Available online at www.sciencedirect.com
SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science

ELSEVIER Electronic Notes in Theoretical Computer Science 115 (2005) 19–38
www.elsevier.com/locate/entcs

Combining Sequence Diagrams and OCL for Liveness

Alessandra Cavarra[1]

*Computing Laboratory, University of Oxford
Wolfson Building, Parks Rd, Oxford OX13QD
England*

Juliana Küster-Filipe[2]

*Laboratory for Foundations of Computer Science
School of Informatics, University of Edinburgh
King's Buildings, Edinburgh EH9 3JZ
Scotland*

Abstract

Sequence diagrams in UML 2.0 have been considerably extended, partially influenced by variants of Message Sequence Charts (MSCs) and Live Sequence Charts (LSCs). However, sequence diagrams cannot satisfactorily express liveness/progress properties or differentiate between *necessary* and *possible* behaviour. To address this limitation, we propose to use an OCL template for liveness and enrich sequence diagrams with constraints as needed. We argue that our extended sequence diagrams are more expressive than LSCs. Further, if automatic code generation from inter-object behaviour specifications is feasible it will lead to more realistic solutions.
Additionally, we discuss several problems and ambiguities in sequence diagrams as defined in the UML 2.0 superstructure specification. We discuss directions for future work.

Keywords: Sequence Diagrams, UML 2.0, OCL 2.0, Liveness

[1] Email:Alessandra.Cavarra@comlab.ox.ac.uk
[2] Email:jkfilipe@inf.ed.ac.uk

1 Introduction

One fundamental concern when designing reactive systems is to capture how the different components in the system interact. Amongst the various existing formalisms to describe interactions, scenario-based approaches are preferred by software engineers because they capture more naturally the way people think about interactions. Commonly used notations include Message Sequence Charts (MSCs) [16] and their UML variant Sequence Diagrams [14]. However, being able to move from models of interaction given in either of these languages to automatically generated code, although desirable, is many steps too far. The reason for this is that these languages still have a very weak expressive power. In this paper, we recall some of the fundamental problems of sequence diagrams and–when possible–try to solve them in the light of Live Sequence Charts (LSCs) [5], a powerful scenario-based formalism.

LSCs allow us to distinguish between *possible* and *mandatory* behaviour, and express forbidden behaviour also called anti-scenarios. Since they were first introduced in [5] several aspects have been added including time, symbolic instances and classes. We do not consider these special features here but focus on basic LSCs with assignments, conditions, subcharts and forbidden elements. A detailed description of LSCs and their (partial) implementation in a tool called the Play Engine can be found in [9].

In UML 2.0, sequence diagrams have been considerably revised and partially influenced by LSCs. Nonetheless, sequence diagrams cannot satisfactorily express liveness/progress properties or differentiate between *necessary* and *possible* behaviour. Moreover, it is not possible to express that if a message is sent it *must* be received, if a particular sequence of messages is executed then a specific interaction *must* always occur, and so on. To address this limitation, we propose to use an OCL template for liveness and enrich sequence diagrams with liveness constraints as needed. As an implication, basic LSCs can be modelled using extended sequence diagrams. The reverse is, however, not true since sequence diagrams in UML 2.0 have several new features which do not exist for LSCs. We will mention some of these new features throughout the paper.

The paper is organised as follows. In Section 2 we introduce the main aspects of LSCs. A discussion on UML 2.0 sequence diagrams in relation with LSCs is presented in Section 3. In Section 4 we propose the OCL template for liveness and define suitable constraints to enrich sequence diagrams with liveness and synchronisation properties. Finally, in Section 5 we discuss further problems and limitations of sequence diagrams and anticipate directions for future work.

2 Live Sequence Charts

In [5] Damm and Harel discuss the weak expressive power of Message Sequence Charts (MSCs) and in particular their inability to model mandatory scenarios as well as forbidden ones. As a result, the authors propose a powerful and elegant extension of MSCs called Live Sequence Charts (LSCs). In this section we give an overview of the main features of LSCs, focusing on those aspects that differentiate them from their predecessors and in particular from UML Sequence Diagrams. For further details, we refer the reader to [5,9].

At the basic level, LSCs contain the objects involved in the interaction, represented by a box with their name from which a vertical line, called instance, departs. Messages going between instances are represented by horizontal arrows (see Fig. 1). A chart can be partitioned into a number of subcharts.

Every instance line contains *locations* which mark the occurrence of some events. All instances have at least three locations: an initial location, a location corresponding with the beginning of the main chart, and a final location. Locations are also associated with the sending and receiving of messages and the beginning and the end of subcharts. The locations along a single lifeline are ordered top-down; therefore, every LSC induces a *partial order* among locations which determines the order of execution. Notice, however, that messages m3 and m4 in Fig. 1 are not restricted by the partial order and can therefore occur in any order.

