A Logical Approach for Behavioural Composition of Scenario-based Models

J. Küster Filipe Bowles[1], B. Bordbar[2], and M. Alwanain[2]

[1] School of Computer Science, University of St Andrews
Jack Cole Building, North Haugh, St Andrews KY16 9SX, UK
jkfb@st-andrews.ac.uk

[2] School of Computer Science, University of Birmingham
Edgbaston, Birmingham B15 2TT, UK
{b.bordbar|m.i.alwanain}@cs.bham.ac.uk

Abstract. As modern systems become more complex, design approaches model different aspects of the system separately. When considering (intra and inter) system interactions, it is usual to model individual scenarios using UML's sequence diagrams. Given a set of scenarios we then need to check whether these are consistent and can be combined for a better understanding of the overall behaviour. This paper addresses this by presenting a novel formal technique for composing behavioural models at the metamodel level through *exact metamodel restriction* (EMR). In our approach a sequence diagram can be completely described by a set of logical constraints at the metamodel level. When composing sequence diagrams we take the union of the sets of logical constraints for each diagram and additional behavioural constraints that describe the matching *composition glue*. A formal semantics for composition in accordance with the glue guides our model transformation to Alloy. Alloy's fully automated constraint solver gives us the solution. Our technique has been implemented as an Eclipse plugin SD2Alloy.

Keywords: Sequence diagrams, Behavioural Composition, Event Structures, Alloy

1 Introduction

As modern systems become more complex, design approaches model different aspects of the system separately. When considering (intra and inter) system interactions, it is usual to model individual scenarios using UML's sequence diagrams. Given a set of scenarios we then need to check whether these are consistent and can be combined for a better understanding of the overall behaviour. The overall behaviour of the system can be obtained step by step by composing individual scenario-based models.

Composing systems manually can only be done for small systems. As a result, in recent years, various methods for automated model composition have been introduced [4, 6, 18, 13, 15, 19–21, 23]. Most of these methods involve introducing algorithms to produce a composite model from smaller models originating from partial specifications [13]. By contrast, in this paper we focus on the composition of models via constraint solvers. This corresponds to producing a number of constraints capturing models and using an automated solver to find a solution that produces the composed model. In this paper, we use Alloy [12] for finding the solution. Using Alloy for model composition is an active area of research [19, 23]. Whilst most existing research focuses on static models, the focus of this paper is on dynamic models. The proposed method in this paper consists of two steps. First, create the *logical constraints* that uniquely characterise each model by restricting the metamodels. Second, produce *behavioural constraints* for combining the models. These consist of constraints indicating how elements from both models may be matched and additional constraints such as orderings that may have to be preserved. The augmented model for the composition (if existing) needs to satisfy the *conjunction of all these constraints*. The composed model is semantically equivalent to one obtained by an enriched form of parallel composition with synchronisation and additional constraints on permitted combined behaviour. The automatic generation of such a solution is the main novelty and contribution of this paper.

In general, metamodels represent the model elements and their relationships. Logical statements written in the context of metamodels play a key role in expressing the well-definedness of model elements, defining model equality, and so on. We extend the use of logical constraints and for a given model we produce further constraints to *uniquely* determine the model. We refer to the process of identifying such logical constraints as *Exact Metamodel Restriction* (EMR). As we show in this paper, EMR can be used in the automated instantiation of models via constraint solvers. For example, in [2] starting from any UML sequence diagram, using the Alloy model finder for the sequence diagram metamodel and correct set of constraints, Alloy can be used to automatically recreate the original sequence diagram. Given any two models M_1 and M_2 representing two partial specifications (e.g., two sequence diagrams), through EMR we produce two sets of constrains $\mathcal{L}_1$ and $\mathcal{L}_2$ on their metamodels that uniquely identify them. To compose the two models we may require *all* constraints in the two sets to be true. This would be a very restrictive form of composition. Instead we give the designer a novel way to influence the obtained composition by specifying behaviour that should never occur or sequences of events that must occur in a given order. In other words, it allows the designer to prioritise on specified behaviour. We refer to these additional constraints as *behavioural composition glue* and present a formal semantics for it.

The notion of glue is not new and is also used within software architecture to describe and formalise component connectors [1, 8]. Our interpretation of glue here is nonetheless more generic and not only a syntactic matching between component elements. Our behavioural glue gives us a new set of constraints $\mathcal{L}_g$ which specifies how the models should be *glued* together to produce the intended composition. Given the sets of constraints $\mathcal{L}_1$, $\mathcal{L}_2$ and $\mathcal{L}_g$, and provided there are no conflicts between them, the models can be composed automatically using Alloy. If there are conflicts between the constraints, Alloy will point out the conflicting statements so that we can redesign the models or the constraints used for the composition. Although the focus of work is on sequence diagrams, the suggested method can be applied to all models with a trace-based semantics. We have applied the method to sequence diagrams and produced an Eclipse plugin which was described in [2]. This work considerably extends the work in [2] by going beyond composition based on syntactic matching of model elements and focusing on the formalisation of *behavioural glue* for composition.

The paper is organised as follows. Section 2 describes interactions in UML and introduces an example which is used throughout the paper to illustrate our approach. Section 3 introduces labelled event structures (LES), our semantic interpretation of interactions and a guide to the correct composition solution. Section 4 shows the transformation into Alloy. Composition is treated with LES in Section 5 and with Alloy in Section 6. Related work is described in Section 7. Section 8 concludes the paper.

2 Interactions in UML

Sequence diagrams are described in UML's superstructure specification [17] both through a *concrete* and an *abstract syntax*. The concrete syntax consists of the graphical notation for a sequence diagram, whereas the abstract syntax is given by a metamodel which defines all the elements of a sequence diagram model and their possible relationships. An *instance* of the metamodel corresponds to a concrete sequence diagram.

Concrete Syntax: An interaction captured by a sequence diagram involves a group of objects which exchange messages between each other to achieve a particular goal. Each object has a vertical dashed line called *lifeline* showing the existence of the object at a particular time. Points along the lifeline are called *locations* (a terminology borrowed from LSCs [11]) and denote the occurrence of events. The order of locations along a lifeline is significant denoting, in general, the order in which the corresponding events occur. An *interaction* between several objects consists of one or more messages, but may be given further structure through so-called *interaction fragments*. There are several kinds of interaction fragments including **seq** (sequential behaviour), **alt** (alternative behaviour), **par** (parallel behaviour), **neg** (forbidden behaviour), **assert** (mandatory behaviour), **loop** (iterative behaviour), and so on [17].

