lifeline, and then giving the decomposition in a separate diagram. As STAIRS do not cover extra global combined fragments, the same effect is obtained by expanding the system lifelines directly in the diagram.

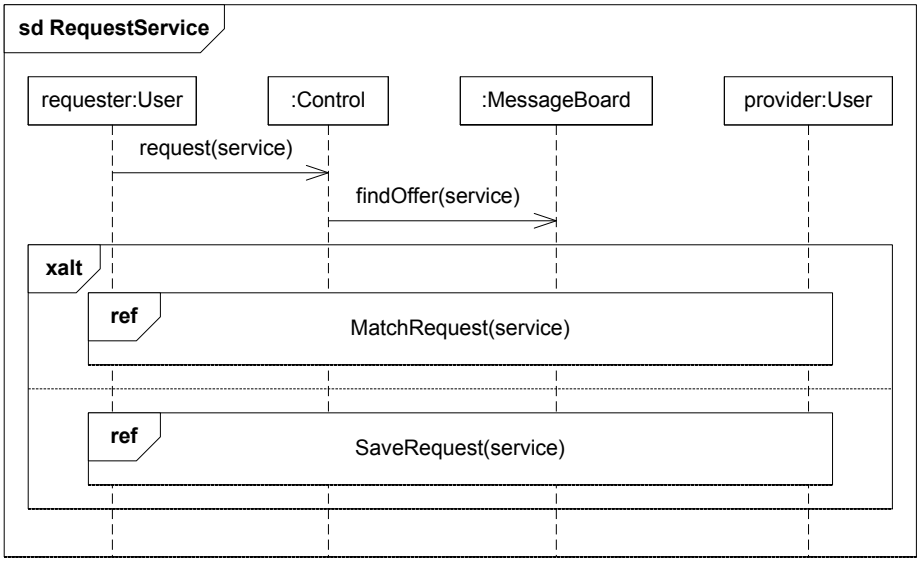

Figure 13: Request service — with control and messageboard

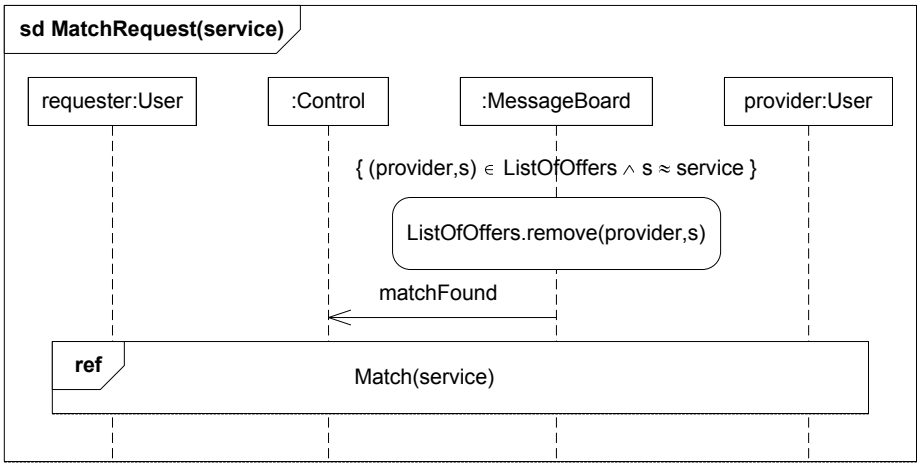

Figure 14: Matching service request with existing offer

communication with the users is the same, these changes constitute a detailing refinement according to Appendix A.2.3, with the lifeline mapping:

$$ID[\text{:Control} \mapsto \text{:System}][\text{:MessageBoard} \mapsto \text{:System}]$$

According to Appendix A.2.4, narrowing and detailing may be performed in a single step, making Figure 14 a valid refinement of Figure 7.

Similarly, SaveRequest(service) in Figure 16 is a narrowing and detailing refinement of Figure 8 with the same lifeline mapping. For Figure 13, we now have that the operands of xalt are refined separately, giving a valid refinement of the total xalt construct. As the only other change from the specification in Figure 6 is the addition of a message between control and messageboard, we have again a case of narrowing and detailing refinement with the same lifeline

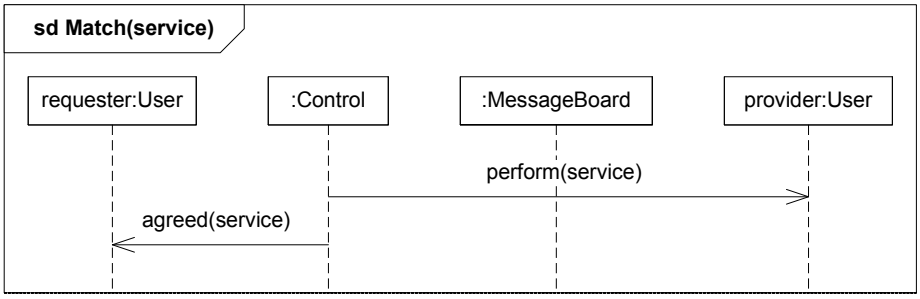

Figure 15: Matching service requester and provider — with control and messageboard

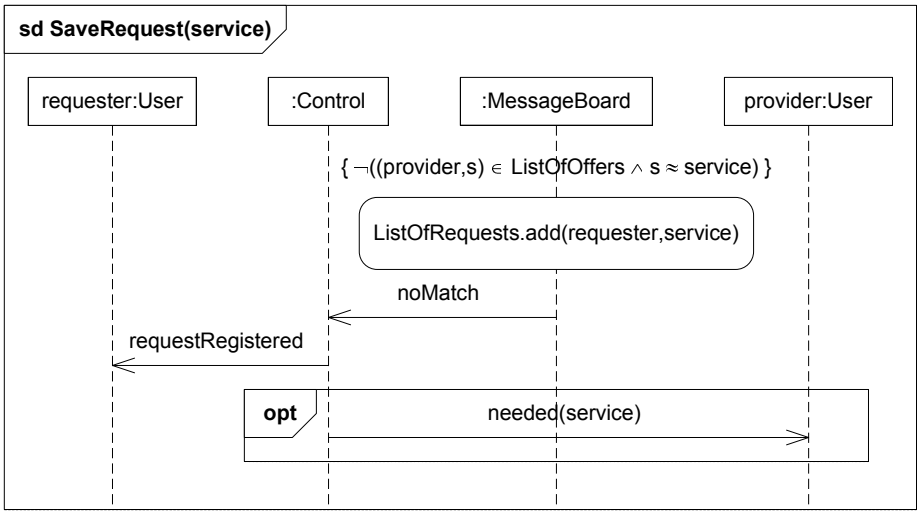

Figure 16: Saving request — with control and messageboard

mapping as above.

5.4 System specification: OfferService

In Figure 17, the specification of OfferService in Figure 10 is changed to account for the new system structure in Figure 12. Similarly, Figure 19 is an update of SaveOffer(service) in Figure 11. For the first xalt-operand in Figure 17, the earlier reference to Match(service) is replaced with a reference to MatchOffer(service) specified in Figure 18.

We now see that MatchOffer(service) in Figure 18 is another narrowing and detailing refinement of Match(service) in Figure 7 with the same lifeline mapping as in Section 5.3. Similarly, Figure 19 is a general refinement of Figure 11, and Figure 17 is a general refinement of Figure 10.

5.5 Finishing the iteration

Before ending the iteration, the resulting diagrams (Figures 13– 19) should be checked against the guidelines for creating interactions given in Appendix A.1.

