Thread–based Analysis of Sequence Diagrams

Haitao Dan, Robert M. Hierons and Steve Counsell

School of Information Systems, Computing & Mathematics,
Brunel University,
Uxbridge, Middlesex UB8 3PH, UK
{hai.dan, rob.hierons, steve.counsell}@brunel.ac.uk

Abstract. Sequence Diagrams (SDs) offer an intuitive and visual way of describing expected behaviour of Object Oriented (OO) software. They focus on modelling the method calls among participants of a software system at runtime. This is an essential difference from its ancestor, basic Message Sequence Charts (bMSCs), which are mainly used to model the exchange of asynchronous messages. Since method calls are regarded as synchronous messages in the Unified Modelling Language (UML) Version 2.0, synchronous messages play a significantly more important role in SDs than in bMSCs. However, the effect of this difference has not been fully explored in previous work on the semantics of SDs. One important aim of this paper is to identify the differences between SDs and bMSCs. We observe that using traditional semantics to interpret SDs may not interpret SDs correct under certain circumstances. Consequently, we propose a new method to interpret SDs which uses thread tags to deal with identified problems.

Keywords: Sequence Diagram, Semantics, Partial Orders, Concurrency, Object Oriented, Thread tags.

1 Introduction

In the Unified Modelling Language (UML) Version 2.0, a Sequence Diagram (SD) is a type of Interaction Diagram (ID), as are Communication Diagrams, Interaction Overview Diagrams and Timing Diagrams [OMG05]. Although an SD is a second-level modelling language in UML 2.0, it is the most commonly used type of notation in ID and is regarded as the most popular UML behaviour modelling language.

In Object Oriented (OO) software, SD-based specifications are usually used to capture system requirements, model function logic or as automatic test models. An SD is a versatile tool that can be used in many parts of the OO software development process; its ancestor, the basic Message Sequence Chart (bMSC), developed in the early 1990s, was designed for modelling communication systems. As a consequence of the difference between application domains, minor changes were introduced into the first version of UML.

Due to their similarity, the semantics developed for bMSC have been naturally inherited by SD. In particular, those based on partial order theory have

been widely adopted in both research and industry, because they are conceptually straightforward when compared with counterparts such as process algebra based semantics [MR94]. Henceforth, we use the term 'traditional semantics' to refer to partial order based semantics.

Although the syntax of SDs and bMSCs are almost identical, we argue that small differences between them may cause significant semantic variations between the two. That is, traditional semantics may not interpret SDs correctly. More specifically, SDs are often used to model OO software systems in which communication is synchronous; bMSCs are normally used in asynchronous message based communication systems. To model communication systems, bMSCs assume that all participants are running concurrently and the messages between them are always asynchronous. On the other hand, the messages between lifelines of SDs are likely to be synchronous and there is no longer a one-to-one correspondence between lifelines and threads of control. The differences described above imply that when traditional semantics are applied on some SDs, it can result in unintentional semantics. To solve this problem, we propose a new method for interpreting sequence diagrams.

To find a proper method of interpreting SDs, we first attempt to solve the problem using only the meta-classes from UML 2.0. We argue that existing UML meta-classes cannot be used because, unlike bMSCs, lifelines are generally orthogonal to threads. This implies that a thread may involve multiple lifelines and a lifeline may involve multiple threads. We thus introduce *thread tags* into SDs and provide an informal semantics for interpreting SDs.

In order to simplify the inference process, we only consider the most important parts of SDs and bMSCs related to our proposed semantics. We assume that a complete semantics based on our work can then be induced.

1.1 Related Work

When discussing the semantics of SDs, it is worth reviewing previous research into bMSCs. Mauw and Reniers [MR94] used a process algebra to interpret the semantics of bMSC and this approach has been adopted as the standard semantics for bMSC [IT98]; Grabowski et al. [GGR93] proposed petri-net based semantics for bMSC; Ladkin and Leue [LL93] used Büchi Automata to capture the meaning of bMSC; Jonsson and Padilla [JP01] used Abstract Execution Machines to describe bMSC semantics and at the same time, considered inline expressions and data in bMSCs; Alur et al. [AHP96] were the first to use labelled partially-ordered structures to formalize bMSCs.

The increased popularity of the UML has led to the semantics of SDs receiving more attention. In UML 2.0, SDs were significantly revised to allow adequate modelling of complex software system based on the new version of bMSC [OMG05]. Although UML 2.0 tried to provide semantics for every modelling language using a meta-model [Sel04], SDs have only been assigned a behaviour informal semantics according to traditional bMSC semantics. In [Sto03,HHRS05,CK04], formal trace based semantics for SDs were provided and [LS06] solved the semantic problem using an automata-theoretic approach. In [GS05], safety and liveness

properties were used for distinguishing valid behaviours from invalid. Finally, Harel and Maoz [HM06] proposed Modal UML Sequence Diagrams (MUSD), an extension of SDs based on the approach used in Live Sequence Charts (LSCs) to extend bMSCs [DH01]. These newly developed SD semantics were based on different kinds of bMSC semantics. The bMSC semantics were revised to conform to the intended semantics of UML 2.0 with added semantics for the new meta-classes of UML 2.0 (eg., for CombinedFramgment and InteractionOperator).