Consider the following sequence diagrams which show a slightly adapted example from [10]. Fig. 1 (left) shows an interaction with two consecutive interaction fragments (a parallel followed by an alternative fragment), and Fig. 1 (right) shows a different interaction involving the same instances and a few additional messages.

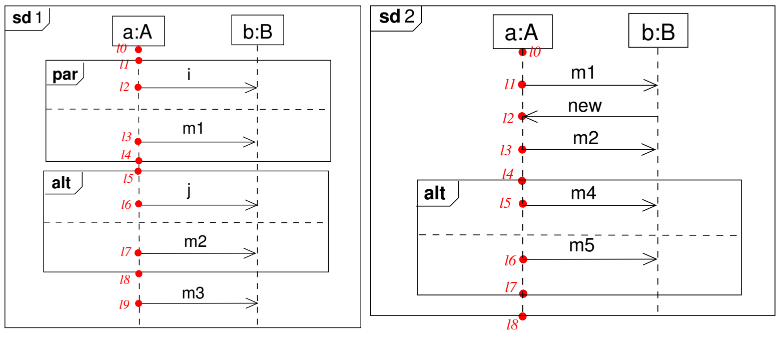

Fig. 1. Two sequence diagrams with fragments involving the same object instances.

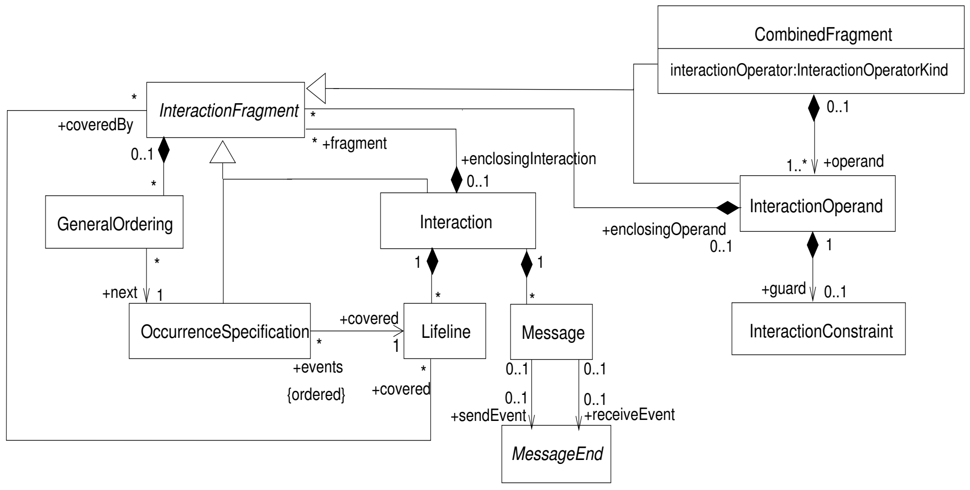

Fig. 2. The Interactions Metamodel.

In both diagrams, all messages are sent asynchronously between objects a and b (only message new is sent by b to a). The locations along the lifeline of object a are shown explicitly in both diagrams. The importance of locations is described later in the paper. In particular, the distinction between the syntactic notion of a location on a sequence diagram from its semantic counterpart of an event will be clarified. In Fig. 1 messages i and m_1 are sent/received in parallel followed by message j or message m_2 (alternative), and further followed by message m_3 (irrespective of the previous alternative choosen). In Fig. 1, three messages are sent/received before reaching an alternative fragment and choosing between messages m_4 or m_5. These diagrams will be used to show how we can compose diagrams under certain constraints.

Abstract Syntax: A metamodel can be understood as a model of a collection of models. A metamodel is usually a structural model given as a UML class diagram often with additional constraints given in UML's constraint language OCL. Metamodels can be built for both static and dynamic models but focus only on the structural aspects of the model. In this paper we look at sequence diagrams. The metamodel of a sequence diagram, also known as an interaction, shows the structure of such a diagram in terms of the model elements present and their relationships. The dynamic interpretation is not given in the metamodel, and must be defined separately. See ours in Section 3.

The UML superstructure specification [17] defines the interaction's metamodel in a package showing different elements and their relationships separately in different diagrams. To make the presentation simpler, we use a subset of the metamodel for interactions and show it as one class diagram in Fig. 2. We capture the main notions that we need for the present paper.

An `Interaction` contains zero or more instances of `Lifeline`, `Message` and `InteractionFragment`. A `Message` usually has a `sendEvent MessageEnd` and a `receiveEvent MessageEnd` associated to it. In the present paper, we assume that `MessageEnd` (an abstract class) is always a special kind of `OccurrenceSpecification` called MessageOccurrenceSpecification (not shown). It is possible for a Message to have been *found*, or similarly *lost*, in which case it does not have a sendEvent or a receiveEvent. A Message cannot be simultaneously found and lost. A Message has attributes messageKind and messageSort (not shown in the diagram). These attributes have a type with the same name which are enumeration types used to indicate whether a message is lost, found, complete or unknown (MessageKind), or a synchronous/asynchronous call, create Message and so on (MessageSort). A Lifeline has attributes for the name and class associated to the object that is denoted by the lifeline (not shown in the diagram). An InteractionFragment is an abstract class which is further specialised into an OccurrenceSpecification, an Interaction, a CombinedFragment or an InteractionOperand. The *locations* mentioned in Section 2 correspond to instances of OccurrenceSpecification. These are the ordered events that cover a Lifeline. A GeneralOrdering represents a binary relation between two OccurrenceSpecifications. The metamodel contains relations before and after, but we restrict ourselves to a relation next which is all we require for our purposes. A CombinedFragment has an attribute interactionOperator of enumeration type InteractionOperatorKind (par, alt, seq, loop, assert, and so on), and contains one or more operands which are InteractionOperands. An InteractionOperand may have a guard which is an InteractionConstraint. An InteractionOperand encloses either several OccurrenceSpecifications, an Interaction or another CombinedFragment indicating nesting of fragments.