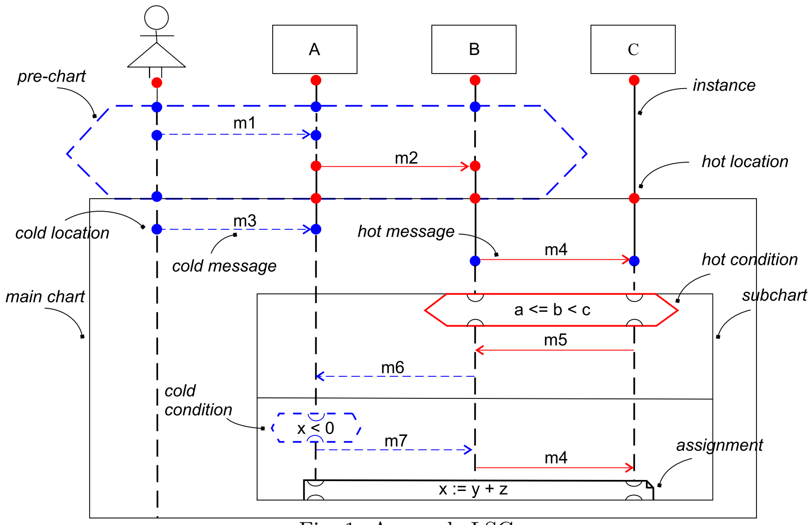

Fig. 1. A sample LSC

Possibility and necessity

MSCs, as well as UML sequence diagrams, can only express that under certain circumstances a certain scenario is *possible* but not *mandatory*. This limitation is solved in LSCs where a basic distinction is made between what *may* happen and what *must* happen both *globally* at the chart level and *locally* at the message and location level.

LSCs include two kinds of charts: *universal* and *existential*. Universal charts are used to specify restrictions that apply to all possible system runs. Usually universal charts contain a *prechart* specifying a precondition scenario. Basically universal charts impose a constraint on the system: for all system runs, if the system behaves according to the prechart, then it must behave according to the chart body. Universal charts are indicated using solid borderlines whilst precharts are annotated with a dashed hexagon (see the example LSC in Fig. 1). Existential charts are similar to message sequence charts and annotated with a dashed borderline. Existential charts never have a prechart.

At the local level, LSCs distinguish between *hot* and *cold* messages, and *hot* and *cold* locations. Hot messages, denoted by solid line arrows, must be received after they are sent. Cold messages, denoted by a dashed arrows, may be sent and never received. Finally, a hot location forces the instance to progress, whilst a cold location enables the instance to remain in its location without violating the chart. Notationally, hot locations are drawn as red circles, cold notations as blue circles.

Conditions and Assignments

Instances can be annotated with *cold* and *hot* conditions. If a cold condition evaluates to true, the chart proceeds to the location that immediately follows the condition; if it is false, then the surrounding (sub-)chart is immediately exited. Hot conditions must be true, otherwise the requirements are violated and the system aborts.

The notation for conditions is an hexagon, as shown in Fig. 1. As a condition can relate to more than one instance, the hexagon can be stretched along several instances.

In order to be able to deal with objects' attribute values, LSCs provide an *assignment* construct. The notation for assignment is shown in Fig. 1. As with conditions, an assignment symbol can be stretched along all the instances it refers to.

Instances involved in the definition of a condition or assignment are annotated by semi-circle connectors at the intersection points of the condition/assignment symbol with the instances line. For example, in Fig. 1 instances `B` and `C` are involved in the hot condition; `A` and `C` but not `B` are involved in the last assignment.

Synchronisation

There are several ways of achieving synchronisation in LSCs. The beginning of a prechart and the main chart are default synchronisation points. All instances enter the prechart simultaneously, while the main chart can be entered only after all the instances have successfully completed their activities in the prechart. In the same way, the beginning and the end of a subchart are synchronisation points for all participating instances. As a result, a subchart can be entered only when all the instances that participate in it arrive at the subchart initial location and can be exited only when all those instances have completed their activities (or if it contains a condition that evaluates to false).

In general, there is no partial order between the events and the condition in the chart body. Therefore, as soon as an instance reaches a condition, if this evaluates to true, the instance can progress beyond the condition. LSCs allow to synchronise the progress of one or several objects with the evaluation of a condition, that is, none of the synchronised instances may progress beyond the condition until all of them reach it and it is actually evaluated.

Finally, instances can be synchronised with an assignment. Again none of the synchronised instances may progress beyond the assignment until all of them reach it and it is actually performed.

Synchronised instances are annotated by a semi-circle connectors at the intersection points of the condition/assignment with the instance line. For example, in Fig. 1 instances `B` and `C` are synchronised on the hot condition; `A` and `C` but not `B` are synchronised upon the last assignment.

Forbidden behaviour

LSCs allow the definition of *anti-scenarios* corresponding to scenarios that the system should not be allowed to execute. This is achieved through universal charts where the forbidden scenario is specified in the prechart and the main chart consists only of the hot condition `FALSE`. If the forbidden scenario is successfully executed, the main chart is activated and the hot condition evaluated. Since this condition is always guaranteed to fail, it causes a violation.

To model that a specific message is not allowed to be sent at a given point within a chart, LSCs allow the definition of *forbidden messages*. Forbidden messages, together with their scope chart, are declared inside a special area located at the bottom of the LSC. They can be hot or cold: if a hot forbidden message is sent while the chart is in the forbidden scope, then the requirements are violated; if a cold forbidden message occurs, it just causes the forbidden scope to be exited, causing no error.