Previous work has been largely based on bMSC semantics, the core ideas of which are commonly derived from corresponding bMSC semantics directly. Although the notation used to describe SDs and bMSCs are almost identical, the differences between them do affect how the diagrams are interpreted. For instance, a lifeline in UML 2.0 no longer represents a process. If we still interpret it as bMSC's instance, the correct concurrency information will not be deduced from an SD in certain circumstances. This observation is the motivation for the work described in this paper.

An interesting variation on mainstream bMSCs are LSCs which use a two-layer approach to distinguish mandatory and provisional behaviour in scenarios. Similar to our approach, the semantics of LSCs consider the problem caused by synchronous method calls. It assumes that a synchronous message is received before the next event on the instance which sent the message. This simple solution addressing the additional orders induced by synchronous messages is sufficient for LSCs, but it may be problematic when applied to SDs, because lifelines of SDs no longer contain a thread of control.

The remainder of this paper is structured as follows. A comparison between the SD and bMSC standards is presented and the traditional semantics are introduced informally in Section 2. In Section 3, some possible problems that can arise when using the traditional semantics to interpret SD are described. In Section 4, two unsuccessful solutions that use UML 2.0 meta-classes are analysed. We argue that there are no meta-classes in UML 2.0 that can achieve correct semantics for SDs. After the analysis, inference rules for interpreting SD based on thread tags are proposed in Section 5. Finally, in Section 6, our work is concluded and potential future directions described.

2 Preliminary

This paper is motivated by the observation that the existing semantics of SDs have problems when interpreting SDs with synchronous messages. To illustrate the problems, SD and bMSC standards are first compared to reveal their differences; second, the traditional semantics of SD are introduced based on bMSC partial order semantics.

2.1 The Difference between SD and bMSC

Both SD and bMSC are complicated modelling languages and both standards use meta-methods to define themselves as hierarchies of meta-classes. We com-

pare the two languages according to the selected constructs of *Lifeline* (*instance* in bMSC), *Message*, *MessageOccurrenceSpecification* (*event* in bMSC) and *ExecutionSpecification* (*method* and *suspension* in bMSC)[1].

In UML 2.0 meta-models, an SD is decomposed into *Lifelines* and *Messages*. A *Lifeline* commonly represents an instance of a class or component in an OO program and it contains different kinds of *OccurrenceSpecifications*. *OccurrenceSpecification* is an equivalent concept to *event* in bMSC. Two main kinds of *OccurrenceSpecification* are *MessageOccurrenceSpecification* and *ExecutionSpecification*. *MessageOccurrenceSpecifications* are used to represent sending or receiving of messages. *ExecutionSpecifications* are specifications of the execution of units of behaviours or actions within the *Lifeline* and always triggered by *Messages*.

The sturctures of *msc* are similar to the structures in an SD meta-model, although *msc* is defined using a meta-language. An *msc body* includes multiple *instances*. Each *instance* has its own thread of control, and each *instance* has a list of *events* which appear along it. *message event* and *method call event* are the two main types of *event*. A pair of *message events* or a pair of *method call events* are used to represent a message between two *instances*. A *method* is a named unit of behaviour inside an *instance*. A *suspension* occurs when a synchronous *method call* is sent and lasts until the reply of the call returns.

Although both modelling languages have similar core constructs and each construct has similar graphical presentations, three underlying differences need to be addressed.

The first difference is that *Lifeline* and *instance* generally represent different things. *instance* of bMSC usually represents a process, a network device or a system. *Lifelines* in SDs always represent objects or instances of a component. *instance* always has its own thread of control (thread)[2], but a *Lifeline* does not.

The second difference is that SD and bMSC's messages are categorised differently. In SD, there are three message types: *synchCall*, *asynchCall* and *asynchSignal*. Generally speaking, *synchCall* is the more commonly used type of message modelling synchronous method call between objects. For bMSCs, *message* only refers to asynchronous communication between two *instances* and is the most often used type of message in bMSC. In general, the difference is due to the fact that SDs are used to model communication between objects, while bMSCs are designed to model message exchange between processes.

The third difference is that SDs and bMSCs use different ways to model the activity of a *Lifeline* or an *instance*. This difference is also due to the fact that an *instance* has a thread but *Lifeline* does not. Since an *instance* has its thread of control, if there is a synchronous *method call* from an *instance*, the caller will enter a *suspension* region where no events occur until the reply of the call

[1] The emphasized words are definitions from the standard. A detailed explanation of them can be found in UML 2.0 and MSC standards [OMG05,IT98].

[2] Here, thread of control represents an abstract notion of control unlike *thread* or *process* in operation system (OS). More specifically, an independent task which is executed sequentially should be regarded as owning its own thread of control.

returns. However, SDs do not restrict a *Lifeline* to map to only one thread of control. It is possible for *ExecutionSpecifications* of multiple threads to overlap in one *Lifeline*. An example of this is given in *Example 9* (Figure 6).

2.2 Traditional Partial Order Semantics

In the UML 2.0, the sequences of *MessageOccurrenceSpecification* are regarded as the meanings of SDs. Thus, traditional semantics can be described by two ordering rules when only considering the meta-classes: *Lifeline*, *Message* and *MessageOccurrenceSpecification*.

1. *MessageOccurrenceSpecifications* that appear on the same *Lifeline* are ordered from top to bottom.
2. Sending *MessageOccurrenceSpecification* always occurs before the corresponding receiving *MessageOccurrenceSpecification*.

