Matching and Merging Scenarios Automatically with Alloy

J. Bowles[1], M. Alwanain[2], B. Bordbar[2], and Y. Chen[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, UK
{m.i.alwanain|b.bordbar|y.chen}@cs.bham.ac.uk

Abstract. The design of large systems often involves the creation of models that describe partial specifications. Model composition is the process of combining partial models to create a single coherent model. This paper presents an automatic composition technique for creating a sequence diagram from partial specifications captured in multiple sequence diagrams with the help of Alloy. Our contribution is twofold: a novel true-concurrent semantics for sequence diagram composition, and a model-driven transformation of sequence diagrams to Alloy that preserves the semantics of composition defined. We have created a tool *SD2Alloy* that implements the technique as follows: two given sequence diagrams are transformed into two Alloy models, and merged according to a set of syntactic logical constraints describing how their elements should be matched. These constraints are in accordance to our compositional semantics. The technique can also be used to detect problems and inconsistencies in the composition of diagrams.

1 Introduction

The process of developing modern systems is gradually becoming more and more complex. Due to the increase in the complexity of such software development processes, we often make use of multiple models for expressing various scenarios and viewpoints. To reduce the complexity of the design, models of the system are usually broken into partial specifications. For example, behaviour related to the interaction between parts can be captured by different sequence diagrams. However, integrating these diagrams into one to describe the whole behaviour requires *model composition* techniques. Manual model composition is error-prone, time-consuming and tedious [1]. In recent years, automated model composition has received considerable attention [2, 3]. For example [2] make use of Alloy for automated composition. Nonetheless, most automated merging methods only focus on static models.

In this paper we focus on the automated integration of sequence diagrams, one of UML's most popular behavioural models [4]. In particular, we focus on the composition of sequence diagrams with the help of Alloy. Our contribution is twofold: a novel true-concurrent semantics for sequence diagram composition, and a model-driven transformation of sequence diagrams to Alloy that preserves the semantics of composition.

Our automated technique follows three main steps. In the first step, multiple sequence diagrams are automatically transformed into Alloy models. For each sequence diagram a unique Alloy model is produced which if solved has as many solutions as there are possible traces of execution in the original sequence diagram. These traces have a direct correspondence to the ones obtained in the underlying semantics of sequence diagrams used, namely labelled event structures (LES) [5, 6]. In the second step, the Alloy models are merged to produce a single Alloy model, which contains elements from the individual Alloy models of each sequence diagram in addition to syntactic logical constraints specifying how the elements are matched and the diagrams should be composed. The logical constraints used for the matching are syntactic and considered in our true-concurrent semantics of composition. In the third step, we use the composed model obtained, that is the conjunction of the overall logical constraints, to formally check if the sequence diagrams can be composed and obtain the composition of the diagrams automatically. These steps are fully automated in our tool *SD2Alloy* which was implemented using Model Driven Architecture (MDA) techniques [7]. Later in the paper, we justify further our choice of Alloy as a target language.

The remainder of the paper is structured as follows: Section 2 gives a general background of sequence diagrams, their formalisation with event structures and Alloy. Section 3 addresses model composition syntactically (at the UML level) and semantically (over labelled event structures) which guides the model transformation from sequence diagrams onto Alloy as discussed in Section 4. Section 5 describes model composition via Alloy, and Section 6 gives some details of our tool. Finally, Section 7 describes related work and Section 8 concludes the paper.

2 Background

2.1 Sequence Diagrams

UML sequence diagrams capture scenarios of execution as object (or in some cases component) interactions. 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 [8]) 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.

A *message* is a synchronous or asynchronous communication between two objects shown as an arrow connecting the respective lifelines, that is, the underlying send and receive events of the message. We only consider synchronous communication in this paper, even though both forms of communication can be addressed in our approach. 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 [4]. Depending on the operator used, an interaction fragment consists of one or more *operands*. In the case of the **alt** fragment, each operand describes a choice of behaviour. Only one of the alternative operands is executed if the guard expression (if present) evaluates to true. If more than one operand

has a guard that evaluates to true, one of the operands is selected nondeterministically for execution. In the case of the **par** fragment, there is a parallel merge between the behaviours of the operands. The event occurrences of the different operands can be interleaved in any way as long as the ordering imposed by each operand as such is preserved.

