Fifth IEEE International Conference on Software Engineering and Formal Methods

A Thread-tag Based Semantics for 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

The sequence diagram is one of the most popular behaviour modelling languages which offers an intuitive and visual way of describing expected behaviour of Object-Oriented software. Much research work has investigated ways of providing a formal semantics for sequence diagrams. However, these proposed semantics may not properly interpret sequence diagrams when lifelines do not correspond to threads of controls. In this paper, we address this problem and propose a thread-tag based sequence diagram as a solution. A formal, partially ordered multiset based semantics for the thread-tag based sequence diagrams is proposed.

Keywords: *Sequence Diagram, Semantics, Partially ordered multiset, Concurrency, Object-Oriented, Thread tag.*

1 Introduction

The Sequence Diagram (SD) is a popular Unified Modelling Language (UML) behaviour modelling tool. It is a versatile technique that can be used in many parts of the Object-Oriented (OO) software development process. For example, it is usually used to capture system requirements and model function logic at the analysis and design stages. In addition, SD based specifications are also used as input to model-checking and model-based testing (MBT). These latter approaches are based on the assumption that SDs are able to maintain enough system runtime information so that their applications can be accurately interpreted.

In certain circumstances, however, SDs can not preserve concurrency information to support the aforementioned formal methods. One of the most important reasons is that lifelines in SDs are generally orthogonal to threads of control (threads) [1] . A lifeline in an SD represents an object or an instance of a component. The events happening along a lifeline may also happen in different threads. This means that, in a standard SD, it is impossible to infer which thread the events belong to and so the correct ordering of events can not be inferred from SDs.

To solve this problem, we propose a thread-tag based approach in which each *OccurrenceSpecification* in an SD is tagged with a label indicating the thread containing the *OccurrenceSpecification*. Intuitively, by using thread tags, the events in an SD belonging to the same thread are grouped together; the grouping contains desired concurrency information which is the key to correct interpretation of SDs.

The observed problem of SDs also induces partial invalidity in all currently applied semantics such as trace based semantics [21, 10] and partially ordered multiset (pomset) based semantics [4]. These semantics are based on the UML 2.0 standard and interpret the order of events in a lifeline by reading the diagram from top to bottom. For that reason, we believe interpreting SDs using current semantics is error prone when multiple threads are involved into single SDs. We thus propose a thread-tag based semantics for effective interpretation of SDs.

The informal thread-tag based semantics for SDs was first introduced in previous work by the authors [6]. There in, only serval core meta-classes of UML 2.0 were considered. The flow control related meta-classes, *CombinedFragment*, *InteractionOperand* and *InteractionOperater* were not included (these three meta-classes are newly introduced members of UML 2.0 that have largely enhanced the expressive power of SD). In this paper, we extend that previous work by defining the semantics for the above three meta-classes. In addition, we upgrade the proposed thread-tag based semantics to be a formal denotational semantics for SDs based on a pomset framework [19].

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

0-7695-2884-8/07 $25.00 © 2007 IEEE
DOI 10.1109/SEFM.2007.19

The remainder of this paper is structured as follows. Related research on semantics of SDs are introduced in Section 2. The abstract syntax of SDs and the traditional semantics for plain SDs are briefly introduced in Section 3. In Section 4, a thread based analysis of SDs is given to reveal problems of the current UML 2.0 standard. A set of informal inference rules for interpreting SD based on thread tags are then introduced to tackle the identified problems. In Section 5, a formal semantics based on pomset framework for complex thread-tag based SDs is proposed. Finally, the paper closes with conclusions and potential future work in Section 6.

2 Related Work

When discussing the semantics of SDs, it is worth reviewing previous research into Message Sequence Charts (MSCs). MSCs are the ancestor of SDs and in UML 2.0, SDs were significantly revised to allow adequate modelling of complex software systems based on the new version of MSC [11].

Mauw and Reniers [15] used a process algebra to interpret the semantics of basic MSCs. This approach has been adopted as the standard semantics for MSC [11]. Grabowski et al. [7] proposed petri-net based semantics for MSCs. Ladkin and Leue [14] used Büchi Automata to capture the meaning of MSC; Jonsson and Padilla [12] used Abstract Execution Machines to describe MSC semantics and at the same time, considered inline expressions and data in MSCs. Alur et al. [2] were the first to use labelled partially ordered structures to formalize basic MSCs; Katoen and Lambert [13] extended the idea using a pomset framework based on V. Pratt's work [19].