Based on the rules, an informal partial order semantics of SDs can be defined. It is the transitive closure of the union of the following two orders:

- the union of total orders of *MessageOccurrenceSpecifications* in each *Lifeline*;
- the ordering relations between the *MessageOccurrenceSpecification* pairs of sending and receiving of the same message;

3 The Effect of Changing from bMSC to SD

In Section 2, we introduced the differences between SD and bMSC and the traditional semantics for SD. In order to illustrate the effects of the changes, five simple SD examples are presented. *Example 1* in Figure 1 shows that synchronous messages convey the events[3] of one thread to multiple lifelines. *Examples 2* and *3* in Figure 2 demonstrate that it is not enough to simply apply traditional semantics for interpreting two kinds of SDs. *Examples 4* and *5* in Figure 3 show what can happen when multiple threads enter the same lifeline in one SD.

Example 1 is an example of an SD where a synchronous message is represented by a solid line with a filled arrowhead ($c1$ and $c2$); an open arrowhead is used to represent asynchronous messages ($m1$). The reply to a synchronous message is represented as a dashed line with an open arrowhead pointing back to the caller ($rc1$ and $rc2$). Each thin rectangle on a lifeline represents an *ExecutionSpecification* defined as "*a specification of the execution of a unit of behavior or action within the Lifeline*" and denotes that the lifeline is active. In an OO program, when the synchronous message is a call to a method of the object represented by the lifeline, the thin rectangle signifies that the method is on the stack.

According to the UML 2.0 standard, we can interpret *Example 1* as a running method of *a:A* calling the method $c1$ in object *b:B* and *b:B* sending an

[3] For the sake of simplicity, we use *event* to replace the lengthy *MessageOccurrenceSpecification*.

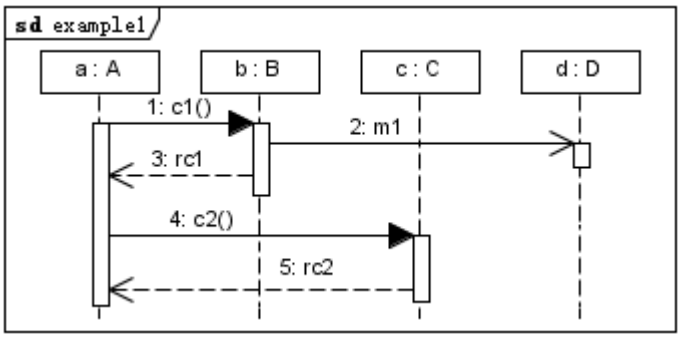

Fig. 1. An SD with synchronous messages

asynchronous message $m1$ to object *d:D*. Method $c1$ returns after $m1$ has been sent. Finally, the method in object *a:A* calls the $c2$ method in object *c:C* and $c2$ returns.

This scenario means that methods $c1$ and $c2$ are successively executed in one thread, so the events $!c1$, $?c1$, $!m1$, $!rc1$, $?rc2$, $!c2$, $?c2$, $!rc2$ and $?rc2$ all belong to one thread but are expanded to three lifelines. Here, the shriek symbol, !, represents sending and the ? symbol represents receiving (of a call or message).

Example 1 illustrates how synchronous messages expand events of one thread into different lifelines. As a consequence, a normal lifeline no longer represents a thread of control.

Applying traditional semantics to this example, the orders of the events are: $!c1 <?c1 <!m1 <!rc1 <?rc1 <!c2 <?c2 <!rc2 <?rc2$ and $!m1 <?m1$ which is equivalent to our intuitive understanding.

Now we assume that the orders in *Example 1* define the traces that we want to model and give two other examples (*Examples 2* and *3*) which try to model the same traces.

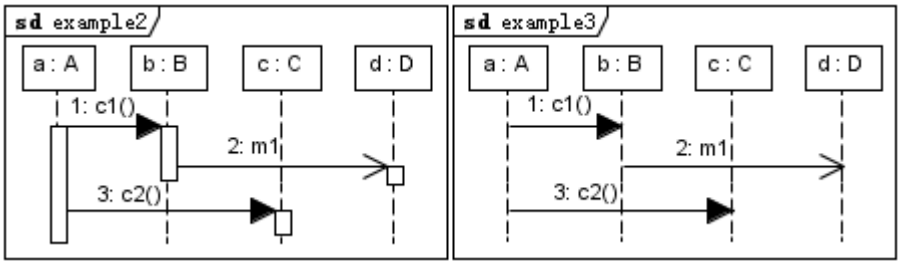

Fig. 2. SDs with *synchCalls*

Example 2 is also a common SD, but the returns of the synchronous calls are not included. Applying traditional semantics, we get following orders: $!c1 < ?c1 <!m1$, $!c1 <!c2 <?c2$ and $!m1 <?m1$. The relations $?c1 <!c2$ and $!m1 < !c2$ are missing. However, according to the meaning of execution specification and synchronous call, the two calls from the same execution specification ($!c1$ and $!c2$) are still in the same thread, so the missing orders should exist. The partial order should be $!c1 <?c1 <!m1 <!c2 <?c2$ and $!m1 <?m1$ which is the partial order of *Example 1* except with reply events removed. This example shows that traditional semantics of SD are not enough to interpret SDs if synchronous messages are included and replies of synchronous calls omitted.