An instance of the metamodel represents a concrete interaction or sequence diagram. The interaction from Fig. 1 can be captured using the abstract syntax as an instance of the metamodel (not shown here).

We have developed a tool SD2Alloy that takes a sequence diagram described by its abstract syntax and transforms it into an Alloy model. Alloy [12] is a declarative textual modeling language based on first-order relational logic. Alloy is supported by a fully automated constraint solver Alloy Analyzer which enables the analysis of system properties by searching for instances of the model. It is possible to check whether certain properties of the system are present. This is achieved via an automated translation of the model into a Boolean expression, which is then analysed by SAT solvers such as SAT4J [5] embedded within the Alloy Analyzer.

3 Semantics of Interactions

The dynamic interpretation of interactions is done in this paper using labelled event structures [22]. Several possible semantics for sequence diagrams have been defined (see [16] for an overview). Labelled event structures (LESs) are very suitable to describe the traces of execution in sequence diagrams being able to capture directly the notions available such as sequential, parallel and iterative behaviour (or the unfoldings thereof) as well as nondeterminism. For each of the notions we use one of the relations available over events: causality, nondeterministic choice and true concurrency. LESs are the only true-concurrent semantics for sequence diagrams available and first defined in [14]. We recall the main notions used for modelling sequence diagrams with LES. We later extend our semantics to model composition of diagrams with glue constraints.

Prime event structures [22], or event structures for short, describe distributed computations as event occurrences together with binary relations for expressing causal dependency (called *causality*) and nondeterminism (called *conflict*). The causality relation implies a (partial) order among event occurrences, while the conflict relation expresses how the occurrence of certain events excludes the occurrence of others. From the two relations defined on the set of events, a further relation is derived, namely the *concurrency* relation co. Two events are concurrent if and only if they are completely unrelated, i.e., neither related by causality nor by conflict. The formal definition as defined for instance in [22] is as follows.

Definition 1 *An* event structure *is a triple* $E = (Ev, \rightarrow^*, \#)$ *where* Ev *is a set of events and* $\rightarrow^*, \# \subseteq Ev \times Ev$ *are binary relations called* causality *and* conflict*, respectively. Causality* $\rightarrow^*$ *is a partial order. Conflict* $\#$ *is symmetric and irreflexive, and propagates over causality, i.e.,* $e\#e^{'} \rightarrow^* e^{''} \Rightarrow e\#e^{''}$ *for all* $e, e^{'}, e^{''} \in Ev$*. Two events* $e, e^{'} \in Ev$ *are* concurrent*,* $e\ co\ e^{'}$ *iff* $\neg(e \rightarrow^* e^{'} \vee e^{'} \rightarrow^* e \vee e\#e^{'})$.

We omit further technical details on the model, but note that for the application of event structures as a semantic model for sequence diagrams we use *discrete* event structures. Discreteness imposes a finiteness constraint on the model, i.e., there are always only a finite number of causally related predecessors to an event, known as the *local configuration* of the event (written $\downarrow e$). A further motivation for this constraint is given by the fact that every execution has a starting point or configuration.

Event structures are enriched with a labelling function $\mu : Ev \rightarrow L$ that maps each event onto an element of the set L. This labelling function is necessary to establish a connection between the semantic model (event structure) and the syntactic model (here a sequence diagram). The labelling function used here is a partial function. Intuitively, each location marked along a lifeline of an object in a sequence diagram corresponds to one (possibly more) event(s) in the labelled event structure. The set of labels used could be the set of locations in a sequence diagram but is usually more concrete information on what the location represents: the initialisation of an object, sending/receiving a message, beginning/ending an interaction fragment, etc.

Let I denote the set of objects involved in the interaction described by sequence diagram SD, and Mes the set of asynchronous messages exchanged. Let the set of labels L be given by $L = \{(m, s), (m, r) \mid m \in Mes\}$. An event with label (m, s) corresponds to the sending of message m whereas an event with label (m, r) indicates the receipt of message m.

Definition 2 *A model* $M_{SD} = (E, \mu)$ *for a sequence diagram* SD *is obtained by composition of the models* $M_a = (E_a, \mu_a)$ *of each object instance* $a \in I$*. In* M_{SD}*, the set of events is given by* $Ev = \bigcup_{a \in I} Ev_a$*, and event labels are as before, that is,* $\mu(e) = \mu_a(e)$ *for* $e \in Ev_a$*. Let* m *be a message sent between object* a *and object* b*, and let* $E_1 \subseteq Ev_a$ *with* $\mu_a(e_1) = (m, s)$ *for all* $e_1 \in E_1$*, and* $E_2 \subseteq Ev_b$ *with* $\mu_b(e_2) = (m, r)$ *for all* $e_2 \in E_2$*. Then necessarily* $|E_1| = |E_2|$ *and for each* $e_1 \in E_1$ *there is a unique* $e_2 \in E_2$ *for each* e_1 *such that* $e_1 \rightarrow e_2$ *and local conflict* $\#_a$ *propagates over* $\rightarrow$ *to obtain conflict* $\#$ *in* M.

More details on the semantics of sequence diagrams using LES can be found in [14].

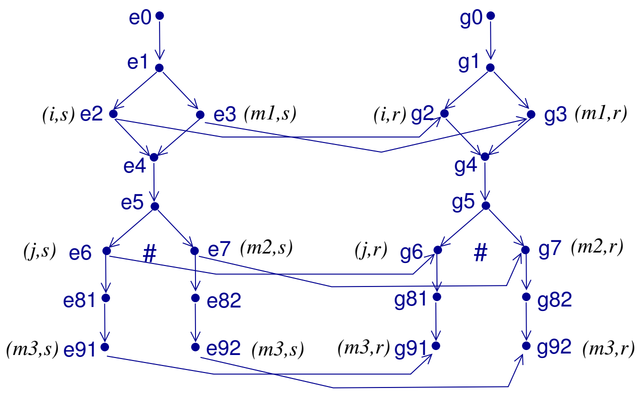

Fig. 3. Model for sequence diagram `sd1`.