The increased popularity of UML has led to the semantics of SDs receiving more attention. Although UML 2.0 tried to provide semantics for every modelling language using a meta-model approach [20], SDs have only been assigned an informal semantics based on both trace and partial order theories. In [21, 10], formal trace based semantics for SDs were provided and [4] delivered a semantics for SDs following the pomset framework approach [13]. In [8], safety and liveness properties were used for distinguishing valid behaviours. Harel and Maoz [9] proposed Modal UML Sequence Diagrams (MUSD), an extension of SDs based on the approach used in Live Sequence Charts (LSCs), to extend MSCs [5]. These SD semantics were based on different kinds of MSC semantics. These MSC semantics were revised to fit with UML 2.0 including additional semantics for the new meta-classes of UML 2.0 (such as the interaction operators *assert*, *alt* and *neg*).

Since the current semantics of SDs are largely based on MSC semantics, the core principles of interpreting SDs have also been inherited from them. For example, the semantics always assume that the *OccurrenceSpecifications* of a lifeline should be ordered from top to bottom [21, 10, 4]. This assumption is true for MSCs, because each participant of an MSC has a thread of control. But it is not the case when it is applied to an SD, because a lifeline only represents an instance of OO programming language element such as *class* or *component*, which can be orthogonal to concurrency information, i.e., there is no longer a one–to–one correspondence between lifelines and threads of control. This is the main motivation of the research described in this paper.

3 SDs of UML 2.0

In this section, we briefly review the SDs' abstract syntax and the traditional semantics for SDs induced by UML 2.0.

3.1 Abstract syntax

The proposed approach of building a thread-tag based semantics for SDs is based on the UML 2.0 abstract syntax. This semantics is defined closely following the spirit of UML 2.0.

SD is a complicated modelling language that uses meta-methods to define itself as hierarchies of meta-classes. In fact, the semantics of SD can be assigned to these meta-classes and each construct contains a portion of the meaning of the whole language. The language has a very large hierarchy of meta-classes which can not all be covered in this paper. For this reason, we concentrate on the following meta-classes relating to behaviourial semantics: *InteractionFragment*, *Lifeline*, *Message*, *OccurrenceSpecification*, *GeneralOrdering*, *CombinedFragment*, *InteractionOperand* and *InteractionOperators* [2].

The part of meta-model related to the mentioned meta-classes is shown in Figure 1. In this model, an *Interaction* means a behaviour modelled by SD, although it can also be an interaction overview, a communication or a timing diagram.[3] An *Interaction* contains *Lifelines*, *Messages* and *InteractionFragments*. A *Lifeline* represents a *ConnectableElement* which refers to an instance of *Device*, *Component* and *Class*. An *InteractionFragment* is a piece of an *Interaction*. A *CombinedFragment* is a subclass of *InteractionFragment* and the type of a *CombinedFragment* is defined by an *InteractionOperator* whose value can be *seq*, *alt*, *opt*, etc. An *InteractionOperand* is contained in a *CombinedFragment*. An *InteractionOperand* represents one operand of the expression given by the enclosing *CombinedFragment*. *Message* represents the communication between *ConnectableElement* and *MessageSort*, a

[2] The meta-classes in UML 2.0 are always denoted in "*CamelCaps*", in the remainder of the paper, as they are in the UML 2.0.

[3] In this paper, only SDs are considered, therefore an *Interaction* refers to SD.

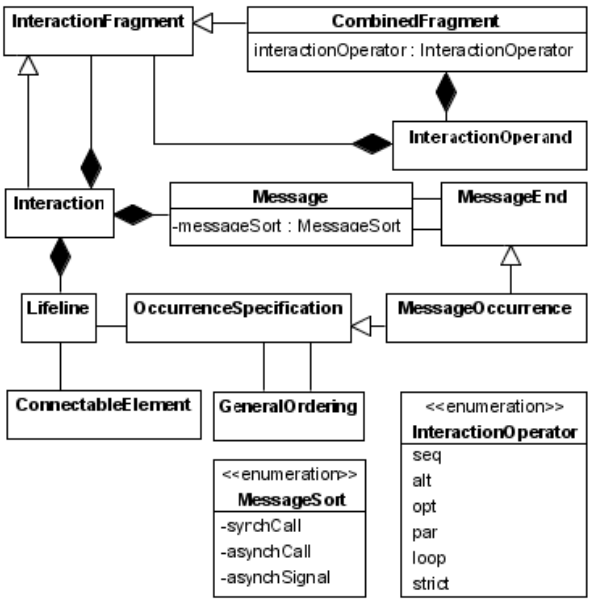

Figure 1. Meta-model of core meta-classes included in SDs

property of *Message* indicating its type. A *Message* contains a pair of *MessageEnds*. *MessageEnd* is an abstract class and there are two concrete meta-classes derived from it: *gate* and *MessageOccurrenceSpecification*. *MessageOccurrenceSpecification* has another superclass – *OccurrenceSpecification*. *OccurrenceSpecifications* are ordered points along *Lifelines* representing the moments when something happens. Since *MessageOccurrenceSpecification* is derived from both meta-classes, it is able to connect a *Message* and a *Lifeline*. In graphical notation, it is the intersection point of a vertical *Lifeline* and a horizontal *Message*. *ExecutionSpecification* is a specification of the execution of a unit of behavior or action within the *Lifeline*. It contains a pair of *OccurrenceSpecifications* representing the start and end of the execution. In addition, the start *OccurrenceSpecifications* commonly refers to receiving a message that triggered the *ExecutionSpecification*. A *GeneralOrdering* represents a binary relation between two *OccurrenceSpecifications* to state that one *OccurrenceSpecification* must occur before the other.