3 Sequence Diagrams in UML 2.0

A major change in UML 2.0 concerns sequence diagrams that have been extended to include a number of features borrowed by MSCs and, to a limited extent, LSCs. As a consequence, UML sequence diagrams are now more expressive. In this section, we first give an overview of sequence diagrams in UML 2.0, and thereafter comment on how they deal with aspects like liveness and synchronisation which we have described for LSCs in Section 2.

Graphically, a sequence diagram has two dimensions: an horizontal dimension representing the instances participating in the scenario, and a vertical dimension representing time. Objects have a vertical dashed line called *lifeline*. The lifeline represents the existence of the instance at a particular time. The order of events along a lifeline is significant denoting the order in which these events should occur.

A *message* is a communication between two instances that conveys information with the expectation that action will ensue. A message will cause an operation to be invoked, a signal to be raised, or an instance to be created or destroyed. Messages are shown as a horizontal arrow from the lifeline of one instance to the lifeline of another instance. A message specifies not only the kind of communication between instances, but also the sender and receiver event occurrences associated to it.

A new feature of UML 2.0 sequence diagrams is that these may contain subinteractions called *interaction fragments* which can be structured and combined using *interaction operators*. The semantics of the resulting combined fragment depends upon the operator and is described informally in the UML 2.0 superstructure specification [14]. Below we give the semantics defined in [14] for some operators that we use in this paper:

alt designates that the combined fragment represents a choice of behaviour. At most one of the operands will execute. The operand that executes must have a guard expression that evaluates to true at this point in the interaction.

par designates that the combined fragment represents a parallel merge between the behaviours of the operands. The event occurrences of the different operands can be interleaved in any way as long as the ordering imposed by each operand as such is preserved.

seq designates that the combined fragment represents a weak sequencing between the behaviours of the operands, i.e. the ordering of event occurrences within each of the operands are maintained whereas event occurrences on different lifelines from different operands may come in any order, and event occurrences on the same lifeline from different operands are ordered such that an event occurrence of the first operand comes before that of the second operand.

strict designates that the combined fragment represents a strict sequencing between the behaviours of the operands. Notationally this means that the vertical coordinate of the contained fragments is significant throughout the whole scope of the combined fragment and not only on one lifeline.

neg designates that the combined fragment represents traces that are defined to be invalid. All interaction fragments that are different from negative are considered positive meaning that they describe traces that are valid and should be possible.

assert designates that the combined fragment represents an assertion. The sequences of the operand of the assertion are the only valid continuations.

It should be noticed that the above description taken literally from [14] is incomplete and ambiguous. For example, it is not clear whether it should be possible to have several guard expressions in an **alt** operator evaluating to true, and in such a case how to determine or enforce a particular operand to be executed. Further, using a **neg** operator it is not clear when a trace becomes invalid (negative), though one could expect that a prefix of the negative trace is still valid and therefore positive.

Comparatively to LSCs, it should be noted that some of the operators available for sequence diagrams are not possible for LSCs including, from the above list, the operator **strict**. Furthermore, it is possible to have interaction fragments combined in various ways and have several **assert** fragments in the same diagram. This is not possible for LSCs either, where an **assert** may only appear at the main chart level of an universal chart (see the discussion below on universal interactions). Other operands not possible in LSCs are discussed at the end of the paper.

Existential Interactions

As already mentioned, sequence diagrams can only express the possibility that a certain scenario occurs. In fact, sequence diagrams model behaviour in form of *possible* interactions, i.e. communication patterns that *can* occur between a set of instances. Therefore, in this sense sequence diagrams are similar to the existential charts in LSCs described in the previous section.

The semantics of an interaction is given as a pair of set of traces, i.e. the set of valid traces and the set of invalid traces. A trace is a sequence of event occurrences denoted $< e_1, e_2, \ldots, e_n >$. An event occurrence will also include information about the values of all relevant objects at this point in time.

Universal Interactions

UML sequence diagrams, in their current setting, seem to be able to express *necessity* only to a very limited extent. In particular, it is not clear whether it is possible to model universal charts as defined in LSCs through the new operator called **assert**. The superstructure specification is ambiguous in the definition of this operator, and it is not obvious from the text whether this operator enforces a sequence of messages to happen, or they are "expected" to happen (see [14], pages 412, 442). However, even if the former case were the intended one, it would still only solve the problem of expressing necessity in sequence diagrams at the interaction level, but not at the local level (lifeline/message).

The important dichotomy between *must* and *may* is discussed in detail in the next section, where we show how to achieve universality, both at the global and the local level, using an extension of OCL for liveness.

Synchronisation

As discussed in Section 2, synchronisation for LSCs can be achieved upon entering/exiting a subchart and when evaluating a condition or an assignment. Unfortunately, none of the above seems to be currently enforced in sequence diagrams. Consequently, it is not possible to impose that a certain condition must hold or an object's attribute must be assigned to a specific value before some specified instances progress in the interaction. Moreover, sequence diagrams allow instances participating in the same interaction fragment to enter and exit it independently.

The following are conditions provided in UML 2.0 sequence diagrams:

(i) An *interaction constraint* is a boolean expression shown in square brackets covering the lifeline where the first event will occur and positioned above the event inside an interaction operand.