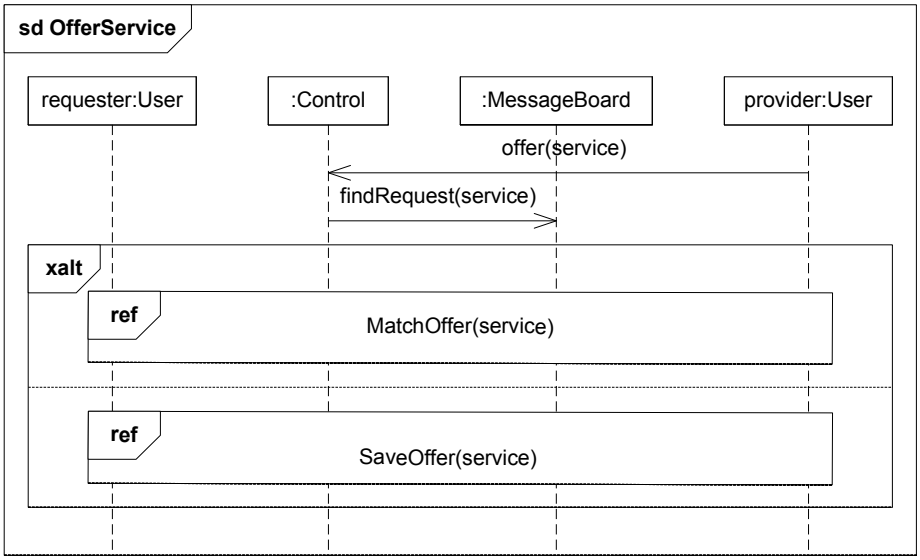

Figure 17: Offer service — with control and messageboard

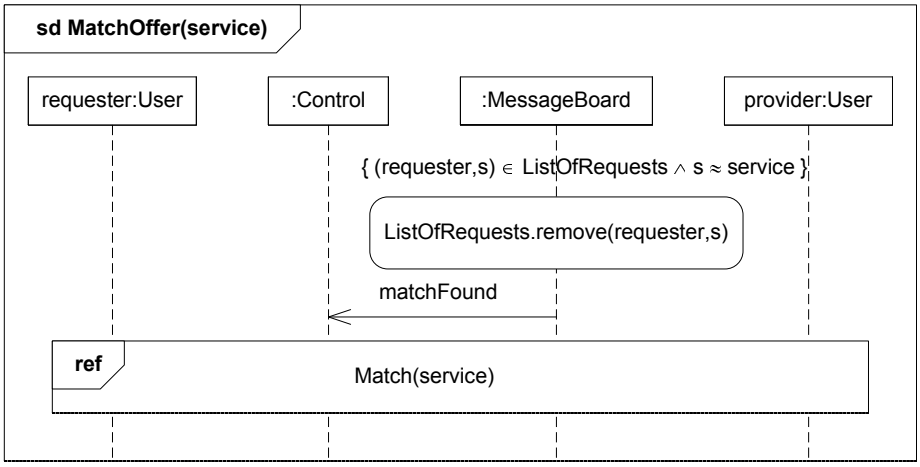

Figure 18: Matching service offer with existing request

For the choice between alt and xalt, this has been checked each time one of these was selected for use in the specifications. Concerning the guards (specified as constraints in Figures 14, 16, 18 and 19), the guidelines given in Appendix A.1.2 state that each guard should capture all possible situations for which the described traces are positive. Obviously, MatchRequest and MatchOffer may only be performed if there exists a corresponding offer, or request, at the messageboard. However, a reasonable question is whether it should be possible to do SaveRequest or SaveOffer also in the case of an existing match. After some considerations, we conclude that it would not be directly wrong, but less user-friendly and probably bad for business. Hence, the guards are left as they are.

Finally, the guidelines on negation given in Appendix A.1.3 state that the

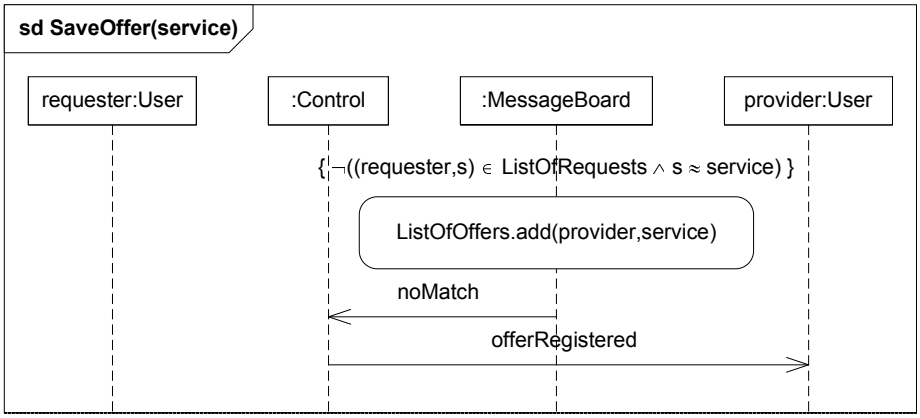

Figure 19: Saving offer — with control and messageboard

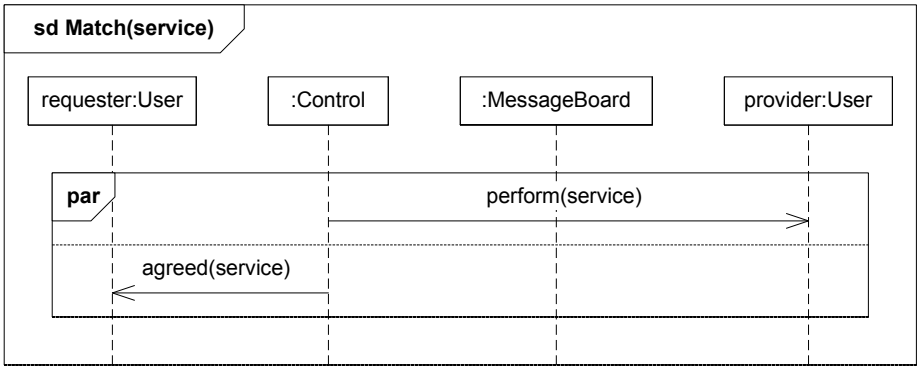

Figure 20: Match(service) after guideline-checking

specification should include a reasonable set of negative traces. In our specifications so far, the only negative traces we have are the traces where the constraints are false. This is clearly not sufficient. A simple way of adding negative traces to the specification would be to use assert as the top-level operator for RequestService and OfferService. However, before doing so we must be certain that all possible positive traces for these scenarios are described. We therefore consider each (sub-)diagram separately, trying to identify more positive and negative behaviours for that diagram.

For Match(service) in Figure 15, the order in which the system sends the messages to the requester and the provider is not important, so we add the par-operator as given in Figure 20. The same observation applies to SaveRequest(service) in Figure 16, where the new specification is given in Figure 21. According to the guidelines in Appendix A.2.1, these changes are valid supplementing refinements as all of the original traces (positive and negative) are kept, while new positive traces are added to the specifications.

For RequestService in Figure 13 and OfferService in Figure 17, it is quite obvious that the two given alternatives are meant to be exclusive. For instance, the alternative with MatchRequest should not be implemented with traces from SaveRequest and vice versa. There are at least two ways of fixing this. One possibility is to use assert on each operand (or on the specification as a whole).