3.2 Traditional semantics

Although UML 2.0 does not provide a formal behavioural semantics, the descriptions of the core meta-classes in Figure 1 show us an attempt to provide a partial order semantics for plain SDs. *Plain SDs* refer to SDs containing the following meta-classes: *Lifeline*, *Message*, *OccurrenceSpecification* and *GeneralOrdering*.

> *An* OccurrenceSpecification *is the basic semantic unit of* Interactions*. The sequences of occurrences specified by them are the meanings of* Interactions.[18, p481]
>
> OccurrenceSpecifications *are ordered along a* Lifeline*.* [18, p481]
>
> *A* GeneralOrdering *is introduced to restrict the set of possible sequences. A partial order of* OccurrenceSpecifications *is defined by a set of* GeneralOrderings*.* [18, p467]

Along with these three definitions, the ordering information among *OccurrenceSpecifications* can also be inferred from the fact that "the sending of a message must be ordered before the receiving of a message".

An informal partial order semantics for plain SDs can be define as the transitive closure of the union of the following three orders:

1. *OccurrenceSpecifications* are ordered along a *Lifeline*;
2. sending *OccurrenceSpecification* always occurs before the corresponding receiving *OccurrenceSpecification*;
3. the orders are defined by *GeneralOrderings* and their directions; rules.

For the sake of convenience, we refer to this informal semantics of plain SDs as *traditional semantics*, because it reads the positions of *OccurrenceSpecifications* along the *Lifeline* for the ordering information.[4]

4 Thread based analysis of sequence diagrams

In this section, the issues of traditional semantics of plain SDs are first described. A Thread-tag based SD and its corresponding informal semantics are then proposed to resolve these issues. These two parts are based on results from [6]. In addition to presenting the two parts in detail, [6] also provides an analysis of the primary differences between SDs and basic MSCs and argue that meta-classes of UML 2.0 can not be used to resolve the issues within traditional semantics. Finally, in this section, informal semantics for plain SDs are extended to more complex SDs in which three new meta-classes: *CombinedFragment*, *InteractionOperand* and *InteractionOperator* are considered. We refer to this kind of SD as a *complex SD*.

[4] We label it *traditional semantics* because it conforms to all semantics mentioned [21, 10, 4]

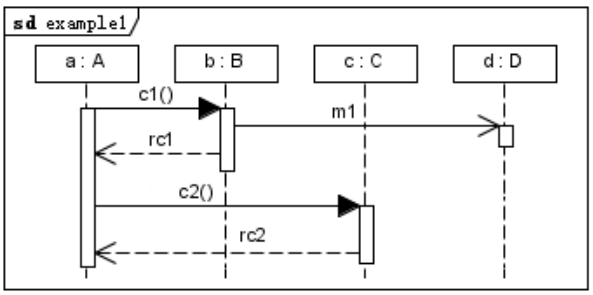

Figure 2. An SD with synchronous messages

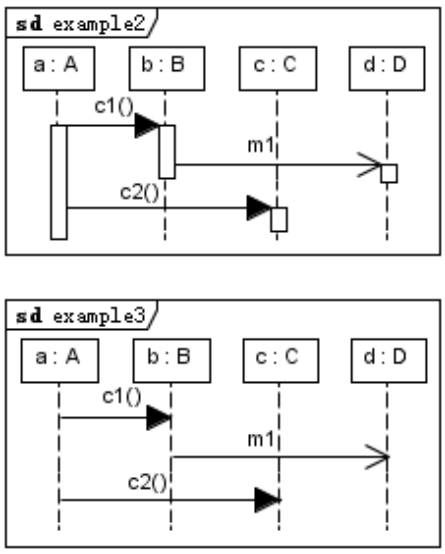

Figure 3. SDs with *synchCalls*

4.1 Traditional semantics for SDs

As aforementioned, *Lifelines* in an SD are generally orthogonal to concurrency information. This results in a situation where the desired order of *OccurrenceSpecifications* of an SD can not be correctly interpreted in some specific scenarios by traditional semantics. A number of SD examples are presented to illustrate this problem.

Example 1 in Figure 2 shows that synchronous messages convey the *OccurrenceSpecification* of one thread to multiple *Lifelines*. As a consequence, a normal *Lifeline* no longer represents a thread of control.