(ii) A *state invariant* is a constraint on the state of a lifeline, whereby 'state' includes the attribute values of the lifeline. The constraint is evaluated during runtime immediately prior to the execution of the next event occurrence. If the constraint is true the trace is valid otherwise it is invalid. Notationally, state invariants are shown as a constraint inside a state symbol or in curly brackets and placed on a lifeline.

The problem of synchronisation for interaction constraints can be reduced to the problem of synchronisation for interaction fragments, that is, instances are synchronised at the beginning of an interaction and are thus not allowed to progress independently.

Concerning state invariants, it would be possible to ensure synchronisation among several instances by extending the constraint scope to more than one instance. This is not directly possible in UML, though an alternative solution is to have a state invariant at the same level within a **strict** fragment.

Sequence diagrams can include definitions of local attributes. These attribute definitions may appear near the top of the diagram frame or within note symbols at other places in the diagram. However, no mention is made of a specific notation for assignments.

Attribute assignments can be attached to return messages [14]. This ensures synchronisation between the two instances involved in the interaction, whereas there is no way of synchronising any other instances upon the assignment on a return message.

In order to avoid inconsistencies within interaction fragments, we need to enforce synchronisation at their beginning and end lines. In Section 4 we give a solution to this problem using OCL.

Forbidden Elements

In the previous versions of UML, it was not possible to express that, at any time, a specific scenario should not occur. However, as mentioned at the beginning of this section, a new operator called **neg** has been introduced for this purpose in UML 2.0. Therefore, to model what is called an anti-scenario in LSCs, we simply place it inside an interaction fragment within the scope of a **neg**.

Things get slightly more complicated if we want to model forbidden messages. To express that a specific message is not allowed to be sent at any point within an interaction fragment, we can define a parallel combined fragment where one operand contains the undesired message and the other contains the scope of the forbidden message.

Notice that a trace can be invalidated also using state invariants. The only limitation of state invariants with respect to a **neg** interaction fragment is that the scope of a state invariant is restricted to the lifeline that contains it.

4 Liveness in Sequence Diagrams of UML 2.0

The fact that the distinction between may and must elements and behaviour is not addressed properly in sequence diagrams leads to not being able to express *liveness/progress* properties. Liveness constraints denote that eventually something must happen. Moreover, in the context of sequence diagrams the "something" can be a message being received, a location [3] in the lifeline of an instance being reached (this means that an instance is forced to progress along its lifeline), or it may concern several messages that must be sent either in parallel or sequentially.

For example, we may want to say that

1 instance i sends a message *msg* to instance j and the message *must* be received.

2 at a certain location l_1 in a sequence diagram the corresponding instance i *must* progress and reach another location l_2. Reaching this latter location forces something to happen. It can be a message that is received, a message that is sent, a condition that is evaluated, an interaction fragment that is entered, and so on.

3 a particular sequence of messages (or a non-ordered set of messages) *must* happen, that is, they reflect a mandatory part of a scenario.

4 after a particular sequence of messages (or a non-ordered set of messages) *eventually* something else *must* happen.

As we have seen before, it is easy to do some of these in LSCs using hot elements: hot messages (case 1) and hot locations (case 2). Additionally, universal charts can be used to model situations as case 3 and case 4, though it should be noted that this is only possible at the main chart level. For sequence diagrams in UML 2.0 we may wish to have such cases at arbitrary levels within a diagram.

Hot messages and locations are not present in UML 2.0 sequence diagrams and therefore the cases above cannot be expressed directly. Notice that without being able to express liveness, synthesis of sequence diagrams is possible but not an issue, because it does not lead to very useful systems. We will come back to such considerations at the end of the paper.

Rather than adding more notation to sequence diagrams, what we propose

[3] Notice, that we use the word *location* here the same way it is used in LSCs. In sequence diagrams, the only locations considered correspond to event occurrences denoting either a message being sent or a message being received. To be able to refer to other "points" in a sequence diagram we need an extended notion of locations as offered in LSCs. These are important to address different kinds of liveness constraints.

here is to use OCL as a further means to express liveness, progress and necessary behaviour. By default a sequence diagram only describes possible traces and/or invalid traces. Further, if we want (part of) the behaviour described in a sequence diagram to become mandatory then we add particular OCL constraints to the given sequence diagram to indicate that.

Consider the sequence diagram in Fig. 2. Notice that it is not our intention to describe a realistic scenario with this diagram, but to illustrate allowed notation in sequence diagrams.

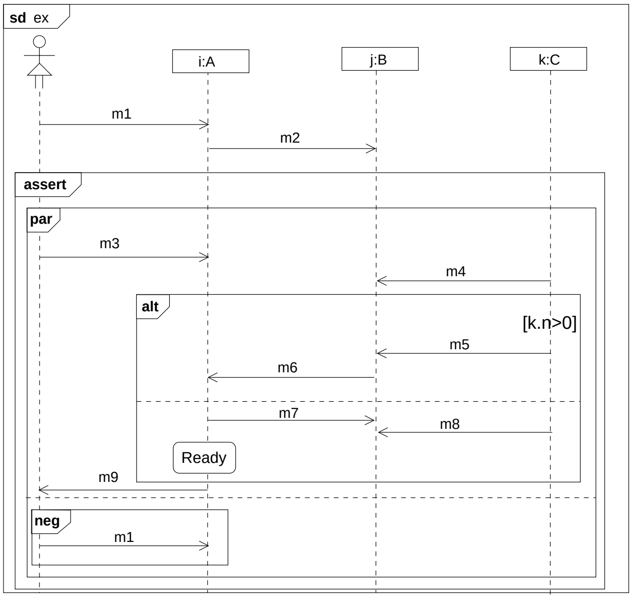

Fig. 2. An example of a sequence diagram in UML 2.0.