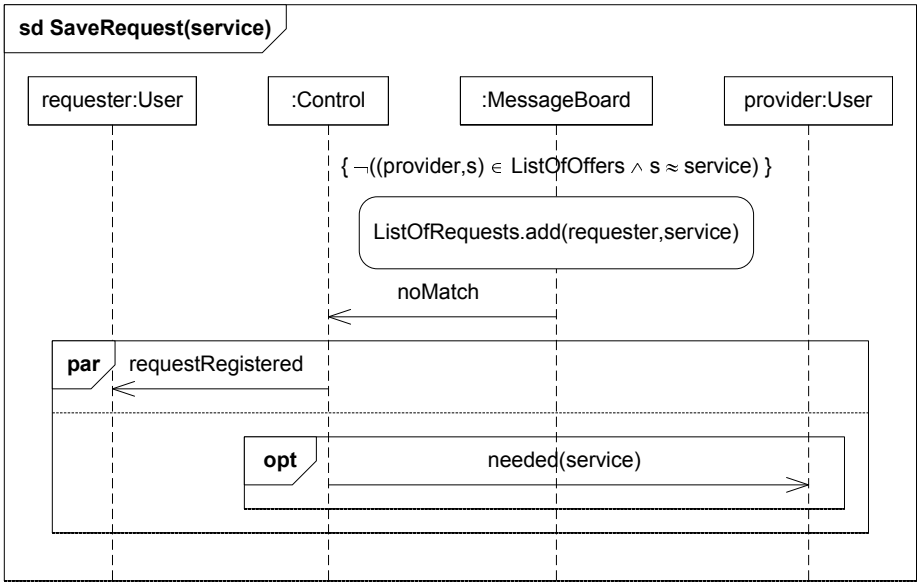

Figure 21: SaveRequest(service) after guideline-checking

However, we already know that the specifications are not complete, as we intend to add subscription checking in a later iteration. Instead, we have chosen to supplement the traces of one xalt operand as negative for the other operand, as shown in Figures 22 and 23. In these specifications, alt is used in order to specify positive traces in one operand and negative traces in another operand. For the negation operator, refuse is used in accordance with the guidelines in Appendix A.1.3.

To conclude this iteration, the specification now consists of the following diagrams:

RequestService	Figure 22
OfferService	Figure 23
MatchRequest(service)	Figure 14
MatchOffer(service)	Figure 18
Match(service)	Figure 20
SaveRequest(service)	Figure 21
SaveOffer(service)	Figure 19

6 Iteration 2: RemoveRequest and RemoveOffer

As described in Section 4, this iteration should specify support for removing service requests and service offers.

6.1 User requirements: RemoveRequest

Figure 24 specifies how a user interacts with the system when removing a service request. For the user, it is very important that the system really removes the request when asked to do so, hence the guidelines in Section A.1.3 tells us to use assert on this message.

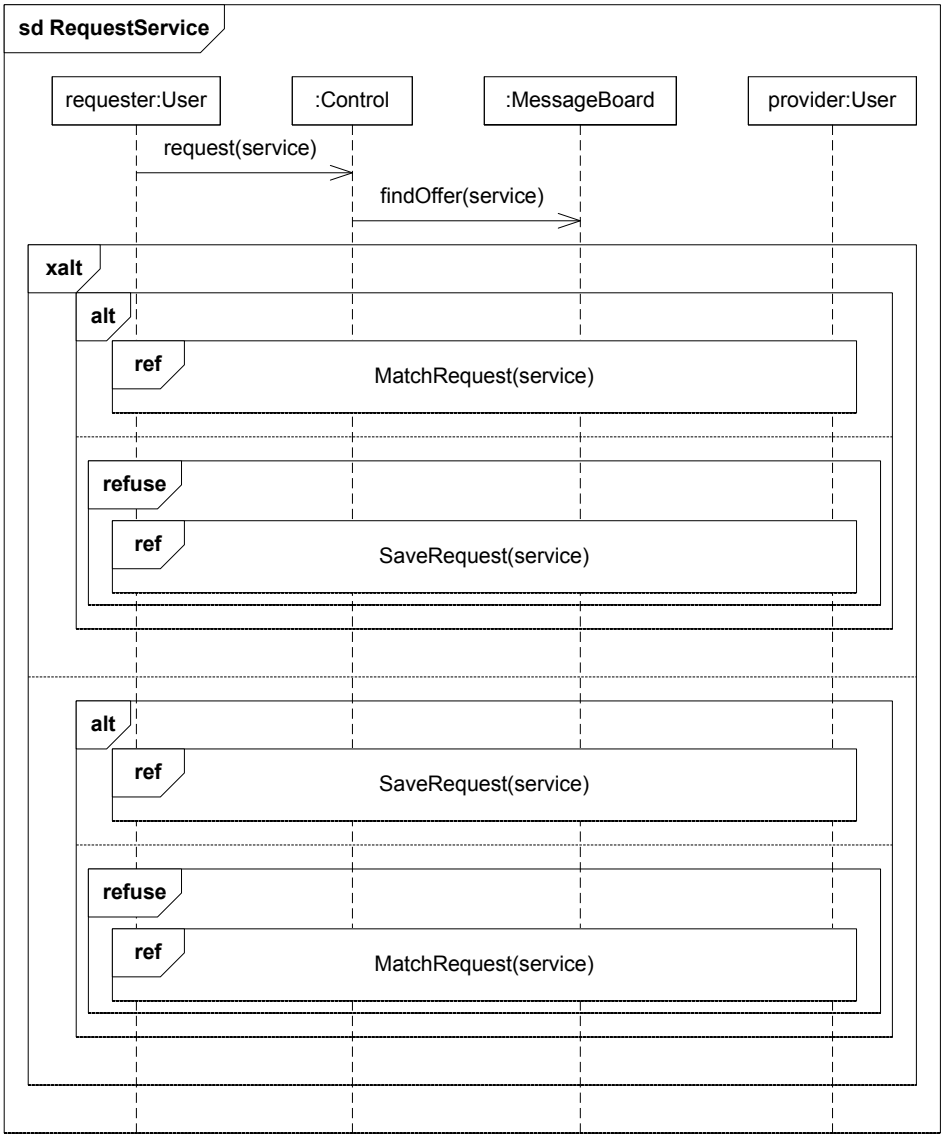

Figure 22: RequestService after guideline-checking

6.2 User requirements: RemoveOffer

Figure 25 specifies how a user interacts with the system when removing a service request. The specification is symmetrical to RemoveRequest in Figure 24.

6.3 System specification: RemoveRequest

After having specified the functionality from the perspective of the user(s), we now turn to specifying how this should be implemented by a system consisting of a control and a messageboard as given in Figure 12.

Figure 26 gives the specification of how the system should handle request removals. A difficult choice here, is deciding how much of the diagram should be