According to the UML 2.0, 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 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 *OccurrenceSpecifications* $!c1$, $?c1$, $!m1$, $!rc1$, $?rc2$, $!c2$, $?c2$, $!rc2$ 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).

Applying the traditional semantics introduced in Section 3.2 to this example, the orders of the *OccurrenceSpecifications* 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.

Example 2 is also a common SD, but the returns of the synchronous calls are not included. Applying traditional semantics, we get the 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 conforms to the partial order in *Example 1* except with reply *OccurrenceSpecifications* 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 *OccurrenceSpecifications* 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 $!m1 < !c2$.

The final example shows that users may draw a diagram based on the assumption that all the calls are in one thread and are synchronised; the assumed orders can not subsequently be retrieved from the diagram when it is formally analysed.

All three examples explain how synchronous messages bring the *OccurrenceSpecifications* 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.

An intuitive interpretation of *Example 4* shown in Figure 4 is that methods $b1$ and $b2$ in object *b:B* are called by *a:A* and *c:C* sequentially from different threads. This exam-

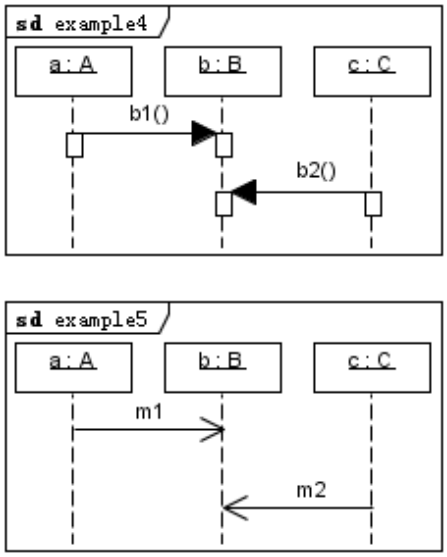

Figure 4. SDs of multiple threads entering one *Lifeline*

ple illustrates that sometimes it is impossible to determine whether the *OccurrenceSpecifications* 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 basic MSC, it induces a canonical race condition [2, 16]. 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.

These examples show that if there are synchronous messages and multiple threads in an SDs and thread information is not available, the correct ordering of *OccurrenceSpecifications* may not be correctly interpreted by traditional semantics.

4.2 SDs with thread tags

Since there appears to be no accurate mapping from *OccurrenceSpecifications* to threads with UML 2.0 metaclasses [6], we propose a new approach that extends the notation of UML 2.0. The extension should have two functions: firstly, to group all *OccurrenceSpecifications* in one

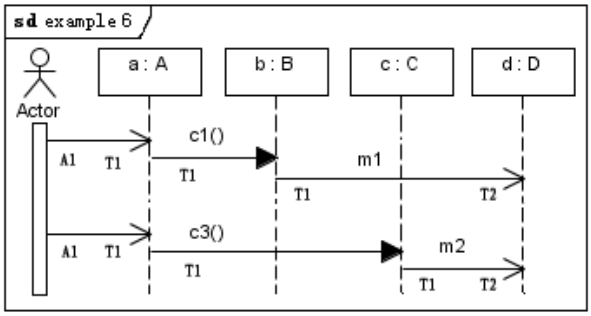

Figure 5. An SD with thread tags

SD to different threads; secondly, to maintain the temporal order of the *OccurrenceSpecifications* belonging to one thread. A straightforward solution is provided by using thread tags to retain the concurrency information of the systems being modelled [6]. *Example 6* in Figure 5 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 used to classify *OccurrenceSpecifications* into different threads while the temporal order of the grouped *OccurrenceSpecifications* is maintained by the positions where the *OccurrenceSpecifications* occur.

In a tagged SD, the *OccurrenceSpecifications* with the same thread tag are naturally grouped together. Based on the definition of thread, the *OccurrenceSpecifications* in the same thread should be sequentially ordered. In a tagged SD, this temporal ordering information is retained in the positions of the *OccurrenceSpecifications*. One *OccurrenceSpecification* should occur before another *OccurrenceSpecification* that resides below it in the same diagram (with the same thread tag). This means that the thread tags and the positions of the *OccurrenceSpecifications* preserve all ordering information for the *OccurrenceSpecifications* from the same thread.

According to our previous analysis, if the *OccurrenceSpecifications* are accurately tagged with thread tags, then the concurrency information retained in the thread tags is more reliable than the orders appearing in *Lifelines*. In addition, using thread tags also resolves issues caused by synchronous messages, since the sending and receiving of a synchronous message are always in the same thread. Therefore, when interpreting SDs with thread tags, we use the rule in the last paragraph to replace the first rule in traditional semantics that orders *OccurrenceSpecifications* alone *Lifelines*.