In *Example 3*, a simplified SD is given. Since execution specification is optional in SD, software engineers may draw SDs as shown in *Example 3* to reflect the traces in *Example 1*. Here, it is not easy to induce the desired partial order from *Example 3*. Calls $c1$ and $c2$ may belong to two different threads, so the events of $c1$ and $c2$ may interleave. As a result, the intended orders may be $!c1 <?c1 <!m1 <?m1$ and $!c1 <!c2 <?c2$, the order produced by applying traditional semantics. Compared with the partial order of *Example 1*, it does not include relations like $?c1 <!c2$ and $!m1 <!c2$.

This example shows that users may draw a diagram based on their own assumption that all the calls are in one thread and are synchronized; the assumed orders can not subsequently be retrieved from the diagram when it is formally analyzed.

The first three examples explain how synchronous messages bring the events of one thread to multiple lifelines, and the problems that may result from this. In fact, in many cases it can also happen that multiple threads enter one lifeline in an SD.

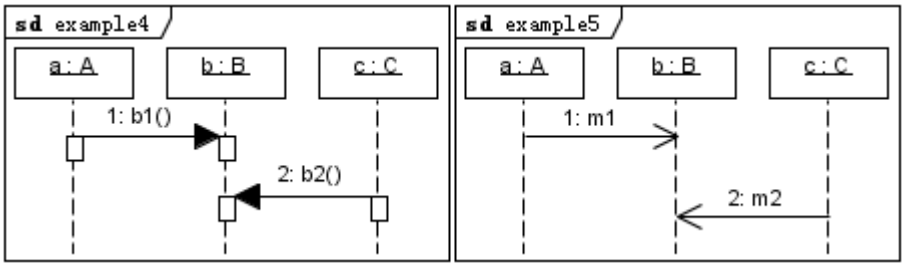

Fig. 3. SDs of multiple thread enter one lifeline

An intuitive interpretation of *Example 4* shown in Figure 3 is that methods $b1$ and $b2$ in object *b:B* are called by *a:A* and *b:B* sequentially from different threads. This example illustrates that sometimes it is impossible to determine whether the events on the same lifeline belong to the same thread. Another version of *Example 4* is shown in *Example 5* (same figure). It is a similar scenario to

Example 4 except that synchronous calls $b1$ and $b2$ are replaced by asynchronous messages $m1$ and $m2$. If *Example 5* is a bMSC, it induces a canonical race condition [AHP96,Mit05]. According to traditional semantics, $m1$, and $m2$ can be sent in either order. There is no way to enforce $m1$ arriving before $m2$ without additional information. If $?m1$ and $?m2$ belong to one thread and the system is implemented following *Example 5*, then a race condition may be introduced into the system. However, when checking this diagram in the context of OO software development, we can not decide whether a race condition applies since $?m1$ and $?m2$ might not belong to the same thread.

According to these examples, we find that the most problematic issue of interpreting an SD with synchronous messages is how to retain thread information in SDs when a lifeline does not correspond to a thread of control.

4 Mapping Events to Threads

To correctly interpret an SD, it is necessary to find a way of mapping different events to existing threads in the SD; we call this *Thread Mapping*.

To achieve this, two related meta-classes in UML 2.0 are selected. First is *execution specification* which can be used for grouping events. The second is *active object* which contains information regarding concurrency. The feasibility of using these meta-classes to do the thread mapping is now analysed.

4.1 Using Execution Specification

Example 2 in Figure 2 shows that using the information contained in execution specifications may help to handle the thread mapping problem. The events triggered by synchronous messages can be grouped together by analysing the connective relations of the execution specifications.

Thread mapping is relatively straightforward for simple diagrams like *Example 1* and *Example 2*. With execution specifications, we can group events inductively as follows:

1. Events that appear on the same lifeline are ordered from top to bottom.[4]
2. A message is always sent before it is received.
3. If there are synchronous messages between two execution specifications a and b, then a and b are connected.
4. If execution specifications a and b are connected, and b and c are connected, then a and c are connected.
5. All events on connected execution specifications are grouped into the same thread.
6. Let us suppose that in an event group, a synchronous message m is sent from execution specification a to execution specification b, then the events on b should always be before the next event on a.

[4] This rule introduces forced orders between events of different threads on the same *Lifeline*.

Now consider applying the above inference rules to interpret *Example 2*. From the diagram, the observed orders are $!c1 <?c1 <!m1 <!c2 <?c2$ and $!m1 <?m1$ which is what we want.

Example 6 shown in Figure 4 illustrates a scenario in which *a:A* is a window object that can accept inputs from an actor. When an asynchronous message arrives, one of the *a:A* methods is activated. The activated method handles the message by calling methods of the connected participants. In this case, the GUI libraries of most programming languages will put the two events on lifeline *a:A* in the same thread[5] and the desired orders of this diagram are $!c1 <?c1 <!m1 <!c3 <?c3 <!m2 <?m2$ and $?m1 <?m2$. However, correct thread information cannot be produced by the rules above and the desired orders cannot be generated. This is due to the fact that, when applying inference rules 3, 4 and 5 to this diagram, the events will be separated into two event groups. The orders obtained by applying the inference rules will be $!c1 <?c1 <!m1 <?m1$, $!c3 <?c3 <!m2 <?m2$, $!c1 <!c3$ and $?m1 <?m2$. In this case, desired relations such as $?c1 <?c3$ and $?c1 <!c3$ are lost.