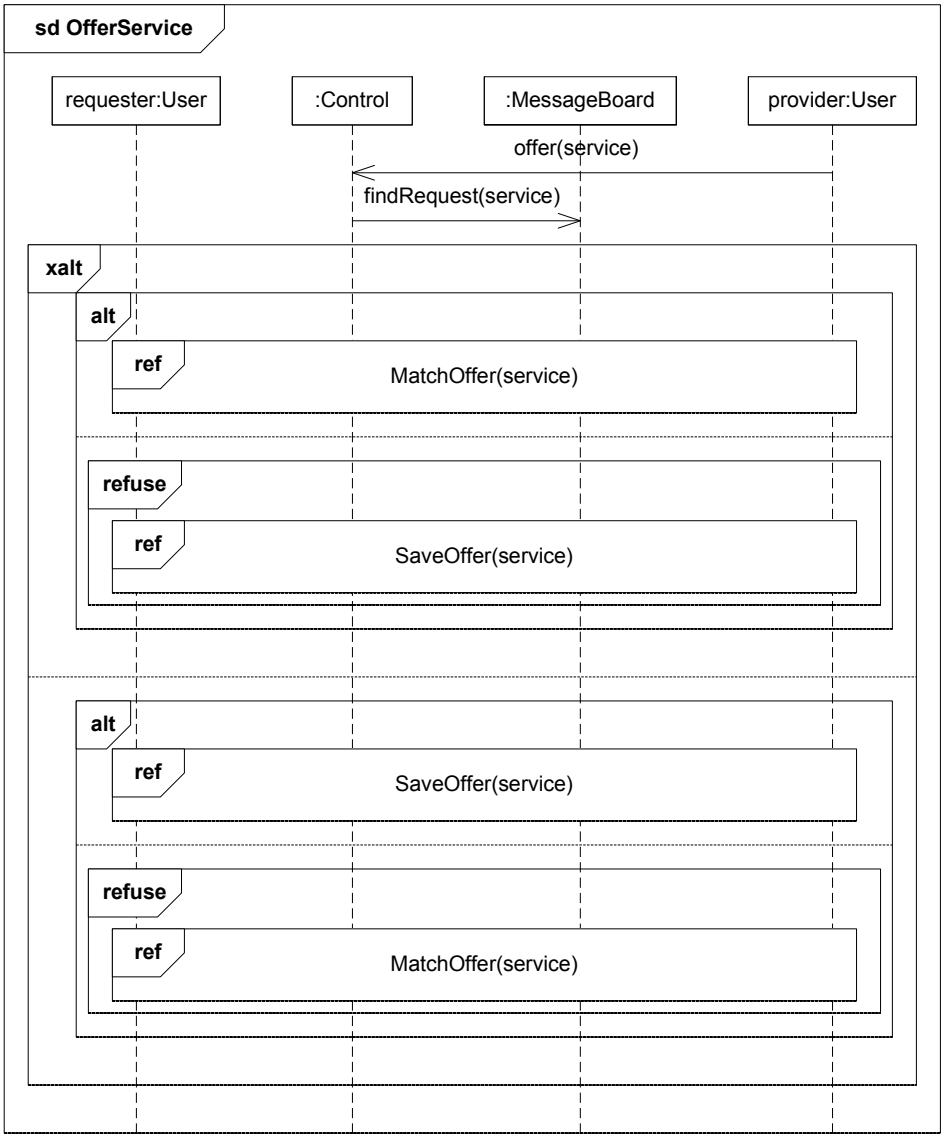

Figure 23: OfferService after guideline-checking

covered by the assert operator. It is not obvious that the one message from control to messageboard together with the assignment on messageboard is the only possible way for the system to handle the removal. For instance, an acknowledgment message back from messageboard to control is now forbidden. However, it is important for us to require that the remove-assignment is actually performed, and the simplest way of doing that is using assert as shown.

As before, comparing the specification in Figure 26 with the original specification if Figure 24, it should be fairly straightforward to see that the user interaction is the same. According to Appendix A.2.3, Figure 26 is then a detailing refinement of Figure 24 with the lifeline mapping:

$$ID[\text{:Control} \mapsto \text{:System}][\text{:MessageBoard} \mapsto \text{:System}]$$

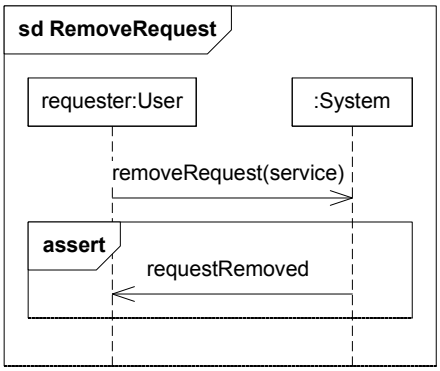

Figure 24: Remove request — user view

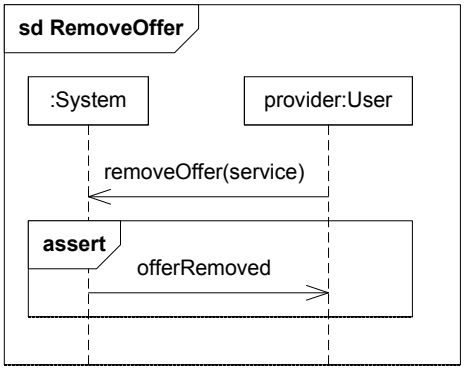

Figure 25: Remove offer — user view

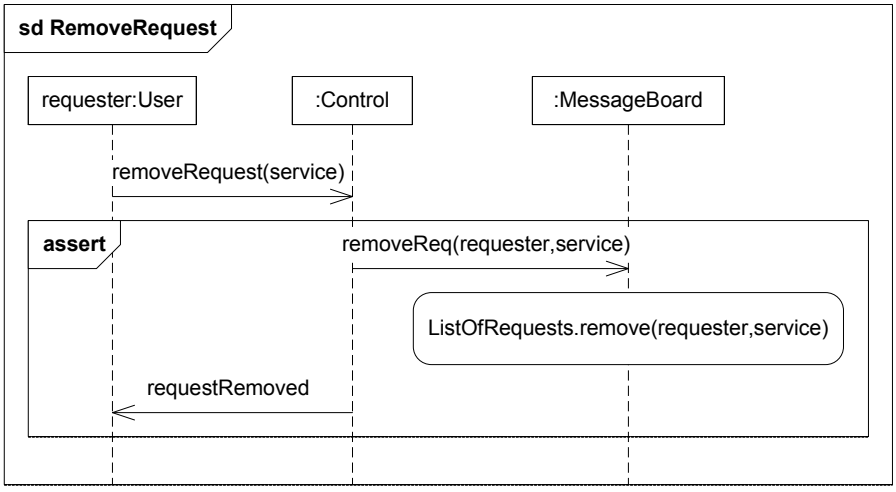

Figure 26: Remove request — with control and messageboard

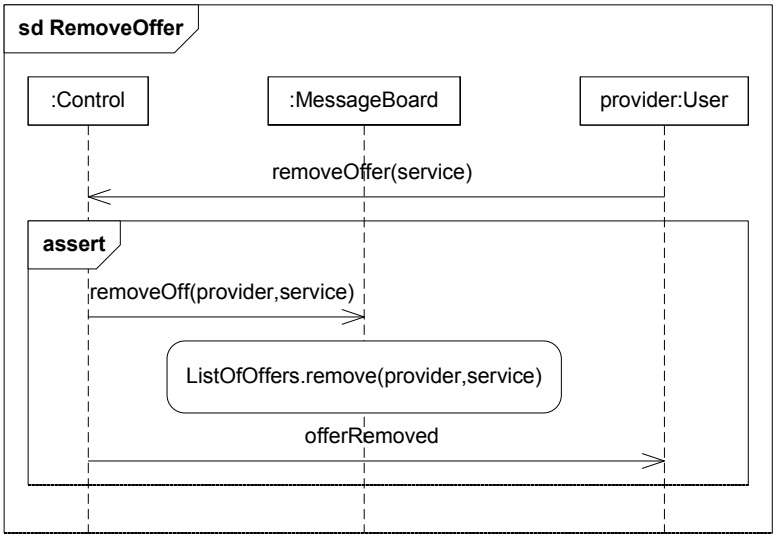

Figure 27: Remove offer — with control and messageboard

6.4 System specification: RemoveOffer

Figure 27 gives the specification of how the system should handle offer removals. Again, this is symmetrical to the specification of RemoveRequest in Figure 26. As the user interaction is the same, Appendix A.2.3 gives that Figure 27 is a detailing refinement of Figure 25 with the lifeline mapping:

$$ID[\text{:Control} \mapsto \text{:System}][\text{:MessageBoard} \mapsto \text{:System}]$$

6.5 Finishing the iteration

Before ending the iteration, the resulting diagrams (Figures 26 and 27) should be checked against the guidelines for creating interactions given in Appendix A.1. These specifications contain no alternatives or guards, meaning that the guidelines in Appendices A.1.1 and A.1.2 are not relevant. According to the guidelines in Appendix A.1.3, the specification should include a reasonable set of negative traces in order to effectively constrain an implementation of the system. In the interactions in Figures 26 and 27 this is ensured by the use of assert. As a conclusion, we leave these interactions as they are.