An informal partial order semantics for thread tagged

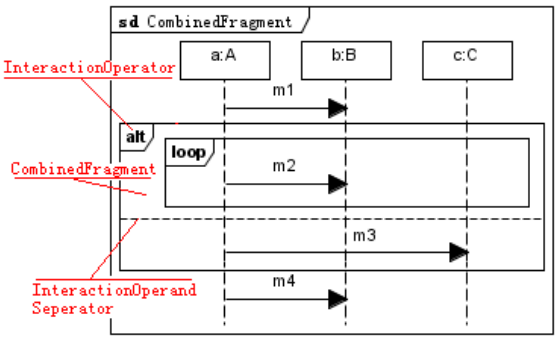

Figure 6. Combination of interactions

SDs can then be derived from the traditional semantics. It is the transitive closure of the union of the following three orders:

1. The *OccurrenceSpecifications* tagged with the same thread tag should be ordered linearly along the SD;
2. sending *OccurrenceSpecification* always occurs before the corresponding receiving *OccurrenceSpecification*;
3. the orders are defined by *GeneralOrderings* and their directions;

Using the proposed rules, it is easy to infer the exact orders from the tagged SD even without execution specifications. For instance, for *Example 6* (in Figure 5), 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.

4.3 Combination of interactions

In UML 2.0, SDs have been given expressive power to model the combination of interactions using *CombinedFragment*, *InteractionOperand* and *InteractionOperator*, as shown in Figure 6.

Using these three meta-classes, different kinds of sequential and parallel compositions, alternation and exception handling can be included in an SD.

A complex SD consists of plain SDs. The example in Figure 6 shows that a *CombinedFragment* splits an SD into three parts at the first level. The first part includes message $m1$; the second part is the whole *CombinedFragment*; the third part includes $m4$. The first and third parts in the SD are both plain SDs.

In addition, a *CombinedFragment* may have multiple *InteractionOperands*. Each *InteractionOperand* contains its *InteractionFragment*, which can be a plain SD or another complex SD. This implies that the *CombinedFragment* can nest in an *InteractionOperand* of its parent *CombinedFragment* recursively. In the example shown in Figure 6, a *CombinedFragment* with *loop* (*InteractionOperator*) is embedded in an *alt CombinedFragment*. It shows that complex SDs may consist of hierarchies of plain SDs for modelling multi-level behaviour in software.

According to the informal semantics for plain SDs, the semantic meaning of a complex SD can also be defined as partial orders among *OccurrenceSpecifications*. The informal semantics of plain SDs with thread tags can be easily achieved by the send-receive relation between the *OccurrenceSpecifications*, the *GeneralOrderings* and the sequential ordering of *OccurrenceSpecifications* belonging to the same thread. However, when inferring the meaning of a complex SD, one of the difficulties is how to define the combination types of the partial orders of the plain SDs in it. This problem can be resolved by defining semantics for each *InteractionOperator* in UML 2.0 as proposed in previous work [21, 10, 4].

Because the semantics change from plain SDs to thread tagged plain SDs, the semantics of *InteractionOperators* also needs to be changed when applying them to tagged SDs. In UML 2.0, the informal semantics for *InteractionOperators* are based on the possible traces of *OccurrenceSpecifications* in an SD. Here, we intend to identify the *InteractionOperators* whose semantics cannot be satisfied using this approach.

The most fundamental *InteractionOperator* in UML 2.0 is *seq*. It appears frequently in SDs, but possibly implicitly. For example, in Figure 6, it appears between two operands: the plain SD above an *alt CombinedFragment* and the *CombinedFragment* itself.

In UML 2.0, the semantics for *seq*, representing weak sequencing, is described as follows [20, p 454-455]:

1. *The ordering of* OccurrenceSpecifications *within each of the operands are maintained in the result.*
2. OccurrenceSpecifications *on different* Lifelines *from different operands may come in any order.*
3. OccurrenceSpecifications *on the same* Lifeline *from different operands are ordered such that an* OccurrenceSpecifications *of the first operand comes before that of the second operand.*

The second and the third ordering rules still use the positions of *OccurrenceSpecifications* on a *Lifeline* as the ordering information. This is not acceptable when there are multiple threads involved in an SD.

For example, Figure 7 shows a weak sequencing of three plain SDs (operand 1,2 and 3 respectively). When applying semantics for thread tagged SDs, the orders for each plain SD are $!m1 <?m1$ (operand 1), $!m2 <?m2$ (operand 2) and $!m3 <?m3$ (operand 3). There is no difference