Finally, interaction fragments can be nested producing expressive and complex scenarios of execution. One simple example illustrating the concepts above and with a parallel nested within an alternative fragment is given in Fig. 1. In this case, all mes-

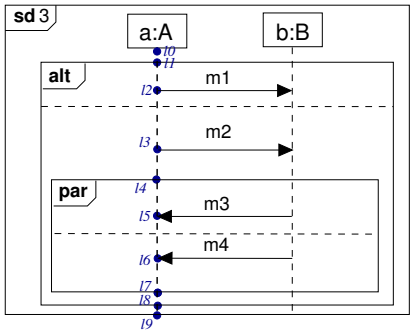

Fig. 1. A sequence diagram with nested fragments.

sages (from m_1 to m_4) are sent synchronously between objects *a* and *b*. The locations along the lifeline of object *a* are shown explicitly. The importance of locations as well as the effect produced through the nesting of fragments (i.e., the possible traces of execution) are described in the next subsection. 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.

2.2 Formal Model

Several possible semantics for sequence diagrams have been defined (see [9] for an overview). In this paper we use the semantics defined in [6] which introduces a very simple and intuitive behavioural model to capture interactions, and is the only true-concurrent semantics available for sequence diagrams.

Prime event structures [5], 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.

Formally, 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. 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 (usually a total 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).

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.

Consider the locations marked on Fig. 1 for object a. The events in the model shown in Fig. 2 have a direct correspondence to the locations of object a.

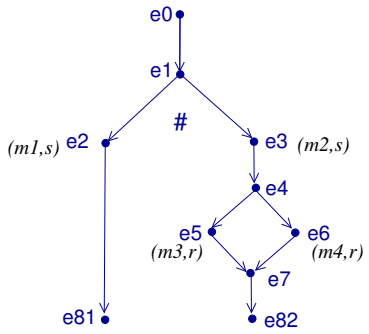

Fig. 2. Event structure for object a of Fig. 1.

The graphical representation of the event structure E_a shows immediate causality between events (e.g., $e_0 \rightarrow e_1$) and direct conflict (e.g., $e_2 \# e_3$). By conflict propagation we also have $e_2 \# e_4$, etc. Unrelated events are concurrent (e.g., $e_5 \; co \; e_6$). Intuitively, events e_1 and e_4 denote the beginning of the alternative and parallel fragments respectively. Consequently events e_5 (denoting the receipt of message m_3) and e_6 (denoting the receipt of message m_4) are concurrent. Events e_{81} and e_{82} both correspond to location l_8 denoting the end of the alternative fragment. These events must be in conflict because they represent different ways to reach the location. Note that there cannot be

one end event in this case, because conflict propagates over causality and it would lead to an event in conflict with itself and hence an invalid event structure (conflict is irreflexive). Some event labels are given where (m_1, s) denotes sending message m_1, and (m_3, r) denotes receiving message m_3.

Let I denote the set of objects involved in the interaction described by sequence diagram SD. A model $M_{SD} = (E, \mu)$ for a sequence diagram SD is obtained by composition of the models $M_i = (E_i, \mu_i)$ of each object instance $i \in I$. In the composed model, the set of events Ev is such that $e \in Ev$ iff there is an object $i \in I$ such that $e \in Ev_i$, or $(e_1, e_2) \in Ev$ iff there are two objects $i \neq j \in I$ with $e_1 \in Ev_i$, $e_2 \in Ev_j$, $\mu_i(e_1) = (m, s)$ and $\mu_j(e_2) = (m, r)$. In other words, shared events (e_1, e_2) correspond to message synchronisation. To keep it simple, we assume that $\mu : Ev \rightarrow Mes$ is a partial function defined over shared events only and indicating the message exchanged. I.e., $\mu(e_1, e_2) = m$ iff $\mu_i(e_1) = (m, s)$ and $\mu_j(e_2) = (m, r)$ for some $i, j \in I$. More details on the semantics of sequence diagrams using event structures can be found in [6].

2.3 Alloy

Alloy [10] is a declarative textual modelling language based on first-order relational logic. An Alloy model consists of a number of signature declarations, fields, facts and predicates. Furthermore, each signature denotes a set of atoms, which are the basic entities of Alloy. Alloy is supported by a fully automated constraint solver called Alloy Analyzer, which permits 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 [11] embedded within the Alloy Analyzer. The Alloy Analyzer has been used in various applications including the composition of static models [2].

In this paper, Alloy is used as part of an automated tool to compose sequence diagrams. The composition is based on a set of logical constraints which we designate *merging glue*. Alloy is a language for describing the *structural* information underlying a design model whereas labelled event structures are needed to make sure the semantics of the *behavioural* model and the composition are as expected.

The choice of Alloy as a target framework makes it straightforward to find a model (if available) for the composition of sequence diagrams. The approach converts each sequence diagram into a set of logical constraints to which it is simple to add additional constraints capturing the *merging glue*. Alloy solves these constraints to *find* a model that complies to both sequence diagrams and the glue.