The diagram shows an interaction between a user and three system instances. The interaction is triggered by the user sending message `m1` to instance `i`. The intended meaning of the diagram is that after `m1` and `m2` have been sent the system must continue as described in the **assert** fragment. Whilst in the **assert** fragment, the user is not allowed to resend message `m1` at any time (this is indicated by the **neg** fragment containing message `m1` within a **par** fragment, whereby the intended execution is described in the first operand. Semantically it means that `m1` is not allowed to occur interleaved with whatever happens in the first operand). After messages `m3` and `m4` are sent[4] an **alt** fragment is entered, where the condition is given by `k.n>0`

[4] Notice that these messages are not ordered. However, we could, if intended, use a **strict** fragment to impose that `m3` has to happen before `m4`.

(where `n` is for instance an attribute of instance `k`). In the second operand of **alt**, `Ready` is a state invariant over instance `i`. Recall that if a state invariant is evaluated to false the whole trace is invalid. The valid execution finishes with instance `i` sending message `m9` to the user.

The next five conditions, which we may want to consider for the example scenario, are not immediately enforced by the informal semantics of sequence diagrams in [14] (see our discussion in Section 3 concerning universal interactions and synchronisation).

(a) Message `m2` if sent must be received (liveness case 1).

(b) After user `u` sends message `m3`, she will eventually receive message `m9` (liveness case 2).

(c) After **assert** starts it will eventually finish (for one instance it denotes a liveness case 2). Since all instances that are involved in the fragment will have to obey to this constraint, it can be understood as defining **assert** as a mandatory interaction fragment (liveness case 3).

(d) Both `m1` and `m2` have to be sent before an instance can enter the **assert** interaction fragment. After `m1` and `m2` have been sent, eventually the **assert** interaction fragment must be entered (liveness case 4).

(e) Message `m8` has been received before instance `i` enters state `Ready` (liveness case 4).

In order to impose these conditions on the diagram we have to add further constraints that say this explicitly. We use OCL for this.

We have enriched OCL in previous work to express simple liveness constraints [2]. Essentially, we have introduced an **after eventually** OCL template which can be used to express that **after** a condition becomes true there is a guarantee that **eventually** another condition will become true. In [2] the interpretation of the "eventually" is left open: it can be immediately or sometime in the future. The template is written as follows.

```
context Classifier
   after:  oclExpression
   eventually:  oclExpression
```

Like an invariant or a pre/post-condition pair, an after-eventually (AE) template is written in the context of a type, typically a classifier such as a class or a component from a UML model. As there, "self" may be used to refer to the instance of this type to which the constraint is being applied. Alternatively, the contextual instance can be declared as a variable writing for example `a:Classifier`.

This template is essentially all we need to be able to impose liveness constraints on a sequence diagram. Further, we use message expressions that have been added in OCL 2.0 [13] (they used to be called action expressions in previous work). This makes it possible to distinguish between messages being sent or received. For example, for a contextual instance `i`, the expression `i.j^m2()` denotes that instance `i` has sent a message `m2` to instance `j`, whilst `j.m2()` is used to express that instance `j` receives message `m2`.

To illustrate how to apply the AE template in the liveness case 1 consider the following OCL constraint

context `i:A`
    let `j:B` **in**
    after: `i.j^m2()`
    eventually: `j.m2()`

This constraint states that if instance `i` has sent message `m2` to instance `j` (the **after** clause is true) then we know that **eventually** the message must be received by instance `j`. Again, because of the subtlety of the "eventually" we do not care here whether the message is being sent synchronously or asynchronously. This information is contained in the sequence diagram and noticeable by the arrowhead used (in this case according to Fig. 2 the communication is asynchronous). The semantics given to the "eventually" of an OCL constraint will therefore only make sense with the corresponding sequence diagram.

It should be noted that message expressions in OCL 2.0 can only be used in a postcondition of an operation/method (say `o`) and have the underlying meaning that the message must have been sent during the execution of `o`. Our usage within an AE template is not incompatible with this intention because messages are always sent between instances while executing certain operations. Further, the OCL 2.0 specification document *does not* include a formal semantics for message expressions. A recent paper describes an extension of the usual semantics given to OCL expressions for message expressions [6]. Our approach is in accordance with the semantics of [6].

Because OCL constraints are written at a classifier level we need to indicate the class to which instance `i` belongs (in this case class `A`). Also, in OCL we can only refer to instances other than the contextual one, if we are able to *navigate* to them. Being able to navigate from one instance to another means that there is an association between the corresponding classes in the class model and consequently a link between the instances. This is naturally a problem, if we are drawing a sequence diagram and want to add some liveness constraints before we have built the class diagram (or even a clear idea about the multiplicities of the association). To address this problem, we assume that instances in a sequence diagram that communicate directly with each other must have a link between them. In the OCL constraint, we indicate the type of the instances in the **let** clause.