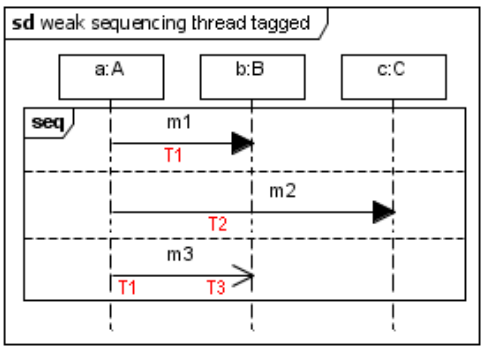

Figure 7. Weak sequencing in an tagged SD

between applying traditional or thread-tag based semantics to interpret these plain SDs. If the *seq* semantics in UML 2.0 is applied to combine these three orders, the orders should be $!m1 <?m1$, $?m2 <!m2$, $!m3 <?m3$ and $!m1 <!m2 <!m3$, $?m1 <?m3$. Now consider the thread tags in the SD, if the SD is tagged as shown in the example; the desired orders should be: $?m2 <!m2$, $!m3 <?m3$ and $!m1 <?m1 <!m3$. The difference between the orders shows that the semantics for weak sequencing is affected by the fact that *Lifelines* are orthogonal to concurrency information. To tackle this problem, the semantics for *seq* should be replaced by rules as follows:

1. The ordering of *OccurrenceSpecifications* within each of the operands are maintained in the result.

2. *OccurrenceSpecifications* with different thread tags from different operands may come in any order.

3. *OccurrenceSpecifications* with the same thread tag from different operands are ordered such that an *OccurrenceSpecifications* of the first operand comes before that of the second operand.

In the above rules, thread tags are used instead of *Lifelines* to retain the concurrency information in SDs.

Another affected *InteractionOperator* is *loop*. In UML 2.0, the semantics of *loop* is defined as follows:

- *The* loop *construct represents a recursive application of the seq operator where the* loop *operand is sequenced after the result of earlier iterations.*

Since these semantics are based on weak sequencing whose meaning has changed according to our previous analysis, the semantics for *loop Interactionoperator* are also changed.

5 Formal semantics

The pomset framework is chosen as the basis of our proposed semantics since it is a well-established model of concurrency in the class of linear-time, non-interleaving models [19]. A pomset framework consists of three main parts: the definition, the operations and temporal logic. The second part is the algebraic portion of the whole framework and can be used for formalising the semantics of the new meta-classes of UML 2.0: *CombinedFragement*, *InteractionOperand* and *InteractionOperator*. These meta-classes are mainly used to model the control flows of the OO programs. From the point of view of the pomset, however, the *CombinedFragment* can be simplified as a calculus on a set of pomsets, each of which represents an *InteractionOperand* in the *CombinedFragment*.

The formalisations of both MSCs and SDs using partially ordered structures have been advocated by many researchers [1, 17, 3]. Following their approaches, we propose a formal semantics based on our previous informal thread-tag based semantics. However, concurrency issues in traditional semantics are modified in the new semantics.

In the following section, the definition of a pomset related to our formalisation are given. The semantic domain of the proposed semantics where the thread tag information are included is then described. Finally, the semantics of the well understood operators, i.e. *strict*, *seq*, *par*, *alt*, *opt* and *loop*, are defined. For the time being, *neg*, *ignore*, *critical*, *consider*, *assert* and *break* are not considered.

5.1 Pomset

In this subsection, the formal definition of pomset and its calculus is given according to [13].

Definition 1 *Let L be a set of labels. A labelled partially ordered set (lposet) is a triple $(E, \leqslant, l)$ with E, a set of events, $\leqslant \in E \times E$, a reflexive, anti-symmetric, and transitive order on E, and $l : E \rightarrow L$ a labelling function.*

$\leqslant$ is called a partial order that represents causality. The empty lposet $(\emptyset, \emptyset, \emptyset)$ is denoted by ε. Commonly, for modelling concurrency, $L = A \times I$, where A is the set of actions and I the set of instances.

Definition 2 *$(E, \leqslant, l)$ and $(E', \leqslant', l')$ are isomorphic iff there exists a bijection $\phi : E \rightarrow E'$ such that $e \leqslant \hat{e}$ iff $\phi(e) \leqslant' \phi(\hat{e})$ and $l = l' \circ \phi$, for all $e, \hat{e} \in E$.*

Definition 3 *A pomset is an* isomorphism *class of lposets.*

The isomorphism class of $(E, \leqslant, l)$ is denoted by $[(E, \leqslant, l)]$.

Let Let $p = [(E_p, \leqslant_p, l_p)]$ and $q = [(E_q, \leqslant_q, l_q)]$ be two pomsets.

Definition 4 (Concatenation)

$$p \cdot q \triangleq [(E_p \cup E_q, \leqslant_p \cup \leqslant_q \cup (E_p \times E_q), l_p \cup l_q)].$$

In $p \cdot q$, every event of p is forced to precede every event of q. It is straightforward to check that $\leqslant_p \cup \leqslant_q \cup (E_p \times E_q)$ is a partial order, so $p \cdot q$ is indeed a pomset.

Definition 5 (Concurrence)

$$p \parallel q \triangleq [(E_p \cup E_q, \leqslant_p \cup \leqslant_q, l_p \cup l_q)].$$

In $p \parallel q$, events in p can randomly interleave with events in q. Also $p \parallel q$ is a pomset.