Also, no changes have been made to the system that affects the specifications from the previous iteration.

7 Iteration 3: SubscribeService and UnsubscribeService

As described in Section 4, in this final iteration we should add subscription and unsubscription mechanisms to the system. First, we decide that the subscription should apply to a single service only, and not to the complete system. This facilitates the same user being subscribed as a requester of one service, and

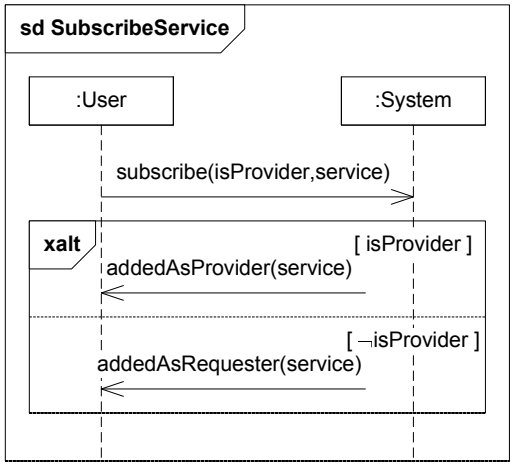

Figure 28: Subscribe service — user view

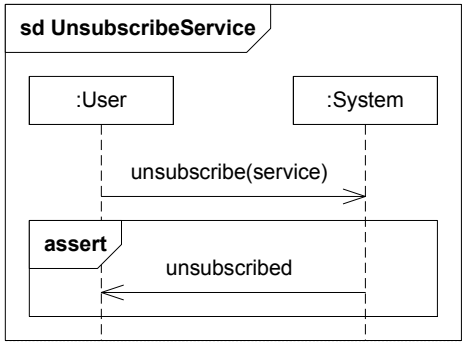

Figure 29: Unsubscribe service — user view

as a provider of another service. Also, there are cases where the user may be subscribed as both requester and provider of the same service.

7.1 User requirements: SubscribeService

Figure 28 specifies how subscription may look from the perspective of the user. Here, isProvider is a boolean flag indicating whether this is a subscription as a provider or as a requester (if the flag is false). If the subscription is as a provider, the user should get the message addedAsProvider as a receipt, if the subscription is as a requester the receipt message should be addedAsRequester. As these are alternatives with conditions, xalt are used according to the guidelines in Appendix A.1.1.

7.2 User requirements: UnsubscribeService

Figure 29 specifies unsubscription from the perspective of the user. As unsubscription should always be possible, assert is used on the receipt message unsubscribed back from the system to the user.

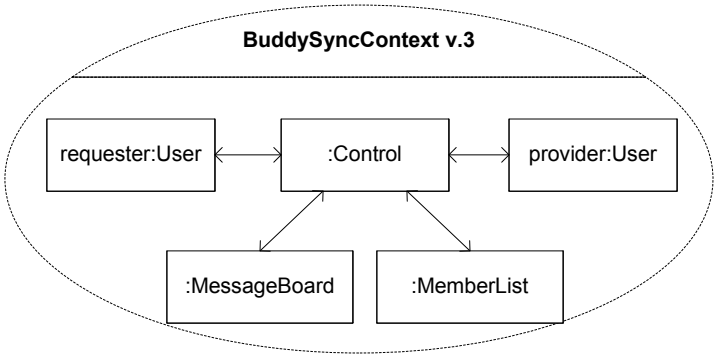

Figure 30: Revised composite structure diagram

7.3 System specification: SubscribeService

Again, we turn to how the system should implement the required functionality. To handle subscriptions, we add a new component MemberList to the system as illustrated by the composite structure diagram in Figure 30. MemberList should maintain a list of service providers and a list of service requesters (not shown in the diagram).

Figure 31 specifies how SubscribeService in Figure 28 should be implemented by the system. The messageboard is not relevant for subscriptions, but the control and memberlist are. As before, we have that the communication with the user stays the same, meaning that Figure 31 is a detailing refinement with the lifeline mapping:

$$ID[\text{:Control} \mapsto \text{:System}][\text{:MemberList} \mapsto \text{:System}]$$

7.4 System specification: UnsubscribeService

Figure 32 specifies how UnsubscribeService in Figure 29 should be implemented by the system. Again, Appendix A.2.3 gives that this is a detailing refinement with the same lifeline mapping as in Section 7.3.

7.5 System specification: RequestService updated

As noted in Section 4, the specification of RequestService should be updated so that only subscribed users may request services. This is achieved by the specification in Figure 33.

Compared to the previous specification in Figure 22, messages are added between Control and MemberList in order to check for subscription. As a result of this checking, the specification includes a new alternative where the message notSubscribed is sent to the requester. This alternative is also specified with xalt according to the guidelines in Appendix A.1.1, as it should be the alternative when the necessary subscription is not found. Finally, the alt+refuse constructs used in Figure 22 are removed, and an assert is added to the complete specification.

We now have to check that Figure 33 is a valid refinement of Figure 22. First of all, we note that it will have to be a detailing refinement, with the lifeline