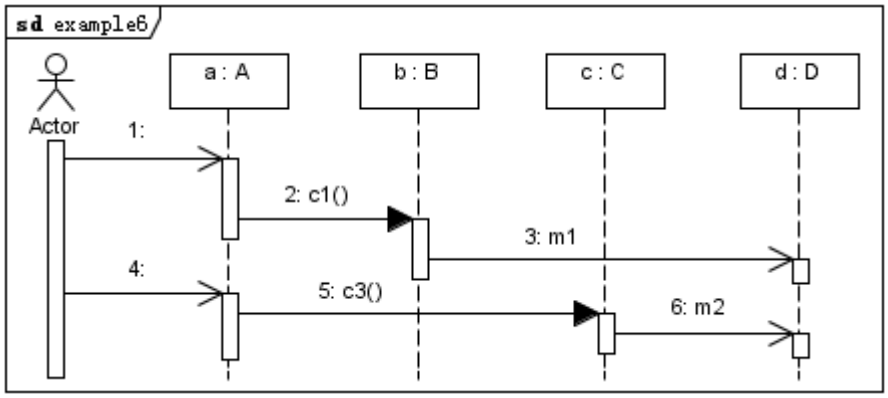

Fig. 4. SDs thread mapping problem

Although execution specifications do not always provide enough information for thread mapping in complicated SDs, these inference rules are still useful because grouped events belong to the same thread.

To apply these inference rules, one issue has to be clarified. According to UML 2.0, overlapping execution specifications on the same lifeline should always be represented by overlapping rectangles. However, a number of UML modelling tools do not follow this definition and this introduces problems in our inference rules.

[5] For example, two Java GUI libraries, Swing and SWT and Visual C++'s MFC.

There are two circumstances in which overlapping execution specifications will occur. Firstly, in the case of callback methods and secondly, for concurrent re-entering methods in the same lifeline.

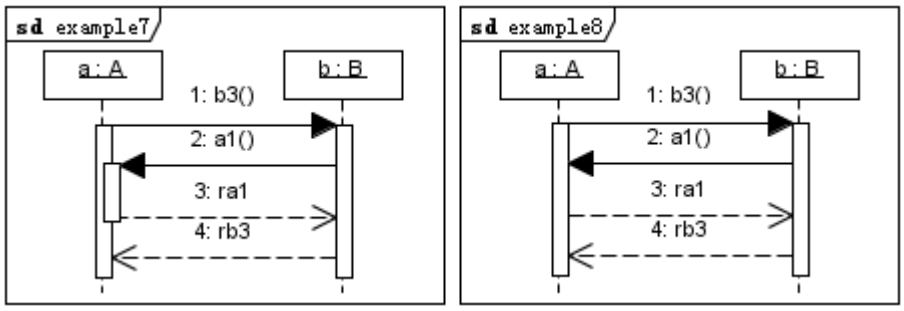

Fig. 5. Callback method

According to UML 2.0 standard, callback methods should be shown as *Example 7* in Figure 5. Some UML tools depict the callback method as *Example 8* in the same figure and although these tools violate the standard definition, our inference rules still apply because all events of a callback method belong to the same thread.

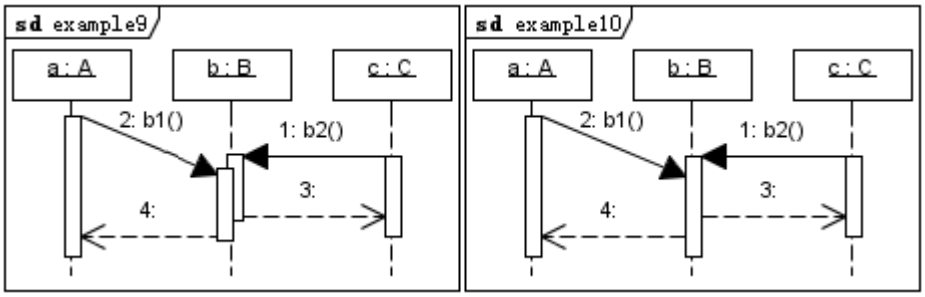

Fig. 6. Re-entering methods of the same lifeline

Concurrent re-entering methods of the same lifeline should be shown as *Example 9* in Figure 6. Some UML tools depict it as in *Example 10* in the same figure. When inference rules 3, 4 and 5 in the previous section are applied to *Example 10*, all execution specifications and events in the diagram are grouped to the same thread. This deduction conflicts with what actually happens, since $b1$ and $b2$ should belong to different threads in such scenarios. If inference rule

6 is applied subsequently to *Example 10*, the next event of $!b2$ on lifeline $c : C$ should follow all events on $b : B$ belonging to the same thread. This means that there exists an order $!4 <!3$, thus the events in the diagram may form a circle following this order, conflicting with the definition of partial order.

The semantics of execution specification will therefore be damaged if overlapping execution specifications are not depicted strictly according to the UML 2.0 standard; our inference rules do not work in this instance.

4.2 Using Active Object

In the UML standard related to SD, the only concept related to concurrency is that of active object.