The overall event structure model for the diagram from Fig. 1 is given in Fig. 3. Conflict propagation is not shown explicitly but is as expected and propagates over the new causality relations gained from communication. For example, $e_6 \#_a e_7$ and consequently $e_6 \# e_7$. In addition, since $e_7 \rightarrow g_7$ by conflict propagation we also have $e_6 \# g_7$.

Definition 3 *Let* $M_{SD} = (E, \mu)$ *be a model for sequence diagram* SD *where* $E = (Ev, \rightarrow^*, \#)$ *is an event structure. A subset of events* $C \subseteq Ev$ *is a* configuration *in* E *iff it is both 1)* conflict free*: for all* $e, e' \in C, \neg(e \# e')$ *and 2)* downwards closed*: for any* $e \in C$ *and* $e' \in Ev$*, if* $e' \rightarrow^* e$ *then* $e' \in C$*. A maximal configuration denotes a* trace.

For example, the following is a trace for Fig. 3: $C = \{e_0, e_1, e_2, e_3, e_4, e_5, e_7, e_{82}, e_{92}, g_0, g_1, g_2, g_3, g_4, g_5, g_7, g_{82}, g_{92}\}$ which denotes the occurrence of m_2 and not j.

4 Exact Metamodel Restriction

We propose a method that considers both the structure and dynamic interpretation of a sequence diagram when producing an Alloy model. The model is obtained by *exact metamodel restriction*, that is, by considering the abstract syntax of a diagram and constraints obtained from the dynamic (LES based) interpretation we generate the exact solution in Alloy that corresponds to the intended sequence diagram. This approach is also used to obtain a composed model for two (or more) sequence diagrams later on.

Alloy's syntax and semantics will be apparent in the following rules and code snippets, but we recall some main notions beforehand. Data domains are defined using signatures (keyword `sig`) and represented as sets. Just as in object-oriented languages, a signature may extend another signature, in which case the domain defined by the first is a subset of the domain of the extended signature. A signature that is declared independently of any other is called a *top-level* signature. Extensions of a signature are mutually disjoint, as are top-level signatures. A signature can also be `abstract` in which case its domain only contains elements that belong to its extending signatures. In addition, signatures may contain fields which are captured by relations. Axioms in Alloy are called `facts` which can be given a name. These must hold at any time. Alloy formulae often use the atomic predicate `in` (inclusion), standard connectives from first-order logic, and quantifiers `all` (universal) and `some` (existential). In general, expressions in Alloy are built using set theoretical relational operators and constants.

All interaction metamodel elements of Fig. 2 are transformed into top-level signatures in Alloy, and separate transformation rules treat each one. We omit the basic rules for Lifeline, Message and Event (denoting `OccurrenceSpecification`). It suffices to say that the lifeline transformation rule creates a domain called `Lifeline` as an abstract signature. Furthermore, each lifeline object has fields `name` and `class`. For each concrete instance declared in a sequence diagram we obtain declarations. The `Event` signature has a field `cover` which corresponds to a relationship with a lifeline it belongs to, and a field `next` which corresponds to a relationship with a set of events. This relationship corresponds to the *immediate* causality relation from our labelled event structures. The `Message` signature has two fields `send` and `receive` both corresponding to one event. We have additional facts to indicate the order of the events associated to a message. Messages also have a name which are introduced when creating a concrete message as shown below.

```
1 one sig sd1_i extends Message {name:one i}
2 one sig sd1_m1 extends Message {name:one m1}
3 lone sig sd1_m2 extends Message {name:one m2}
4 lone sig sd1_j extends Message {name:one j}
5 one sig sd1_m3 extends Message {name:one m3}
```

The lines above show the declaration of the messages from `sd1` (see Fig. 1 on the left). In Alloy, we cannot have two signatures with the same name. Since messages may be repeated accross different sequence diagrams we avoid this problem by adding the information from which diagram it belongs to, in this case `sd1`. Similarly for `sd2`.

```
one sig sd2_m1 extends Message {name:one m1}
one sig sd2_m2 extends Message {name:one m2}
```

Some of the messages (lines 3-4 above) are declared as `lone` (a multiplicity keyword in Alloy meaning 0 or 1), while others are `one` (exactly one). This has to do with the fact that messages within an alternative fragment are not guaranteed to occur. We will explain this in more detail later on.

```
6  lone sig e2 extends Event{}
7  lone sig e3 extends Event{}
8  lone sig e6 extends Event{}
9  lone sig e7 extends Event{}
10 lone sig e9 extends Event{}
11 lone sig g2 extends Event{}
12 lone sig g3 extends Event{}
13 lone sig g6 extends Event{}
14 lone sig g7 extends Event{}
15 lone sig g9 extends Event{}
16
17
18 //assigning events to messages
19 fact {sd1_i.send=e2 and sd1_i.receive=g2 and
20      sd1_m1.send=e3 and sd1_m1.receive=g3 and
21      sd1_j.send=e6 and sd1_j.receive=g6 and
22      sd1_m2.send=e7 and sd1_m2.receive=g7 and
23      sd1_m3.send=e9 and sd1_m3.receive=g9}
```

Lines 6-15 above declare the `sd1` events corresponding to sending/receiving a message. All events are declared as `lone` as their occurrence is dependent on whether the associated message is sent/received. For consistency, we use the same event names as used in our semantic model for the same diagram (see Fig. 3). Incidentally, we do not need to duplicate events `e9` and `g9` since Alloy will produce two solutions to represent the two possible alternative executions. In order to associate messages and events, we add a `fact` in line 19 to specify this. The following `fact EventToLifeline` connects the model events to the lifelines.

```
25 fact EventToLifeline{
26    e2.cover=L1 and g2.cover=L2 and e3.cover=L1
27     ...
28    e9.cover =L1 and g9.cover =L2  }
```

Rule 1 - Combined Fragment: A combined fragments has an interaction operator (given by `type`) and one or more interaction operands. An interaction operand covers a set of Events, CombinedFragments, or both.

```
29 abstract sig CombinedFragment{
30     operand:set InteractionOperand,type:one CF_TYPE}
31
32 abstract sig InteractionOperand
33 {cover:set Event + CombinedFragment }
34
35 fact{all e:Event| lone op:InteractionOperand |
36                e in op.cover }
37
38 fact{all cf:CombinedFragment |
39    lone op:InteractionOperand | cf in op.cover }
40
41 fact{all op:InteractionOperand |
42    one cf:CombinedFragment | op in cf.operand }
```

Lines 29-33 define the abstract signatures for combined fragments and interaction operators with the fields mentioned. Fragment nesting is given by the fact that an `InteractionOperator` may cover a `CombinedFragment`. In addition, three facts impose further constraints on the elements of these domains. Fact on line 35 states that every event `e` belongs to at most one `InteractionOperand`, and fact on line 38 states that every combined fragment `cf` belongs to at most one interaction operand (indicating fragment nesting). Finally, fact in line 41 states that all interaction operands are operands of a combined fragment.