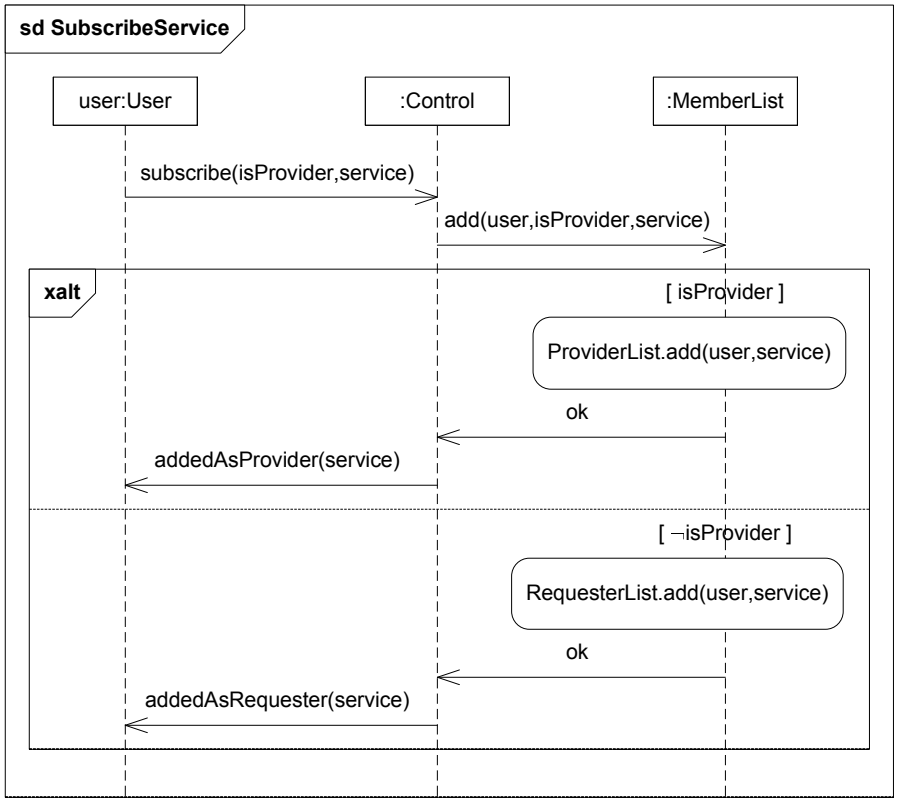

Figure 31: Subscribe service — with control and memberlist

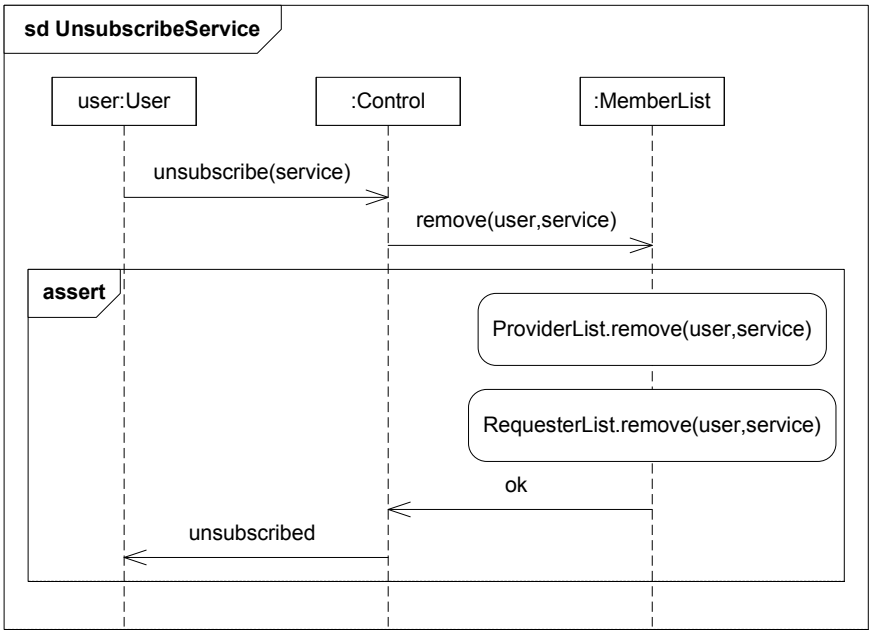

Figure 32: Unsubscribe service — with control and memberlist

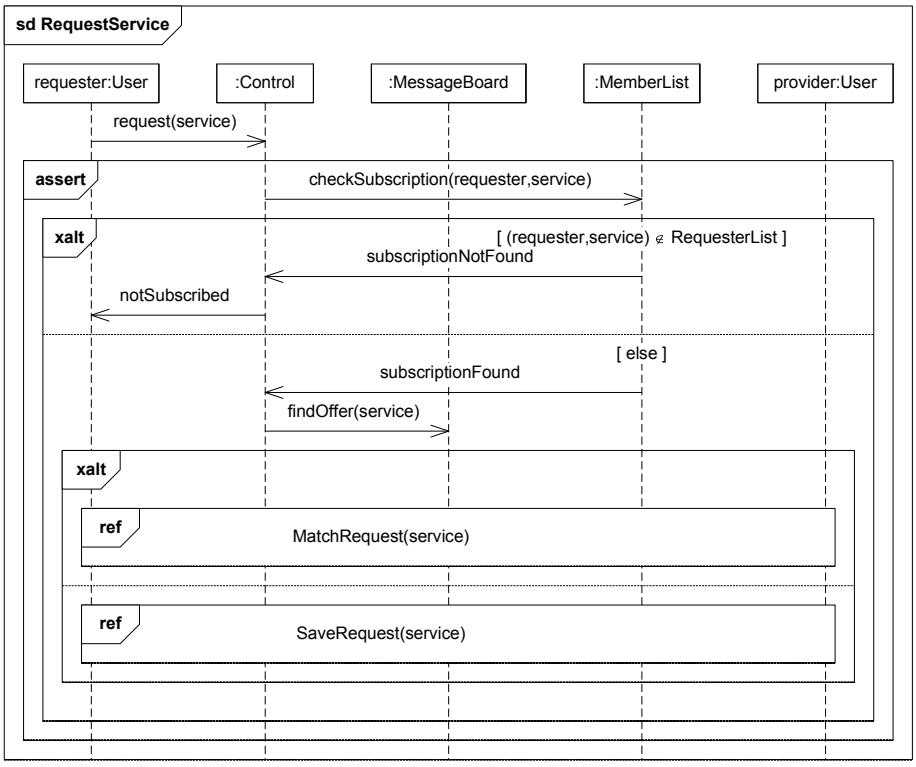

Figure 33: Request service — with subscription checking

mapping:

$$ID[\text{:MemberList} \mapsto \text{:Control}]$$

A standard way of checking for refinement is to separately take each xalt-operand of the original diagram, and find a refining xalt-operand in the new diagram.

The first xalt-operand in Figure 22 corresponds to a scenario that starts with the messages request(service) and findOffer(service), and then performs MatchRequest(service) and *not* SaveRequest(service). This scenario is found in Figure 33 by choosing the second operand of the outer xalt, and then the first operand of the inner xalt. Here, SaveRequest is not explicitly negative, but its traces still become negative due to the outer assert. As all original positive and negative traces are kept, according to the guidelines in Appendix A.2.1 this is a valid supplementing refinement where the assert construct adds more negative traces to the specification.

Similarly, the second xalt-operand in Figure 22 is found in Figure 33 by choosing the second operand of both xalt's, making this another instance of supplementing refinement.

Finally, we notice that Figure 33 also adds a new xalt-alternative to the specification, which is a valid refinement step.

To conclude, Figure 33 is a valid general refinement of Figure 22 combining supplementing and detailing. This is also a limited refinement, as the first xalt-operand in Figure 33 specifies behaviour that is inconclusive in Figure 22.

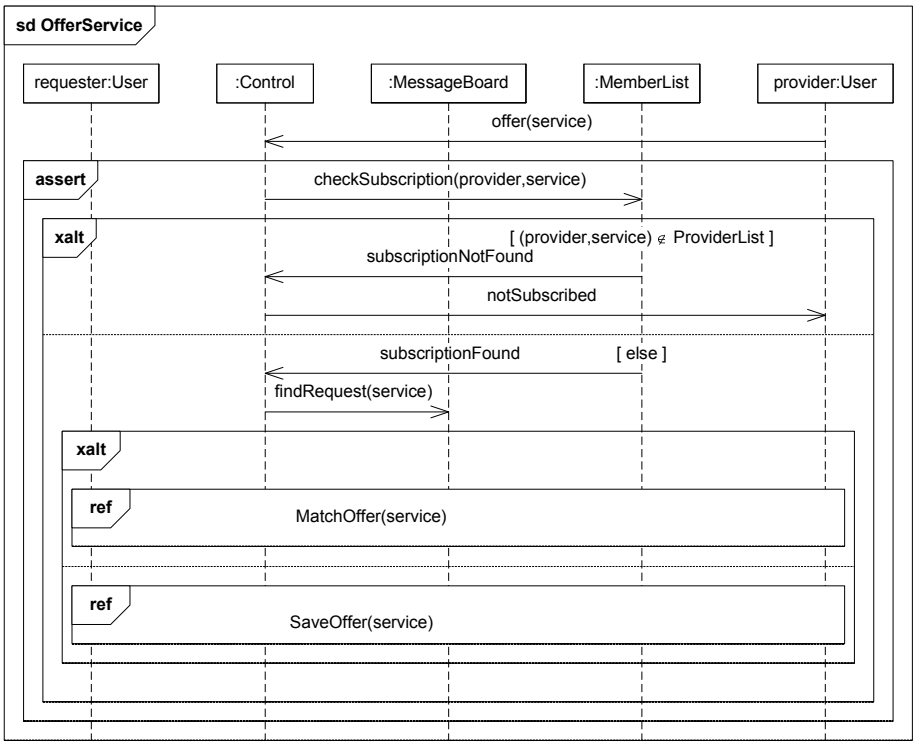

Figure 34: Offer service — with subscription checking

7.6 System specification: OfferService updated

Also OfferService must be updated with subscription checking. This is done in Figure 34. With an argument similar to the one performed for RequestService in the previous section, we may conclude that Figure 34 is a valid general refinement of Figure 23 combining supplementing and detailing. It is also a valid limited refinement.