> *A class may be designated as active (i.e., each of its instances having its own thread of control) or passive (i.e., each of its instances executing within the context of some other object).* [OMG05, p423]
> *An active object is an object that, as a direct consequence of its creation, commences to execute its classifier behavior, and does not cease until either the complete behavior is executed or the object is terminated by some external object. (This is sometimes referred to as "the object having its own thread of control.")* [OMG05, p424]

When an instance of a class with *isActive* property is set to be *true*, it is an active object, otherwise it is a passive object.

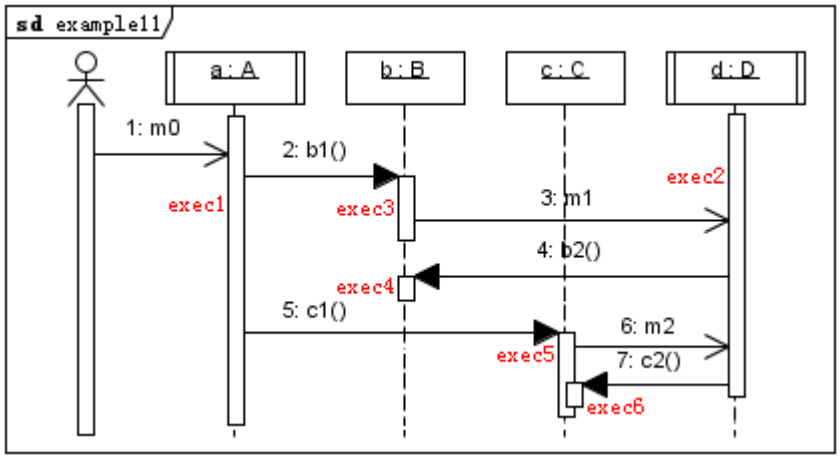

Fig. 7. An SD with active objects

Let us assume that active objects are represented by some lifelines, and execution specifications are fully specified; we could then claim that events can

be mapped to threads using the inference rules in Section 4.1. To explain, *Example 11* is provided in Figure 7. Active objects are represented by rectangles, each with an additional vertical bar on either side (eg. *a:A* and *d:D* in *Example 11*). Since active objects are active from creation to termination, the execution specifications of active objects persist from top to bottom of the lifelines in this diagram. We assume that each active object in the diagram contains a thread. In addition, to simplify the discussion, some terms representing the execution specifications are added to the diagram, for example $exec1$ refers to the execution specification on lifeline *a:A*. The detailed inference steps are:

- By applying rule 1, orders $?m0 <!b1 <!c1$, $?b1 <!m1 <?b2$, $?c1 <!m2 <?c2$ and $?m1 <!b2 <?m2 <!c2$ are obtained.
- By applying rule 2, orders $!b1 <?b1$, $!m1 <?m1$, $!b2 <?b2$, $!c1 <?c1$, $!m2 < ?m2$ and $!c2 <?c2$ are obtained.
- By applying inference rule 3, the pairs $exec1$ and $exec3$, $exec1$ and $exec5$, $exec2$ and $exec4$, $exec2$ and $exec6$ are connected respectively.
- By applying rules 4 and 5, $exec1$, $exec3$ and $exec5$ are connected; events $?m0$, $!b1$, $?b1$, $!m1$, $!c1$, $?c1$ and $!m2$ belong to the thread containing active object $a : A$. Similarly, $exec2$, $exec4$ and $exec6$ are connected; events $?m1$, $!b2$, $?b2$, $?m2$, $!c2$ and $?c2$ belong to the thread containing active object $d : D$.
- By applying rule 6, orders $?b1 <!c1$, $!m1 <!c1$ and $?b2 <?m2$ are obtained.

After applying these inference steps, the union of all obtained orders are: $?m0 <!b1 <?b1 <!m1 <!c1 <?c1 <!m2$, $?m1 <!b2 <?b2 <?m2 <!c2 <?c2$, $!m1 <?b2$ and $!m2 <?c2$, as expected.

But in OO software, it is hard to judge whether a lifeline represents an active object or not, since active object is defined more specifically than lifeline and, in most situations, they are not equivalent.

The concept of active object originates from research into Concurrent Object Oriented Programming Language (COOPL) [KL89,Nie93]. Active objects of COOPL keep both concurrency and OO features, such as encapsulation and inheritance, together. Consequently, the structure of active objects is generally more complex than common objects in Object Oriented Programing Language (OOPL). Mainstream OOPLs such as Java and C++ use a different approach to realize concurrent computing. They utilize special entities in the language itself or OS to implement concurrent computing, such as *Thread* class in Java and *process* or *thread* in Windows OS. Other research has shown how to implement active objects using normal OOPLs to benefit concurrent programming [CKV98,LS96]. In [LS96], active object is a behavioral pattern with multiple participants, such as Proxy, Scheduler, Servant etc. As a result, using a single lifeline to represent an active object for common OO software is unreasonable.

To summarise, these thread mapping approaches are impossible because the concurrent information kept by UML 2.0 meta-classes is not sufficient for doing so.

5 Inference for SDs with Thread Tags

Since there appears to be no canonical way to map events to threads with UML 2.0 meta-classes, we propose a new approach that extends the notation of UML 2.0. The extension should have two functions: firstly, to group all events in one SD to different threads; secondly, to maintain the temporal order of the events belonging to one thread. A straightforward solution is provided by using thread tags to retain the concurrent information of the systems being modelled. *Example 12* in Figure 8 shows an SD with extended thread tags. In this approach, an *id* is given to every thread in an SD. Each message is tagged with two thread *ids*, one for the source thread and one for the target thread. However, when sending and receiving of a message belong to the same thread, only one thread *id* is tagged in the middle of the message instead of two. The *ids* are then used to classify events into different threads while the temporal order of the grouped events is kept by the positions where the events occur.