Rule 2 - Alternative Fragment:

```
43 // alt: exactly one operand will be executed
44 fact Alt-Execution {all cf: CombinedFragment |
45  (cf.TYPE = cf_TYPE_ALT) => # cf.operand = 1}
```

In order to preserve the semantics of alternative combined fragments, the fact above states that exactly one operand is executed. Note the # in line 44 corresponds to Alloy's cardinality operator. A consequence of this fact is that every time we run the code a different set of events (associated with a particular operand) may be executed, but every time we only execute one operand of an alternative fragment.

The Alloy code lines below describe an alternative fragment with two operands and no guards, as is the case of the second combined fragment from `sd1` of Fig. 1.

```
46 one sig sd1_CF2 extends CombinedFragment{}
47 lone sig sd1_CF2_Op1 extends InteractionOperand{}
48 lone sig sd1_CF2_Op2 extends InteractionOperand{}
49 fact{all cf: sd1_CF2 | cf.TYPE = CF_TYPE_ALT }
```

At the model elements level, the first step is to define the combined fragment and its operands (lines 46-49). Notice the `lone` keyword at the beginning of the operand signatures. This is necessary as only one operand will be able to execute in accordance with the fact `Alt-Execution` (line 44). Line 48 specifies the type of `sd1_CF2` (the second combined fragment of `sd1`) as an alternative fragment.

```
50 fact OperandToCF{
51 sd1_CF2_Op1 in sd1_CF2.operand
52 sd1_CF2_Op2 in sd1_CF2.operand }
53
54 fact  EventToCF{
55 e6 in sd1_CF2_Op1.cover and g6 in sd1_CF2_Op1.cover
56 and e7 in sd1_CF2_Op2.cover and
57 g7 in sd1_CF2_Op2.cover}
```

The fact `OperandToCF` connects each operand of the second combined fragment of `sd1` to its combined fragment, while the fact `EventToCF` connects the events declared earlier belonging to this combined fragment to the corresponding operands.

Rule 3 - Parallel Fragment: The representation of a parallel combined fragment is similar to that of an alternative combined fragment, but without the fact `Alt-Execution`. The Alloy model for `sd1`, which contains a parallel combined fragment, must show a parallel execution of its operands. In other words, the events covered by different operands can occur in an arbitrary order in accordance with our LES interpretation.

To capture the notion of `GeneralOrdering` from the metamodel where it captures a binary relationship between two instances of `OcurrenceSpecification`, here events, is as follows.

Rule 4 - GeneralOrder: A GeneralOrdering represents a binary relationship between two events. This is specified in Alloy by a fact specifying the order in which all messages and their underlying events occur along the lifelines of the corresponding object instances. The transitive closure of the general ordering is irreflexive.

```
58 fact GeneralOrder {
59
60 all  l: L1 + L2, ev1:sd1_cf1.operand.cover,
61       ev2:sd1_cf2.operand.cover | ev1.cover = l
62          and ev2.cover = l => ev2 in ev1.^next
63 and
64 all  l: L1,  ev1:sd1_cf2.operand.cover,
65   ev2:e9 |   ev1.cover = l =>  ev2 in  ev1.^next
66 and
67 all  l: L2,  ev1:sd1_cf2.operand.cover,
68   ev2:g9 |   ev1.cover = l => ev2 in  ev1.^next
69 }
```

In the above fact we make use of the unary operator $^\wedge$c to denote the transitive closure of c. The `fact GeneralOrder` depicts the order of the element in the `sd1` Fig. 1. Lines 60-62 state that all events `ev1` and `ev2` such that `ev1` belongs to the first combined fragment and `ev2` belongs to the second combined fragment, if they cover the same lifeline then `ev2` belongs to the transitive closure of `ev1.next`, that is, it necessarily occurs after `ev1`. Note that `ev1` $\neq$ `ev2` since they are elements from different extensions of `CombinedFragment` and necessarily disjoint in Alloy. Lines 64-68 show that the occurrence of an event `e9` or `g9` must be preceded by the occurrence of events covered by the second combined fragment. In other words, sending/receiving message `m3` can only occur if the combined fragments have executed beforehand.

5 Semantics of Composition

We define the semantics of composition for sequence diagrams in the context of labelled event structures. We restrict ourselves to the composition of two diagrams. The case for the composition of a finite number of diagrams can be generalised from here. In the sequel, let SD_1 and SD_2 be two sequence diagrams, with sets of instances and messages given by I_1, I_2, Mes_1 and Mes_2 respectively.

When composing diagrams SD_1 and SD_2 we consider *interleaving* and *shared behaviour*. In the case of interleaving, the diagrams evolve completely autonomously of one another. That is, the *interleaving* of diagrams SD_1 and SD_2 is written $SD_1 \parallel SD_2$ and equivalent to $par(SD_1, SD_2)$. In other words, the composition is behaviourally equivalent to a diagram with a par fragment and two operands where each operand contains the behaviour described in SD_1 and SD_2 respectively.

The model for $SD_1 \parallel SD_2$, $M_{SD_1 \parallel SD_2} = (E, \mu)$, is an event structure where $Ev = Ev_1 \cup Ev_2$, all relations are preserved, and $\mu(e)$ is defined for all e iff $\mu_i(e)$ is defined for some $i \in \{1, 2\}$ in which case $\mu(e) = \mu_i(e)$. For shared instances $o \in I_1 \cap I_2$ we further match the initial events for o in Ev_1 and Ev_2. Recall that an *initial event* for an object is an event for which $\downarrow e = \{e\}$ which means that the local configuration only contains itself (cf. Section 3). We use $\downarrow Ev_o$ to indicate the singleton containing the initial event of instance o.