For liveness constraints of case 2 we mainly need to refer to different locations in the lifeline of one instance, which is the contextual instance of the constraint. Several situations are possible depending on what either of the locations are: message send event, message receive event, condition, assignment, state invariant, entering or exiting an interaction fragment. Notice that state invariants are also considered interaction fragments (see [14] page 433) but they should be treated differently as they only concern one instance.

To avoid informal and ambiguous conditions written possibly in natural language, we assume the use of OCL for conditions instead. In sequence diagrams assignments can be given to set parameter values, indicate return values of messages, and setting a time variable (e.g., `t=now`). Again we assume the usage of OCL for consistency. The following constraint describes condition (b) of liveness case 2.

```
context u:User
   let: i:A in
   after: u.i^m3()
   eventually: u.m9()
```

Notice, that in all examples given above, the OCL constraints are general in the sense that they describe constraints which are independent of particular locations in a sequence diagram. For example, if a constraint states that after a message is sent it will be received at some point, this applies to all locations in the sequence diagram where the same message is sent. To make further distinctions, conditions can be added in the **after** or **eventually** clauses.

We have not yet addressed the cases where a location corresponds to the beginning or the end of an interaction fragment. In order to be able to do so using OCL, we introduce the following two properties that apply to all objects

```
oclAtBegInteractFrag(op:InteractionOperator):Boolean
oclAtEndInteractFrag(op:InteractionOperator):Boolean
```

The operation `i.oclAtBegInteractFrag(x)` returns true if instance `i` is at a location which is the beginning of an interaction fragment of kind `x`. To be able to refer to a particular operand within an interaction fragments we can use the notation `par#1` for the first, `par#2` for the second, and so on. For nested fragments we can use the double colon '::'(a usual notation for indicating paths). Notice that there may be several locations in a diagram where `oclAtBegInteractFrag(x)` or `oclAtEndInteractFrag(x)` are true (the same kind of fragment is repeated in the diagram) and this is a perfectly reasonable situation. Indeed, the constraints we want to write (see below) must apply to any such locations.

Consequently, we can now use the above operations in our OCL liveness template to express condition (c) of liveness case 2 for one of the instances that will participate in the interaction described by the fragment.

```
context i:A
    after:  i.oclAtBegInteractFrag(assert)
    eventually:  i.oclAtEndInteractFrag(assert)
```

The constraint states that after instance `i` enters the interaction fragment `assert` it is forced to progress and eventually reach the location which corresponds to the end of the (same) interaction fragment `assert`. Notice that even if we define the same constraint for the remaining instances it still does not mean that the instances synchronise at the beginning and at the end of the interaction fragment. To indicate this we have to define a global constraint at the sequence diagram level. We will come back to this shortly.

In order to deal with liveness cases 3 and 4, we need to be able to write OCL constraints at the global level of a sequence diagram or an interaction fragment. We want to express that a collection of messages (sequentially or in parallel) have been sent and/or received, and naturally this involves several instances. This means that the constraint can no longer be given from the perspective of one of the instances participating in the interaction. Consequently, the context shifts from one instance to a global level of several instances.

We consequently allow the context to be given by

```
context  sd name::interactfragment1::...::interactfragmentN
```

where `sd` denotes a sequence diagram and `name` is the name of the diagram. The context can be the whole diagram or a specific part, and we can therefore denote a particular interaction fragment in the diagram. This is indicated using the path notation '::' to navigate from top level of the entire diagram to the target interaction fragment.

Condition (d) basically states that the messages `m1` and `m2` constitute a precondition for the **assert** fragment. Both messages have to be sent before any instance is allowed to progress and reach the assert fragment or any location thereafter. We can add here the constraint that the instances have to synchronise when entering the **assert** fragment. Condition (d) is described by the following constraint

```
context sd ex
  let:  u:User, i:A, j:B, k:C  in
  after:  u.i^m1()  and i.j^m2()
  eventually:  u.oclAtBegInteractFrag(assert)  and
    i.oclAtBegInteractFrag(assert) and j.oclAtBegInteractFrag(assert)
     and k.oclAtBegInteractFrag(assert)
```

Further, we can now describe also that an **assert** fragment must finish, i.e., after all instances are at the beginning of the interaction fragment, eventually all instances must reach the end of the (same) fragment and synchronise. This refers to condition (c).

```
context sd ex
  let:  u:User, i:A, j:B, k:C  in
  after:  u.oclAtBegInteractFrag(assert)  and
    i.oclAtBegInteractFrag(assert) and j.oclAtBegInteractFrag(assert)
     and k.oclAtBegInteractFrag(assert)
  eventually:    u.oclAtEndInteractFrag(assert)  and
    i.oclAtEndInteractFrag(assert) and j.oclAtEndInteractFrag(assert)
     and k.oclAtEndInteractFrag(assert)
```

Finally, in case (e) we are imposing that `m8` has to be received before instance `i` enters state `Ready`. The constraint is global, because message `m8` is exchanged between instances `k` and `j` whilst the state invariant refers to instance `i`. Without this constraint, instance `i` is allowed to enter state `Ready` before `m8` was sent and received. There are two possible ways to deal with this case. Either we use OCL as described below or we enclose the messages and the state invariant within a **strict** combined fragment.