Definition 6 (Local concatenation)

$$p \circ q \triangleq [(E_p \cup E_q, \left(\leqslant_p \cup \leqslant_q \bigcup_i (E_p^i \times E_q^i)\right)^+, l_p \cup l_q)].$$

For pomset p the set E_p^i denote the set of events in p that occur at instance i, $i \in I$. In $p \circ q$, events in p should precede those events in q that appear at the same of instance. Notice that for $p \circ q$, a transitive closure is taken to guarantee that the resulting relation is indeed a partial order.

Definition 7 (Union)

$$p \cup q \textit{ is simply the pomset of } p \textit{ or pomset } q\ .$$

In $p \cup q$, p or q means that the choices between two pomsets. The $p \cup q$ is not a pomset but a set of pomsets.

5.2 Semantic domain

Recall the semantic domain for plain SDs in [4], which contains two subdomains for *Lifelines* $\mathbb{I}$ and *Messages* $\mathbb{M}$, I and M then represent the subsets of $\mathbb{I}$ and $\mathbb{M}$ respectively. The semantic domain of a plain SD is an event-labelled poset $[(E, \leqslant_E, \lambda_E)]$.

- E is the set of *OccurrenceSpecifications* in the SD;
- $\leqslant_E$ is a partial order, $\leqslant_E \in E \times E$, retained by the SD;
- λ_E is a labelling function which maps E to L, where $L = A \times I \times M$. A is the set of event types, which can be send and receive.

Considering the semantic domain of a plain thread-tag based SD, the most notable change is the use of thread tags to retain concurrency information instead of using *Lifelines* in an SD. The set of thread tags should be involved into the semantic domain of thread-tag based SDs.

To further our analysis, we define another subdomain in plain tagged SDs. Let's define the domain of thread tags, written $\mathbb{T}$, and then T is the subset of $\mathbb{T}$ to represent the threads involved in this SD. We keep the definition for A, which is a set of types, send, receive belonging to A; the set E for *OccurrenceSpecifications* occurring in that SD and M for the set of *Messages*. The *Lifelines* are orthogonal with the ordering information in a thread-tag based SD, but the set of *Lifelines* remains in the semantic domain for preserving information about where the message is sent and where it is received, written I. A label function can then be defined as $l : E \rightarrow A \times T \times I \times M$.

Based on the pomset definition, a plain tagged SD is also a pomset, $[(E, \leqslant_T, l)]$. Comparing with [4], there are two differences: one is the labelling function changed from $\lambda_E : E \rightarrow A \times I \times M$ to $l : E \rightarrow A \times T \times I \times M$, where a new subdomain, $\mathbb{T}$, is considered; the other is the partial order, $\leqslant_E$, is replaced by $\leqslant_T$ obtained by applying the deductive rules in Section 4.2.

When extending the formalisation to complex tagged SDs, a special *InteractionOperator* needs to be introduced. Alternative, written as *alt* represents a choice of behaviour, thus different traces can appear. It means that if there is a *CombinedFragment* with *alt* in a tagged SD, then the SD can not be mapped to a single pomset. The semantic domain of a complex SD thus comprises all pomsets of each plain SD in the complex SD. Let's assume that there are n plain SDs in a complex SD and E_i, $\leqslant_{iT}$ and l_i respectively represent the i_{th} set of *OccurrenceSpecifications*, partial order and label function , $i \in [0, n]$, then $\mathbb{P}$, the semantic domain is the combination of $[(E_i, \leqslant_{iT}, l_i)]$, $i \in [0, n]$, according to the different *Interactionoperators* attached with the plain SDs.

5.3 Formalising the *Interactionoperators*

On the basis of this semantic domain definition, the semantics for each well understood operator are separately defined. In the remainder, A and B denote plain tagged SDs; $[(A)]$ and $[(B)]$ thus refer to the pomsets of the SDs.

Strict sequencing In the UML 2.0, *strict* was defined as:*"the semantics of strict sequencing defines a strict ordering of the operands"*. According to the Definition 4, *strict* is then the concatenation of two pomsets as follows:

Definition 8 (strict)

$$[(\textit{strict}(A, B))] = \{a \cdot b |\ a \in [(A)], b \in [(B)]\}.$$

Weak sequencing In Section 4.3, we have listed the rules for weak sequencing two plain tagged SDs. Again, let A

and B be the two plain SDs. The *OccurrenceSpecifications* in A and B sharing the same thread tags should be strict sequenced but for the others, those from A can interleave with those from B. Considering Definition 9, events within the same instance should be ordered strictly. Because the instance should refer to a thread in tagged SDs according to our analysis in Section 4.3, we define a new operation, weak sequencing by thread, in the tagged SD semantic domain, written $\circledcirc$.