The composition of diagrams with *shared behaviour* is written $SD_1 \parallel_G SD_2$ where G indicates the *composition glue*. In this paper we go beyond a syntactic matching of objects and/or messages from the different diagrams. We assume that the composition glue can in addition impose restrictions on the occurrences of messages, their ordering, and so on. The case of basic syntactic matching was treated informally in [2] and we cover *behavioural* composition glue here which subsumes syntactic matching.

We define the composition of two models formally in two stages. First we define the model obtained by syntactic matching of objects and messages of both models. We then take the glue constraints and apply a restriction on the matched composed model that satisfies the glue constraints.

Let $\Delta \subseteq L_1 \times L_2 \cup I_1 \times I_2$ be a binary relation over labels or instances satisfying if $(l, l') \in \Delta$ and $(l, l'') \in \Delta$ then $l' = l''$; and if $(l', l) \in \Delta$ and $(l'', l) \in \Delta$ then $l' = l''$. We call Δ a *matching* over labels and instances. Let $\overline{Ev_1}$ (and similarly $\overline{Ev_2}$) correspond to the set of events in Ev_1 with a label not matched in Δ.

Definition 4 *Let $M_1 = (E_1, \mu_1)$ and $M_2 = (E_2, \mu_2)$ be models for sequence diagrams SD_1 and SD_2, and Δ be a matching over labels and instances. $SD_1 \parallel_\Delta SD_2$ is a* matched composition model *for Δ given by $M_\Delta = (E, \mu)$ such that events in M_Δ are given by*

$$Ev = \overline{Ev_1} \cup \overline{Ev_2} \cup$$

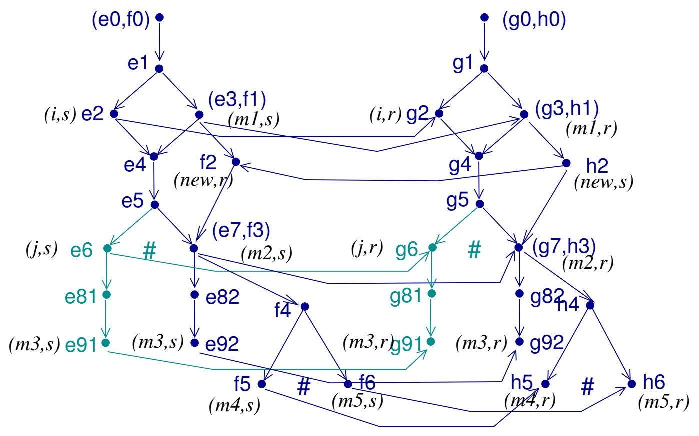

Fig. 4. Matched composition model.

$$\{(e_1,e_2)|(L(e_1),L(e_2))\in \Delta\}\cup$$
$$\{(e_1,e_2)|(e_1 \in\downarrow Ev_{i_1}, e_2 \in\downarrow Ev_{i_2} \text{ and } (i_1,i_2)\in\Delta)\}$$

The labels are unchanged, that is, $\mu(e)=\mu_i(e)$ *for* $e\in\overline{Ev_i}$ *with* $i\in\{1,2\}$ *and* $\mu(e_1,e_2)=\mu_1(e_1)=\mu_2(e_2)$. *Event relations in* M_Δ *are derived from the relations in* M_1 *and* M_2 *as follows* $(e_1,e_2)\rightarrow^* e$ *iff* $(e_1\rightarrow_1^* e$ *or* $e_2\rightarrow_2^* e)$*;* $e_i\rightarrow e_i'$ *iff* $e_i\rightarrow_i^* e_i'$*; and* $(e_1,e_2)\rightarrow^*(e_1',e_2')$ *iff* $(e_1\rightarrow_1^* e_1'$ *and* $e_2\rightarrow_2^* e_2')$*. Similarly for the conflict relation with additional conflict derived from propagation over causality.*

According to the above definition, the event pairs (e_1,e_2) in Ev correspond to events matched by Δ or denoting initial events for shared objects. Relations and labels are preserved in the composition as expected.

If the model obtained above is a valid labelled event structure then a composition for SD_1 and SD_2 according to Δ exists. Otherwise the models are not composable.

Proposition 1 *Let* $M_1=(E_1,\mu_1)$ *and* $M_2=(E_2,\mu_2)$ *be models for sequence diagrams* SD_1 *and* SD_2*, and* Δ *be a matching over instances and labels. The diagrams are* composable *according to* Δ *iff the matched composition model* $M_\Delta=(E,\mu)$ *is a well defined labelled event structure.*

A case that illustrates a non composable model is one where the same two messages (say m_1 and m_2) are sent in the reverse order in two diagrams. The model obtained by matching the respective send/receive events in both diagrams would lead to an invalid labelled event structure as the model would contain a cycle which is not allowed. We illustrate the idea of shared behaviour further with the example from Section 2 to obtain the composition of `sd1` of Fig. 1. We consider the matching of messages and lifelines with the same name, i.e., messages `m1` and `m2`, and lifelines for object `a` and object `b`. There is a matched composition model M_Δ for `sd1` and `sd2` as shown in Fig. 4. It shows the matched initial events (e.g., (e_0,f_0)) and events matched by Δ (e.g., (e_3,f_1) for label (m_1,s)). Event relations are derived from the original relations and any conflict that arises from propagation over the extended causality relation. In this case, $e_6\#(e_7, f_3)$ since $e_6\#e_7$ and consequently also $e_6\#f_4$, and so on.

We want to allow a designer to add further constraints on the expected composition by for example specifying behaviour that should never occur (forbidden events) or sequences of events that must occur in a given order, and so on. This can be seen as a way to give priority to certain specified interactions, and eliminates some of the possible traces in the composed model.

In the following, let $M_1 = (E_1, \mu_1)$ and $M_2 = (E_2, \mu_2)$ be composable models over Δ for sequence diagrams SD_1 and SD_2 with Δ a matching over labels and instances. Let $M_\Delta = (E, \mu)$ be the matched composed model obtained, and Γ be the set of maximal configurations (traces) in M_Δ.