In OCL we can express states using the predefined operation `oclInState()`. For example `i.oclInState(Ready)` can be used as a condition on the state of instance `i`. Other possible state invariants in sequence diagrams are written within curly brackets, for example {`i.p==15`} which are similarly written in OCL as `i.p=15`. The next global constraint (not an AE template) describes case (e).

```
context sd ex::assert::par#1::alt#2  inv:
  let:  i:A, j:B, k:C  in
  if not(j.m8()) then  not(i.oclInState(Ready))
```

Notice that the constraint concerns the second of the **alt** operands only (indicated by the contextual path) though this restriction would not be required for this particular sequence diagram since the message and state invariant only appear in that operand.

An additional (local) OCL constraint is need to enforce that instance `i` indeed reaches state `Ready` and corresponds to the following liveness constraint of case 2.

```
context i:A
```

after: `i.j^m7()`
eventually: `i.oclInState(Ready)`

5 Conclusions

In this paper, we give an overview of some of the most relevant and newly added features of sequence diagrams in UML 2.0. Even though sequence diagrams have been influenced by LSCs, they can still not adequately distinguish between must and may elements, necessary and possible behaviour. We have seen how this can be addressed by simply enriching a sequence diagram with constraints for liveness. The constraints are given in a simple temporal extension of UML's constraint language OCL. The extension consists of a liveness template **after eventually** which can be used *locally* over a message or a lifeline, and *globally* over an interaction. For example, locally it can be used to ensure that if sent a message must be received, or to indicate that an instance must progress between locations on its lifeline. Further, when used at the global level it enforces an interaction to occur.

We have described some of the problems of UML 2.0's specification concerning sequence diagrams. Naturally, a formal semantics is essential to solve such issues. In this paper we have not focused on a formal semantics but on how (assuming defined a semantics for sequence diagrams in UML 2.0) we can make this language richer for expressing inter-object behaviour. Liveness properties are normally only expressed at the level of state diagrams, but being able to express liveness at a higher level of abstraction is particularly important because it makes automatic generation of intra-object behaviour from inter-object specifications more realistic even though more complex. For example, the algorithm for translating MSCs to statecharts given in [12] is much simpler than a corresponding algorithm for LSCs as pointed out in [8]. A further aspect of interest which is not considered by these approaches (and others in the literature) is that a collection of MSCs or LSCs may actually "imply" further unspecified scenarios which are undesired. An interesting framework addressing such issues for the most basic form of MSCs is given in [1] and should be explored for more expressive languages including our extension of sequence diagrams.

Our liveness extension for sequence diagrams corresponds to a feasible *soft extension* of UML. That is, we do not add new language constructs to UML and instead we use OCL liveness constraints to add expressiveness to sequence diagrams. By contrast, in a recent paper [10] a new interaction operator called **xalt** is introduced to capture one particular kind of mandatory behaviour, namely, *mandatory alternatives.* This operator is illustrated with interaction overview diagrams (basically a specific variant of activity diagrams for capturing the flow of interactions at a higher level) but is probably (though not explicitly mentioned) also intended for sequence diagrams. The semantics of the new operator is, however, never explained formally. Further, the trace semantics for other interaction operators is only given using natural language. Other recent work describing the trace semantics of UML 2.0 sequence diagrams is given in [15].

It should be noted that the usage of OCL in combination with sequence diagrams has further advantages, namely the fact that we have a language for navigation expressions and describing object models. This is recognised as an important issue that needs exploring in the context of LSCs and the Play Engine (see chapter 21.2 of [9]).

The importance of liveness constraints for requirements languages is also recognised for MSCs. In this context [11] introduces a new composition operator called "trigger composition" to extend the expressiveness of MSCs for liveness/progress properties. This operator is equivalent to our OCL liveness template, consequently MSCs with trigger composition can be modelled with our liveness-enriched sequence diagrams.

In a companion paper [4] we have defined a formal operational semantics for UML 2.0's sequence diagrams and our liveness extension. The semantics is based on Abstract State Machines (ASMs) [7]. We believe that the choice of ASMs is advantageous becase they are a state-based model and consequently are likely to facilitate synthesis. The semantics of the used liveness template has been given in a previous paper [2] though in that paper the template was only used locally. In this paper, we have used OCL 2.0 message expressions within our liveness template whereby it should be noted that it is done in accordance to the semantics given to message expressions in [6].

Unfortunately, many concepts introduced in [14] are unclear, ambiguous or inconsistent. One example concerns the intended reaction of the system to violations, i.e. invalid traces produced by an interaction fragment (**neg**) or by a single lifeline (state invariants that do not hold). It is unclear whether the intention is that the system aborts (LSCs solution in case of "hot violations"), or whether the current interaction fragment is exited (LSCs solution in case of "cold violations").

We have mentioned the problems that can occur if instances are not forced to synchronise at the beginning and the end of interaction fragments, and at interaction constraints. In particular, in the case of **alt** combined fragments, although never mentioned in the specification, it appears that synchronisation *should* be guaranteed among participating instances which should block until the constraint is evaluated and the operand to be executed decided.