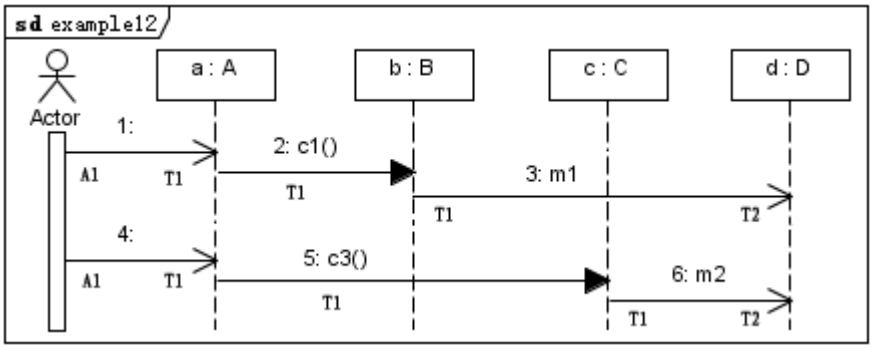

Fig. 8. An SD with thread tags

With a tagged SD, if we only consider the events of synchronous messages, then the orders of events of a *single thread* can be easily obtained using the following inference rules:

1. A message is always sent before it is received.
2. The events should be ordered linearly along the SD.

In the following text, we use $<_T$ to represent the orders obtained from thread tags[6] and $<_L$ to represent the orders obtained from lifelines[7]. We observe that there are differences between $<_T$ and $<_L$. Intuitively, $<_T \setminus <_L$ represents those sound orders that are missing when applying traditional partial order semantics. But it is also worth considering what $<_L \setminus <_T$ means.

[6] Orders are obtained by applying the above inference rules to every thread in the SD.
[7] Orders are obtained by applying the first inference rule in Subsection 2.2 to the SD.

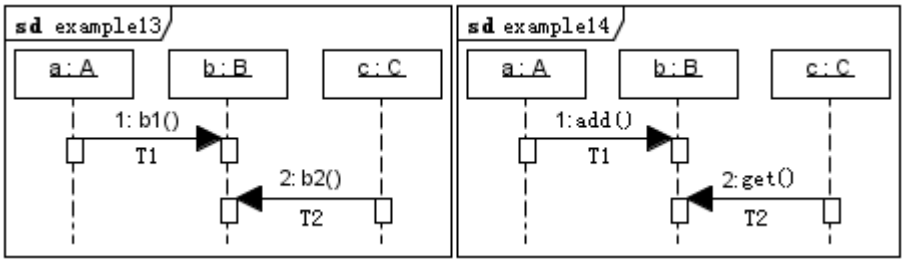

Fig. 9. What does $<_L \setminus <_T$ mean?

Recall *Example 4* in Figure 3; if we tag *Example 4* with thread *id*, we get *Example 13* as shown in Figure 9. Applying the rules for ordering events in one thread, only $!b1 <?b1$ and $!b2 <?b2$ are observed. Applying the first inference rule for inferencing traditional partial order set, we can get one more order relation $?b1 <?b2$. Since threads $T1$ and $T2$ run concurrently, events of $T1$ can interleave with the events of $T2$, so $?b1 <?b2$ is redundant. It is reasonable to remove the orders obtained using lifeline information from the partial order set while there are thread tags in SDs.

Sometimes, forced orders need to be added to the events of different threads. *Example 14* shows a similar scenario to *Example 13*. The only difference is that the messages have two different signatures. Intuitively, the traditional semantics of this diagram are meaningful. It describes a scenario that $c : C$ can get something from $b : B$ only after $a : A$ has added something to $b : B$.

The dilemma is whether the orders from lifelines should be preserved. If they are, some redundant orders will be added to the final partial order set when we represent parallel executions in one SD. If they are removed, extra meta-classes are needed to maintain the forced orders in the SDs. In fact, there is a meta-class, *GeneralOrdering*, used to express the forced order relation between two events [OMG05, p466]. The notation of *GeneralOrdering* is shown by a dotted line connecting the two events and the direction of the relation is given by an arrowhead placed in the middle of the dotted line. When compared with the first case which may introduce errors into SDs, we believe that using *GeneralOrdering* to maintain the forced orders in the second case is a credible solution.

Moreover, since forced orders are ignored in traditional semantics, we adopted in the Section 2.2, we will also ignore forced orders in SDs here when interpreting thread tagged SDs. The inference rules for interpreting tagged SDs can be revised as follows:

1. A synchronous message is always sent before it is received.
2. The events tagged with the same thread *id* should be ordered linearly along the SD even if the events are on different lifelines.

The traditional inference rules only need positional information about events on each lifeline, but the proposed rules need all event positional information in

one thread. Using the proposed rules, it is easy to infer the exact orders from the tagged SD even without the execution specifications. For instance, for *Example 12* shown in Figure 8, because $!c1$, $?c1$, $!m1$, $!c3$, $?c3$, $!m2$ all belong to thread *T1*, the orders are $!c1 < ?c1 < !m1 < !c3 < ?c3 < !m2$, $?m1 < ?m2$, $!m1 < ?m1$ and $!m2 < ?m2$ as desired.