7.7 Finishing the iteration

Before ending this iteration, the last one, we take a final check that the resulting diagrams (Figures 31, 32, 33 and 34) are as recommended by the guidelines in Appendix A.1.

For the choice between alt and xalt, this has been checked each time one of these has been used in the specifications. Concerning the guards in Figure 31 these are obviously sufficiently covering as required by the guidelines in Appendix A.1.2. Adding a user as provider should only be performed if the isProvider flag is true, and adding him as a requester should only be performed if the flag is false. For the guards in Figures 33 and 34, these are also correct as the error message notSubscribed should be a result only if the user is not subscribed to the service, and continuing to handle the request or offer should be done only if the user is subscribed.

With respect to negation (Appendix A.1.3), Figures 32, 33 and 34 all contain assert, meaning that the specification also includes a number of negative traces as recommended. Figure 31 contains some negative traces (the traces with a

false guard), but we could consider adding an assert here as well. For simplicity, the resulting diagram is not shown here as it is not important for the case study.

8 Discussion

8.1 Validating the specification

In this section we validate the final specification with respect to the initial example scenarios given in Section 3. As the interactions in Figures 1 and 2 do not contain any negative behaviours, every possible interaction will be a valid general refinement of these two. For validating that our specification is in accordance with the intention behind Figures 1 and 2, we must instead use informal reasoning.

In Figure 2, the user first requests a taxi, and then removes the request again. Taking the message requestTaxi(upos) as an instance of the generic message request(service), requestReceived as requestRegistered and removeRequest as removeRequest(service), the given sequence may be found in the specification by first performing RequestService as specified in Figure 33, choosing the alternative with SaveRequest, and then performing RemoveRequest as specified in Figure 26.

In Figure 1, a user requests a taxi in parallel (or sequence) with a taxi offering its service, after which the system notifies both parts. Even with the natural translation of the messages, this scenario is not found as positive (but rather as negative due to the use of assert) in our final specification. The reason for this is that if the user's request comes first, this will already be registered when the offer from the taxi arrives, meaning that according to our specification the taxi does not need a receipt-message. Instead he is immediately notified with a message to actually perform the service. This means that although the original scenario is not explicitly covered, the combination of RequestService (Figure 33) and OfferService (Figure 34) results in a scenario that fulfils the same purpose as the one intended in Figure 1, and we may conclude that the final specification is in accordance with the initial example scenarios.

8.2 Evaluating STAIRS

In this section we evaluate STAIRS with respect to the evaluation criteria in Section 2, based on the specification of the BuddySync system as presented in Sections 5–7.

1. **All relevant knowledge should be expressible.**

   For the BuddySync system, we have encountered no problems expressing the desired functional requirements. In Section 3, it is suggested that the communication between the system and its users is performed via SMS. This is not described in the interactions, but could have been included by e.g. using the name UserMobile instead of User for the requester and the provider lifelines.

2. **The concepts should be general.**

All concepts used are general in the sense that they are not tailored towards this particular application, but may be used to capture a number of different requirements.

3. **The concepts should be composable.**

   Using the various compositions operators (in particular opt (alt), xalt, refuse and assert), it has been easy to group alternative positive behaviours for the same system functionality, and also to specify the related negative behaviours together with the positive behaviours.

4. **Both precise and vague knowledge should be expressible.**

   In this case study, vague knowledge has primarily been expressed using abstraction, i.e. by not describing system details that are irrelevant or not known. For instance, it is not described how the messageboard and memberlist maintain their lists or the exact nature of these lists (sets, unordered/ordered sequences, ...).

   Another way to describe vague knowledge in STAIRS is using underspecification (i.e. alt), describing possible alternative behaviours for the system.

5. **The concepts should be easily distinguished from each other.**

   STAIRS includes two operators, alt and xalt, for describing alternative behaviours. Using the guidelines in Appendix A.1.1, it has been easy to decide which one of these to use in each particular situation.

   We also have three operators, assert, refuse and veto, for describing negative behaviours. The difference between assert and the two others are obvious, but there is only a small difference between refuse and veto. However, using the guidelines in Appendix A.1.3, it has been easy to select between them.

   In the specification of e.g. RequestService in Figure 13, the constraints in the referenced interactions MatchRequest(service) (Figure 14) and SaveRequest(service) (Figure 16) are interpreted as guards. From this, it follows that the distinction between constraints and guards is not obvious. Also, the similarities and differences between these two concepts are not covered by any of the guidelines in Appendix A.

6. **A concept should mean the same thing every time it is used.**

   The meaning of each concept is independent of the context in which it is used. As can be seen from the guidelines in Section A.1.1, xalt may be used in different situations, but the underlying semantics is always that all alternatives must be reflected in the final implementation.

7. **The concepts should allow flexibility in the level of detail.**

   Our different specifications of the same functionality (e.g. Figures 5, 6, 13, 22 and 33 of RequestService) demonstrates that the constructs are suitable for creating interactions with a varying degree of detail.

8. **It should be possible to divide the models into natural parts.**

In the specification of the BuddySync system, one interaction was created for each main part of the functionality. In addition, sub-interactions were created wherever natural in order to increase readability and the possibility of reuse.

9. **The most frequent kinds of requirements should be expressible in a compact form.**

   The use of xalt-alt-refuse together with ref as illustrated in Figures 22 and 23, is a common pattern in STAIRS specifications, specifying that the positive behaviours of the first xalt-operand should be negative for the second xalt-operand and vice versa. This combination of operators is not at all compact, and adding a new high-level operator for this use should be considered in future work on STAIRS.

   Apart from this, we find that all requirements are expressed fairly compact.

10. **The mapping from syntactic constructs to the underlying concepts must be unambiguous.**

    In STAIRS, there is a one-to-one mapping between the main concepts and the syntactic constructs used to express these.

11. **The constructs should be easily distinguished from each other.**

    STAIRS does mainly use the syntax of UML 2.x interactions, which are not part of what is being evaluated here. The STAIRS operators refuse and veto are easily distinguished syntactically. The STAIRS operators alt and xalt are more similar, but we have never experienced any problems with these either. The 'x' in xalt makes the difference between xalt and alt clearly visible.

12. **A construct should represent the same concept in all contexts.**

    This follows from the one-to-one mapping between constructs and concepts.

13. **The constructs should be composable.**

    All of the advanced interaction operators of UML 2.x are composable in the sense that they may be nested to an arbitrary depth. The same applies to the specific STAIRS operators.

14. **Constructs without any information should be avoided.**

    In the specification of the BuddySync system, none of the interactions contain any informationless construct.

    In general, it would probably be easy to make constructs without any information if that was the aim. However, using common UML techniques and the guidelines in Appendix A, we believe that all resulting constructs will provide meaningful information.

15. **The refinement relations should be powerful enough to capture all refinement steps made in practice.**

    For the BuddySync system, all interactions specifying the same functionality could be related using the notion of refinement. With respect to

the relation between e.g. Figures 5 and 6, this may be seen as a detailing refinement. However, the guidelines in Appendix A.2.3 associate detailing with decomposition, which is a term that intuitively does not fit very well with respect to these two figures. A possible solution to this could be to extend the guidelines in order to capture that detailing may mean more than just decomposition.

16. **The refinement relations should be general.**

    All refinement relations used are general in the sense that they are not defined for this particular application, but may be used for a number of different specifications.