Definition 5 *A* behavioural glue *for $M_\Delta = (E, \mu)$ is given by $G = (Ev_g, \rightarrow_g^*, \#_g, Fv_g)$ where $Ev_g, Fv_g \subseteq Ev$ are subsets of events that occur in E, and $\rightarrow_g^*, \#_g \subseteq Ev_g \times Ev_g$ are binary relations (causality and conflict) defined over the events in Ev_g. Events in Fv are forbidden events.*

A behavioural glue G as defined above may contain relations over events which disagree with the relations in M_Δ. However, we can always obtain an equivalent glue G' that preserves the relations in $M_\Delta = (E, \mu)$ by considering all the events that violate the original relations as forbidden events. We omit a formal proof here, but illustrate the idea with an example.

Definition 6 *A composed model $SD_1 \parallel_G SD_2$ for relation preserving glue G is given by $M_G = (E_G, \mu_G)$ such that it corresponds to M_Δ by removing all traces $t \in \Gamma$ such that $Fv \cap t \neq \emptyset$.*

Consider the two cases of behavioural glue as shown in Fig. 5. The behavioural glue `G1` imposes that the occurrence of message `j` is forbidden in the composed model. Glue `G2` imposes that for `m3` to occur, `m2` must have happened before. For `G1` we have $G_1 = (\emptyset, \emptyset, \emptyset, \{e_6, g_6\})$ where the events associated to message j are forbidden. This means that the composed model for `sd1` and `sd2` wrt `G1` removes all traces which contain events e_6 and g_6 from the matched composition model shown in Fig. 4. Since the events in $\downarrow e_5$ (and similarly $\downarrow g_5$) belong to another valid trace they are not removed. We obtain a composed model which is identical to the matched composition model but where the highlighted relations and events have been removed (i.e., events $e_6, e_{81}, e_{91}, g_6, g_{81}, g_{91}$ and relations).

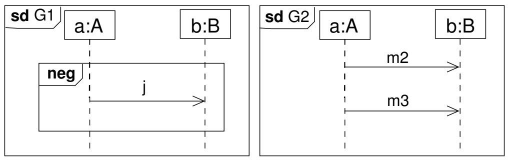

Fig. 5. Examples of behavioural glue.

For `G2` we consider an equivalent glue which preserves the relations, namely $G_2 = (Ev_{g2}, \rightarrow_{g2}^*, \emptyset, Fv_{g2})$ where $Ev_{g2} = \{(e_7, f_3), (g_7, h_3), e_{92}, g_{92}\}$, $Fv_{g2} = \{e_{91}, g_{91}\}$ and the causality relation is such that $\rightarrow_{g2}^* = \{((e_7, f_3), e_{92}), ((g_7, h_3), g_{92})\}$. In this case we need to remove all traces which contain e_{91} and g_{91} from the matched composition model shown in Figure 4. The composed model for sd1 and sd2 wrt G2 coincides with the composed model wrt G1 described earlier. This follows because the traces affected by the forbidden events are the same. We show how the model is generated automatically with Alloy in the next section.

6 Composition with Alloy

We describe how the formal composition semantics from the previous section is integrated in our SD2Alloy approach. We capture the syntactic matching of labels and instances (given by Δ in Section 5) by additional axioms (facts). The following describes the syntactic matching of labels and instances (lifelines) for our example.

```
fact LifelineMatching{
//matching lifelines from sd1 and sd2
all l1:sd1_L1, l2:sd2_L2 |
(l1.name=l2.name && l1.class=l2.class) => #l2=0
}
fact  MessageMatching{
//matching message sd1_m1 and sd2_m1
all m:sd1_m1, n: sd2_m1 |
(m.name=n.name) => #n=0 and #sd2_e3=0 and #sd2_g3=0

//matching message sd1_m2 and sd2_m2
all m:sd1_m2, n:sd2_m2 |
(m.name =n.name) => #n= 0 and #sd2_e7=0 and #sd2_g7=0
}
```

The fact LifelineMatching matches the shared lifelines in both diagrams, and the fact MessageMatching matches the messages with the same name. The idea in Alloy is that the messages and events from one of the models are kept (here sd1) and the others are hidden by limiting its occurrence to zero (i.e., its cardinality is zero).

The examples of behavioural glue introduced in Fig. 5 can be captured as facts in Alloy. G1 and G2 are given in the following facts.

```
fact Glue1{#sd1_j=0
all sd1_j_send:sd1_e6, sd1_j_receive:sd1_g6 |
 #sd1_j_send=0 and #sd1_j_receive=0}

fact Glue2{
#sd1_m3=#sd1_m2
all sd1_m2_send:sd1_e7, sd1_m3_send:sd1_e9 |
 sd1_m3_send in sd1_m2_send.^next

all sd1_m2_receive:sd1_g7, sd1_m3_receive:sd1_g9 |
 sd1_m3_receive in sd1_m2_receive.^next
 }
```

Glue1 states that j does not occur and in addition the associated events also do not occur. Glue2 states that every time m3 occurs it must occur with m2. In other words, m2 must have happened before. Again, we control occurrence with the cardinality operator #. In addition, the behavioural glue for G2 also defines the order between m3 and m2 and underlying send and receive events.

As we have seen in the previous section, the effect of each behavioural glue in the composed model is identical. This has been checked with Alloy, and message j does not occur in any solution obtained. Traces obtained with our tool have a direct correspondence with the traces of our semantic model.

7 Related work

Zhang et al. [23] and Rubin et al. [19] use Alloy for the composition of class diagrams. They transform UML class diagrams into Alloy and compose them automatically. They focus on composing static models and the composition code is produced manually. Widl et al. [21] deal with composing concurrently evolved sequence diagrams in accordance to the overall behaviour given in state machine models. They make direct use of SAT-solvers for the composition. Liang et al. [15] present a method of integrating sequence diagrams based on the formalisation of sequence diagrams as typed graphs. Both these papers focus on less complex structures. For example, they do not deal with combined fragments which can potentially cause substantial complexity.