Moreover, we have seen that to specify in LSCs that a certain sequence of communications is forbidden it is enough to consider a universal chart such that the prechart corresponds to the undesirable behaviour and the main body contains a hot false condition. By contrast, there are two (apparent) ways to describe this in sequence diagrams. On the one side, sequence diagrams offer an interaction operator called **neg** for describing forbidden behaviour, that is, whatever interactions are described within this operand, they denote an invalid trace. We have shown that it can also be used for modelling forbidden conditions and messages. On the other side and similarly to what is done in LSCs, we can use false *state invariants* following a sequence of interactions. According to UML, if a state invariant is false when evaluated, then the previous trace is invalid. Notice, however, that there is a problem with this option. State invariants only belong to one instance in the diagram, and consequently we are not synchronising all the instances participating in the interaction at the time that the constraint is evaluated. This may lead to a situation where the instance with the invariant progresses earlier and reaches the constraint before the other instances have done their activities. This is clearly undesirable, but it is not immediate in sequence diagrams that situations like this are or have to be prevented.

In addition to the interaction operators considered in this paper, sequence diagrams in UML 2.0 have operators called **ignore** and **consider**. Intuitively, the first operator can be used to model interactions *ignoring* that other particular messages are allowed to occur in parallel, whilst the second operator can be used to model an interaction where only certain messages are *considered* significant (which is equivalent to saying that all other messages are ignored). Notice that this more abstract way of modelling interactions is not possible using LSCs but can be particularly useful for testing purposes.

Finally, sequence diagrams in UML 2.0 (through the use of interaction operators like **ignore** and **consider**) and our *liveness-enriched* sequence diagrams have advantages for automatic test generation. This means that we are able to specify scenarios that the system must exhibit, avoid, and so on. Future work includes the extension of our language for test directives in [3] with respect to liveness-enriched sequence diagrams.

Acknowledgement

The second author was supported by the EPSRC grant GR/R16891. We thank the anonymous referees for detailed comments on a previous version of this paper.

References

[1] R. Alur, K. Etessami, and M. Yannakakis. Inference of message sequence charts. *IEEE Transactions on Software Engineering*, 29(7):623–633, July 2003.

[2] J. Bradfield, J. Küster-Filipe, and P. Stevens. Enriching OCL using observational mu-calculus. In R.-D. Kutsche and H. Weber, editors, *Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering (FASE), Grenoble, France, April 2002*, volume 2306 of *LNCS*, pages 203–217. Springer, 2002.

[3] A. Cavarra, C. Crichton, and J. Davies. A method for the automatic generation of test suites from object models. *Information and Software Technology*, 46(5):309–314, April 2004.

[4] A. Cavarra and J. Küster-Filipe. Formalizing liveness enriched sequence diagrams using ASMs. In *Proceedings of the 11th International Workshop on Abstract State Machines (ASM 2004), Halle (Saale), Germany, 24-28 May 2004*, LNCS. Springer, 2004. To appear.

[5] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[6] S. Flake and W. Müller. Formal semantics of OCL messages. In *Proceedings of the UML'2003 Workshop on OCL 2.0 - Industry standard or scientific playground?, San Francisco, USA, 21st October 2003*, ENTCS. Elsevier, 2004. To appear.

[7] Y. Gurevich. Specification and validation methods. In E. Börger, editor, *Evolving Algebras 1993: Lipari Guide*, pages 9–36. Oxford University Press, 1995.

[8] D. Harel and H. Kugler. Synthesizing state-based object systems from LSC specifications. *International Journal of Foundations of Computer Science*, 13(1):5–51, 2002.

[9] D. Harel and R. Marelly. *Come, Let's Play: Scenario-based Programming Using LSCs and the Play-Engine*. Springer, 2003.

[10] Ø. Haugen and K. Stølen. STAIRS - Steps to analyze interactions with refinement semantics. In P. Stevens, J. Whittle, and G. Booch, editors, *Proceedings of UML 2003*, volume 2863 of *LNCS*, pages 388–402. Springer, 2003.

[11] I. Krüger. Capturing overlapping, triggered, and preemptive collaborations using MSCs. In M. Pezzè, editor, *Proceedings of the 6th International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 2621 of *LNCS*, pages 387–402. Springer, 2003.

[12] I. Krüger, R. Grosu, P. Scholz, and M. Broy. From MSCs to Statecharts. In F. Rammig, editor, *Proceedings of Distributed and Parallel Embedded Systems (DIPES'98)*. Kluwer Academic Publishers, 1999.

[13] OMG. *UML 2.0 OCL Specification, Version 1.6.* OMG document ad/03-01-07, available from www.uml.org, August 2003.

[14] OMG. *UML 2.0 Superstructure Draft Adopted Specification.* OMG document ptc/03-08-02, available from www.uml.org, August 2003.

[15] H. Störrle. Semantics of interactions in UML 2.0. In *Proceedings of IEEE Symposium on Human Centric Computing Languages and Environments, Auckland, New Zealand, 28-31 October 2003*. IEEE Computer Society Press, 2003.

[16] ITU-TS Recommendation Z.120. *Message Sequence Chart (MSC)*. ITU-TS, Geneva, 1996.