17. **The refinement relations should be easily distinguished from each other.**

    Using the guidelines in Appendix A.2, it is easy to distinguish between supplementing, narrowing and detailing. As demonstrated by the Buddy-Sync specifications, most refinement are a combination of these, and it is easy to find out which of the refinement relations that explains each of the changes in the interactions.

    However, the distinction between general and limited refinement is not clear from the guidelines, which gives no help for e.g. finding out whether OfferService in Figure 34 is a limited refinement of OfferService in Figure 23 or not.

18. **It should be possible to refine the different parts of a specification separately.**

    Separate refinement is possible due to the monotonicity results in previous papers on STAIRS, and we have used this for refining the different system functionality separately, and also for performing separate refinement of sub-interactions such as e.g. SaveRequest(service) (Figures 8, 16 and 21) and SaveOffer(service) (Figures 11 and 19).

    However, this is not explicitly covered by the guidelines in Appendix A.2. For relating the specifications of RequestService in Figures 22 and 33, we used that for general refinement we may take each xalt-operand of the original diagram and find a refining xalt-operand in the second diagram. This is also not covered by the guidelines. Implicitly, we also used the fact that with general refinement, new xalt-operands may be added freely to the specifications.

The main difficulty we encountered during the specification of the Buddy-Sync system, was how and when to use assert. For instance, it was not clear if it could be used in the specifications of RequestService and OfferService in Figures 22 and 23. Also, it was difficult to decide on how much should be covered by assert in Figures 26, 27 and 32.

9 Conclusions

The presented case study has demonstrated the usefulness of the STAIRS method. As argued in Section 8.2, most of the evaluation criteria from Section 8.2 have

been met. In particular, the guidelines given in [RHS06] proved to be very useful, but they did not cover everything needed for the specifications in this case study. In particular, the following guidelines would have been useful:

- General refinement: Except from the operands of assert, all operands in an interaction may be refined separately.
- General refinement: With general refinement, all xalt-operands of the original interaction must be reflected in the refinement, but new xalt-operands may be added freely.
- Limited refinement: With limited refinement, new xalt-operands may be added to the interaction only if the specified behaviour is a refinement of some behaviour specified in the original interaction. In particular, behaviour that is not described by the original interaction (i.e. inconclusive behaviour) may be added in the new xalt-operands.

We have also identified the need for more guidelines with respect to constraints and guards, and the use of assert. As it is not obvious what the necessary guidelines are, we leave the formulation of these to future work. Also, a natural next step for future research on STAIRS would be to use it in a real world project where an actual implementation is created.

Acknowledgements

We thank Øystein Haugen and Ketil Stølen for useful feedback on previous versions of this paper.

References

[HHRS05a] Øystein Haugen, Knut Eilif Husa, Ragnhild Kobro Runde, and Ketil Stølen. STAIRS towards formal design with sequence diagrams. *Journal of Software and Systems Modeling*, 22(4):349–458, 2005.

[HHRS05b] Øystein Haugen, Knut Eilif Husa, Ragnhild Kobro Runde, and Ketil Stølen. Why timed sequence diagrams require three-event semantics. In *Scenarios: Models, Transformations and Tools*, volume 3466 of *LNCS*, pages 1–25. Springer, 2005.

[Kro05] John Krogstie. Quality of UML. In *Encyclopedia of Information Science and Technology (IV)*, pages 2387–2391. Idea Group, 2005.

[Kru04] Philippe Kruchten. *The Rational Unified Process: An Introduction.* Addison-Wesley, third edition, 2004.

[KS03] John Krogstie and Arne Sølvberg. *Information Systems Engineering: Conceptual Modeling in a Quality Perspective.* Kompendieforlaget, Trondheim, Norway, 2003.

[OMG06] Object Management Group. *UML 2.1 Superstructure Specification*, document: ptc/06-04-02 edition, 2006.

[RHS05a] Ragnhild Kobro Runde, Øystein Haugen, and Ketil Stølen. How to transform UML neg into a useful construct. In *Norsk Informatikkonferanse NIK'2005*, pages 55–66. Tapir, 2005.

[RHS05b] Ragnhild Kobro Runde, Øystein Haugen, and Ketil Stølen. Refining UML interactions with underspecification and nondeterminism. *Nordic Journal of Computing*, 12(2):157–188, 2005.

[RHS06] Ragnhild Kobro Runde, Øystein Haugen, and Ketil Stølen. The pragmatics of STAIRS. In *Proc. 4th Int. Symposium on Formal Methods for Components and Objects (FMCO'05)*, volume 4111 of *LNCS*, pages 88–114. Springer, 2006.

[RRS06] Atle Refsdal, Ragnhild Kobro Runde, and Ketil Stølen. Underspecification, inherent nondeterminism and probability in sequence diagrams. In *Proc. Formal Methods for Open Object-Based Distributed Systems (FMOODS 2006)*, volume 4037 of *LNCS*, pages 138–155. Springer, 2006.

A Guidelines from "The Pragmatics of STAIRS"

For easy reference, this appendix includes the guidelines given in [RHS06].

A.1 The pragmatics of creating interations

A.1.1 The pragmatics of alt vs xalt

- Use alt to specify alternatives that represent similar traces, i.e. to model
  - underspecification.
- Use xalt to specify alternatives that must all be present in an implementation, i.e. to model
  - inherent nondeterminism, as in the specification of a coin toss.
  - alternative traces due to different inputs that the system must be able to handle;
  - alternative traces where the conditions for these being positive are abstracted away.

A.1.2 The pragmatics of guards

- Use guards in an alt/xalt-construct to constrain the situations in which the different alternatives are positive.
- Always make sure that for each alternative, the guard is sufficiently general to capture all possible situations in which the described traces are positive.
- In an alt-construct, make sure that the guards are exhaustive. If doing nothing is valid, specify this by using the empty diagram, skip.

A.1.3 The pragmatics of negation

- To effectively constrain the implementation, the specification should include a reasonable set of negative traces.
- Use refuse when specifying that one of the alternatives in an alt-construct represents negative traces.
- Use veto when the empty trace (i.e. doing nothing) should be positive, as when specifying a negative message in an otherwise positive scenario.
- Use assert on an interaction fragment when all possible positive traces for that fragment have been described.

A.2 The pragmatics of refining interaction

A.2.1 The pragmatics of supplementing

- Use supplementing to add positive or negative traces to the specification.
- When supplementing, all of the original positive traces must remain positive and all of the original negative traces must remain negative.
- Do not use supplementing on the operand of an assert.

A.2.2 The pragmatics of narrowing

- Use narrowing to remove underspecification by redefining positive traces as negative.
- In cases of narrowing, all of the original negative traces must remain negative.
- Guards may be added to an alt-construct as a legal narrowing step.
- Guards may be added to an xalt-construct as a legal narrowing step.
- Guards may be narrowed, i.e. the refined condition must imply the original one.

A.2.3 The pragmatics of detailing

- Use detailing to increase the level of granularity of the specification by decomposing lifelines.
- When detailing, document the decomposition by creating a mapping L from the concrete to the abstract lifelines.
- When detailing, make sure that the refined traces are equal to the original ones when abstracting away internal communication and taking the lifeline mapping into account.

A.2.4 The pragmatics of general refinement

- Use general refinement to perform a combination of supplementing, narrowing and detailing in a single step.
- To define that a particular trace *must* be present in an implementation use xalt and assert to characterize an obligation with this trace as the only positive one and all other traces as negative.

A.2.5 The pragmatics of limited refinement

- Use assert and switch to limited refinement in order to avoid fundamentally new traces being added to the specification.
- To specify globally negative traces, define these as negative in all operands of xalt, and switch to limited refinement.