Composition is also important in other domains such as aspect-oriented modelling. Whittle and Jayaraman [3] introduce a tool called MATA for weaving based on sequence diagrams. They put less emphasis on the semantics of the composition. Grønmo et al. [10] propose a semantics-based technique for weaving behavioural aspects into sequence diagrams. The example we use in this paper is an adaptation of the example introduced there. However, we have a true-concurrent semantics and consider and treat parallelism in interactions. In subsequent work, Grønmo et al. [9] propose the conformance issue for aspects in ensuring that the woven always leads to the same result regardless of the order in which aspects are applied. When looking at the integration of several model views or diagrams, Bowles and Bordbar [6] present a method of mapping a design consisting of class diagrams, OCL constraints and sequence diagrams into a mathematical model for detecting and analysing inconsistencies.

Checkik et al. [7] identify model integration operators, such as merge, weave, and composition, and describe each operator along with its applicability. In addition, they provide a set of desirable criteria (completeness, non-redundancy, minimality, totality, soundness) to evaluate the merge operator. This is a direction orthogonal to our research and remains an area for future investigation.

8 Conclusion

This paper presents an automated method for sequence diagram composition. The outline of the method involves the creation of logical constraints that uniquely identify each component sequence diagram as an instance of the metamodel. To combine the models, logical constraints that synchronise the two models are produced. Some of these logical constraints declare matching elements and some are to enforce behaviour involved in the composition, such as specifying behaviour that should never occur or sequences of events that must occur in a given order. This makes it possible for a designer to give priority to certain specified interactions, which is considered in the solution by eliminating unwanted traces from an initial matched model obtained.

To ensure correctness of the composition process, we have formalised the semantics of the composition with the help of labelled event structures. The result obtained automatically with Alloy preserves our formal interpretation of parallel composition with synchronisation glue. Our Alloy-based automated method of composition has been implemented as an Eclipse plugin for the composition of sequence diagrams. Throughout the paper a small example of composing sequence diagrams inspired by an example from [10] related to weaving aspects is used.

References

1. Allen, R., Garlan, D.: Formalizing architectural connection. In: ICSE 1994. pp. 71–80. IEEE Computer Society Press (1994)
2. Alwanain, M., Bordbar, B., Bowles, J.: Automated composition of sequence diagrams via alloy. In: Pires, L., Hammoudi, S., Filipe, J., das Neves, R. (eds.) MODELSWARD 2014. pp. 384–391. SciTePress (2014)
3. Araújo, J., Whittle, J.: Aspect-oriented compositions for dynamic behavior models. In: Moreira, A., Chitchyan, R., Araújo, J., Rashid, A. (eds.) Aspect-Oriented Requirements Engineering, pp. 45–60. Springer (2013)
4. Araújo, J., Whittle, J., Kim, D.: Modeling and composing scenario-based requirements with aspects. In: RE 2004. pp. 58–67. IEEE Computer Society Press (2004)
5. Berre, D.L., Parrain, A.: The SAT4j library, release 2.2 - system description. Journal on Satisfiability, Boolean Modeling and Computation 7, 59–64 (2010)
6. Bowles, J., Bordbar, B.: A formal model for integrating multiple views. In: ACSD 2007. pp. 71–79. IEEE Computer Society Press (2007)
7. Chechik, M., Nejati, S., Sabetzadeh, M.: A relationship-based approach to model integration. Innovations in Systems and Software Engineering 8(1), 3–18 (2012)
8. Fiadeiro, J., Lopes, A., Wermelinger, M.: A mathematical semantics for architectural connectors. In: Backhouse, R., Gibbons, J. (eds.) Generic Programming, LNCS, vol. 2793, pp. 178–221. Springer (2003)
9. Grønmo, R., Runde, R., Møller-Pedersen, B.: Confluence of aspects for sequence diagrams. Software & Systems Modeling 12(4), 789–824 (2013)
10. Grønmo, R., Sørensen, F., Møller-Pedersen, B., Krogdahl, S.: Semantics-based weaving of uml sequence diagrams. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008, LNCS, vol. 5063, pp. 122–136. Springer (2008)
11. Harel, D., Marelly, R.: Come, Let's Play: Scenario-based Programming Using LSCs and the Play-Engine. Springer (2003)
12. Jackson, D.: Software Abstractions: logic, language and analysis. MIT Press (2006)
13. Klein, J., Hélouët, L., Jézéquel, J.: Semantic-based weaving of scenarios. In: AOSD'06. pp. 27–38. ACM (2006)
14. Küster-Filipe, J.: Modelling concurrent interactions. Theoretical Computer Science 351, 203–220 (2006)
15. Liang, H., Diskin, Z., Dingel, J., Posse, E.: A general approach for scenario integration. In: MoDELS 2008. pp. 204–218. LNCS 5301, Springer (2008)
16. Micskei, Z., Waeselynck, H.: The many meanings of UML 2 sequence diagrams: a survey. Software and Systems Modeling 10, 489–514 (2011)
17. OMG: UML: Superstructure. Version 2.4.1. OMG, http://www.omg.org. (2011), http://www.omg.org, document id: formal/2011-08-06. [accessed 1-6-2012]
18. R.Reddy, Solberg, A., R.France, Ghosh, S.: Composing sequence models using tags. In: Proc. of MoDELS Workshop on Aspect Oriented Modeling (2006)

19. Rubin, J., Chechik, M., Easterbrook, S.: Declarative approach for model composition. In: MiSE 2008. pp. 7–14. ACM (2008)
20. Whittle, J., Araújo, J., Moreira, A.: Composing aspect models with graph transformations. In: Proceedings of the 2006 international workshop on Early aspects at ICSE. pp. 59–65. ACM (2006)
21. Widl, M., Biere, A., Brosch, P., Egly, U., Heule, M., Kappel, G., Seidl, M., Tompits, H.: Guided merging of sequence diagrams. In: SLE 2012. pp. 164–183. LNCS 7745, Springer (2013)
22. Winskel, G., Nielsen, M.: Models for Concurrency. In: Abramsky, S., Gabbay, D., Maibaum, T. (eds.) Handbook of Logic in Computer Science, Vol. 4, Semantic Modelling, pp. 1–148. Oxford Science Publications (1995)
23. Zhang, D., Li, S., Liu, X.: An approach for model composition and verification. In: NCM 2009. pp. 1102–1107. IEEE Computer Society Press. (2009)