Finally, without considering the forced orders, an informal semantics for SDs based on partial order theory can be defined as the transitive closure of the union of the following two orders:

- the union of orders of events belonging to the same thread;
- the ordering relation between the event pairs of sending and receiving of a message.

6 Conclusion and Future Work

In this paper, some primary differences between SDs and bMSCs were analysed. Based on these differences, we argued that traditional semantics for SDs had drawbacks when interpreting SDs. Two meta-classes of UML 2.0 were used to resolve the problems within traditional semantics. However, these meta-classes cannot maintain concurrency information needed in order to interpret SDs. As a consequence, an informal semantics for SD with thread tags was proposed. We believe that intended event sequences can be generated by applying this semantics to SDs.

An important area of future work is the development of a formal semantics for SDs with thread tags and then extend it to the Interaction Diagrams (IDs) of UML 2.0. In addition to developing the semantics of IDs, it would also be interesting to conduct a formal analysis of IDs based on the developed semantics, for example, identifying the pathologies of IDs and ID model checking. One of the problems considered in this paper is caused by the absence of return messages. An alternative solution may be to infer these missing return messages but the use of such an approach is a topic for future work.

References

[AHP96] R. Alur, G. Holzmann, and D. Peled. An analyzer for message sequence charts. *Software Concepts and Tools*, 17(2):70–77, 1996.

[CK04] M. V. Cengarle and A. Knapp. UML 2.0 interactions: Semantics and refinement. In *Proceedings of the 3rd Intl. Workshop on Critical Systems Development with UML*, pages 85–99, Lisbon, Portugal, 2004. Technische Universität München.

[CKV98] D. Caromel, W. Klauser, and J. Vayssiere. Towards seamless computing and metacomputing in Java. *Concurrency Practice and Experience*, 10(11-13):1043–1061, 1998.

[DH01] W. Damm and D. Harel. LSCs: breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 7 2001.

[GGR93] J. Grabowski, P. Graubmann, and E. Rudolph. Towards a petri net based semantics definition for message sequence charts. In *Proceedings of SDL'93 - Using Objects*, pages 179–190, Darmstadt, Germany, 1993. North-Holland.

[GS05] R. Grosu and S. A. Smolka. Safety-liveness semantics for UML 2.0 sequence diagrams. In *Proceedings of the Fifth International Conference on Application of Concurrency to System Design*, pages 6–14, Los Alamitos, CA, USA, 2005. IEEE Computer Society Press.

[HHRS05] Ø Haugen, K. E. Husa, R. K. Runde, and K. Stølen. STAIRS towards formal design with sequence diagrams. *Software and Systems Modeling*, 4(4):355–357, 2005.

[HM06] D. Harel and S. Maoz. Assert and negate revisited: modal semantics for UML sequence diagrams. In *Proceedings of the 2006 International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools*, pages 13–20, Shanghai, China, 2006.

[IT98] ITU-T. ITU-T Recommendation Z.120 Annex B: Formal semantics of message sequence charts, 4 1998.

[JP01] B. Jonsson and G. Padilla. An execution semantics for MSC-2000. In *Proceedings of SDL 2001: Meeting UML*, volume 2078 of *Lecture Notes in Computer Science*, pages 365–378, 2001.

[KL89] D. G. Kafura and K. H. Lee. Inheritance in actor based concurrent object-oriented languages. *The Computer Journal*, 32(4):297–304, 1989.

[LL93] P. B. Ladkin and S. Leue. What do message sequence charts mean? In *Proceedings of the IFIP TC6/WG6.1 Sixth International Conference on Formal Description Techniques*, pages 301–316, Boston, MA, USA, 1993. North-Holland.

[LS96] R. G. Lavender and D. C. Schmidt. Active object: an object behavioral pattern for concurrent programming. *Pattern Languages of Program Design*, pages 483–499, 1996.

[LS06] M. S. Lund and K. Stølen. A fully general operational semantics for UML 2.0 sequence diagrams with potential and mandatory choice. In *Proceedings of Formal Methods 2006*, volume 4085 of *Lecture Notes in Computer Science*, pages 380–395, 2006.

[Mit05] B. Mitchell. Resolving race conditions in asynchronous partial order scenarios. *IEEE Transactions on Software Engineering*, 31(9):767–784, 2005.

[MR94] S. Mauw and M. A. Reniers. An algebraic semantics of basic message sequence charts. *The Computer Journal*, 37(4):269–277, 1994.

[Nie93] O. Nierstrasz. Regular types for active objects. In *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 1–15, Washington, D.C., USA, 1993.

[OMG05] OMG. Unified Modeling Language: Superstructure, 8 2005.

[Sel04] B. V. Selic. On the semantic foundations of standard UML 2.0. In *Proceedings of the 4th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Real Time*, volume 3185 of *Lecture Notes in Computer Science*, pages 181–199, Bologna, Italy, 2004.

[Sto03] H. Storrle. Semantics of interactions in UML 2.0. In *Proceedings of the 2003 IEEE Symposium on Human Centric Computing Languages and Environments*, pages 129–136, Los Alamitos, CA, USA, 2003.