Definition 9 (thread concatenation)

$$p \circledcirc q \triangleq$$
$$[(E_p \cup E_q, (\leqslant_p \cup \leqslant_q \bigcup_t (E_p(t) \times E_q(t)))^+, l_p \cup l_q)].$$

For pomset p the set $E_p(t)$ denotes the set of *OccurrenceSpecifications* in p that occur at thread t (with thread tag t), $t \in T$. In $p \circledcirc q$, *OccurrenceSpecifications* in p should precede those *OccurrenceSpecifications* in q that appear at the same thread (with the same thread tag). Again, $p \circledcirc q$ needs a transitive closure to assure it is a pomset.

seq can then be defined as follows:

Definition 10 (seq)

$$[(\mathit{seq}(A, B))] = \{a \circledcirc b |\ a \in [(A)], b \in [(B)]\}.$$

Parallel In the UML 2.0, *strict* was defined as: *"the* OccurrenceSpecifications *of the different operands can be interleaved in any way"*. Then according to Definition 5, *par* is the concurrence of two pomsets as follows:

Definition 11 (par)

$$[(\mathit{par}(A, B))] = \{a \parallel b |\ a \in [(A)], b \in [(B)]\}.$$

Alternatives In the UML 2.0, *alt* was defined as: *"the set of traces that defines a choice is the union of the traces of the operands"*. Then according to Definition 7, *alt* is the union of two pomsets as follows:

Definition 12 (alt)

$$[(\mathit{alt}(A, B))] = [(A)] \cup [(B)].$$

Option In the UML2.0, *opt* was defined as: *"an option is semantically equivalent to an alternative* CombinedFragment *where there is one operand with non-empty content and the second operand is empty"*. Then according to Definition 7 and 12, *opt* is the union of one pomset and a empty pomset as follows:

Definition 13 (opt)

$$[(\mathit{opt}(A))] = [(A)] \cup [(\varepsilon)].$$

Loop According to the analysis in Section 4.3, *loop* should be defined by *seq* operator. The UML 2.0 states *"The loop operand will be repeated a number of times."* It is convenient to define *loop* recursively which means going through *loop* operand sequenced with the meaning of the *loop* itself or doing nothing. Its formal definition is as follows:

Definition 14 (loop)

$$[(\mathit{loop}(A))] = [(\varepsilon)] \cup [(\mathit{seq}(A, \mathit{loop}(A)))].$$

This means, that A can be executed any number of times and then exit.

6 Conclusion and Future Work

In this paper, a formal thread-tag based semantics of SDs was proposed. The drawbacks of traditional semantics for interpreting SDs and the informal semantics for plain thread tagged SD were reviewed [6]. It was shown that traditional semantics was not suitable for capturing concurrency information when there were multiple threads involved in a single SD and the *Lifelines* were replaced by thread tags to retain the concurrency information in SDs. Secondly, the semantics for three new meta-classes in UML 2.0, i.e. *CombinedFragment*, *InteractionOperator* and *InteractionOperand*, were considered. The change from traditional semantics of plain SD and its effect on the semantics of were thread-tag based SDs was extended to a formal semantics for complex thread-tag based SDs based on the pomset framework.

Planned future work includes the following. Different type of semantics for common SDs exist, such as trace based semantics. However, they all follow the traditional way of using *Lifeline* to maintain concurrency information. It would be interesting to apply our semantics on other forms of semantics and compare them. Since SDs are only one technique in the set of UML 2.0 interaction diagrams (IDs), it would be worth extending this semantics to IDs. In addition, it would also be useful to conduct a formal analysis of IDs based on the developed semantics, such as identifying the pathologies of IDs and ID model checking.

References

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

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

[3] P. Baker, P. Bristow, C. Jervis, D. King, R. Thomson, B. Mitchell, and S. Burton. Detecting and resolving semantic pathologies in UML sequence diagrams. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 50–59, Lisbon, Portugal, 2005. ACM Press.

[4] 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.

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

[6] H. Dan, R. M. Hierons, and S. Counsell. Thread-based analysis of Sequence Diagrams. *Accepted in the 27th International Conference on Formal Methods for Networked and Distributed Systems*, 2007.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] J. P. Katoen and L. Lambert. Pomsets for message sequence charts. In *Proceeding of First Workshop SDL and MSC (SAM'98)*, pages 197–208, Berlin, Germany, 1998.

[14] 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.

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

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

[17] A. Muscholl, D. Peled, and Z. Su. Deciding properties for message sequence charts. In *Proceedings of the 1st Conference on Foundations of Software Science and Computation Structures*, volume 1378 of *Lecture Notes in Computer Science*, pages 226–42, Lisbon, Portugal, 1998. Springer-Verlag.

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

[19] V. Pratt. Modeling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, 1986.

[20] 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.

[21] 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.