3 Model Composition

For the integration of two or more scenarios we define syntactic composition of sequence diagrams and its underlying semantics.

Our mechanism for composition of sequence diagrams considers interleaving of diagrams and shared behaviour. In the first case, diagrams evolve completely autonomously

whereas in the latter case diagrams have shared behaviour (shared objects and messages). We treat the cases separately and consider only the composition of two diagrams. The case for an arbitrary number of diagrams is easily 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.

The *interleaving* of diagrams SD_1 and SD_2 with $Mes_1 \cap Mes_2 = \emptyset$ is written $SD_1 \parallel SD_2$ and is defined syntactically as $par(SD_1, SD_2)$. In other words, it consists of a diagram with a par fragment and two operands where each operand contains the behaviour described in SD_1 and SD_2 respectively.

Semantically, the model for $SD_1 \parallel SD_2$ is an event structure $M_{SD_1 \parallel SD_2} = (E, \mu)$ 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 and maximal events in Ev_1 and Ev_2. We illustrate this with an example (see Fig. 3) showing shared objects but different messages.

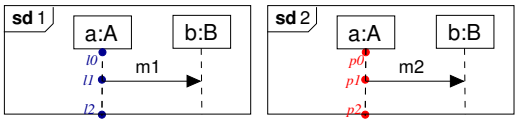

Fig. 3. Two simple sequence diagrams.

The models associated to SD_1 and SD_2 are given in Fig. 4.

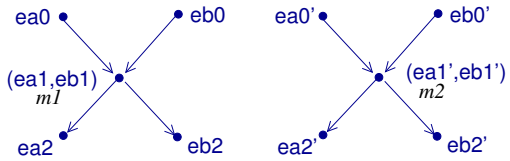

Fig. 4. Model for SD_1 (left) and SD_2 (right).

As described above, if we compose both models we can merge initial and maximal events for shared objects which in this case corresponds to events e_{a0} and $e_{a0}{}'$, e_{b0} and $e_{b0}{}'$, e_{a2} and $e_{a2}{}'$, and e_{b2} and $e_{b2}{}'$. The final composition $SD_1 \parallel SD_2$ is shown in Fig. 5. This is the exact model obtained for a sequence diagram which consists of a parallel fragment with two operands where the first operand is taken from $SD1$ and the second operand is taken from SD_2.

The composition of diagrams SD_1 and SD_2 with *shared behaviour* is written $SD_1 \parallel_G SD_2$ where $G = Mes_1 \cap Mes_2$ indicates the shared behaviour.

If $G = Mes_1$, in other words, all the behaviour in SD_1 is shared, then we say that SD_1 is *syntactically contained* in SD_2, and the composition $SD_1 \parallel_G SD_2$ can be reduced to SD_2.

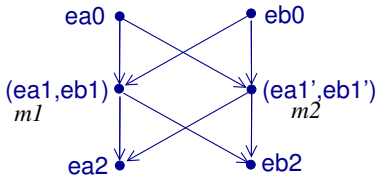

Fig. 5. Model for $SD_1 \parallel SD_2$.

We now consider the case that $G = \{m\}$. This case can be generalised to a finite number of messages, but we omit it here for simplicity.

Consider $SD_1 = seq(\varphi_0, m, \varphi_1)$ and $SD_2 = seq(\varphi_0{}', m, \varphi_1{}')$ where seq denotes a sequential fragment, φ_0, φ_1, $\varphi_0{}'$ and $\varphi_1{}'$ are interactions which on their own would define a valid sequence diagram and may be empty. The composition $SD_1 \parallel_G SD_2$ is defined syntactically by $seq(par(\varphi_0, \varphi_0{}'), m, par(\varphi_1, \varphi_1{}'))$.

Note that the seq fragment describes the default (sequential) behaviour of a sequence diagram and can be omitted in a diagram, but is useful here to describe composition in general. For example, SD_1 from Fig. 3 can be seen as $seq(\varphi_0, m_1, \varphi_1)$ with φ_0 and φ_1 both empty.

Consider a more complex case where $SD_1 = f(seq(\varphi_0, m, \varphi_1), \varphi_2)$ and $SD_2 = seq(\varphi_0{}', m, \varphi_1{}')$ where f denotes an arbitrary fragment (e.g., par, alt, etc). The composition $SD_1 \parallel_G SD_2$ is defined syntactically by:

$$f(seq(par(\varphi_0, \varphi_0{}'), m, par(\varphi_1, \varphi_1{}')), \varphi_2)$$

In other words, if the shared behaviour is contained in an arbitrary fragment, then this fragment is preserved in the composed behaviour.

Consider the sequence diagrams SD_1 and SD_2 given in Fig. 6 which share message m_2.

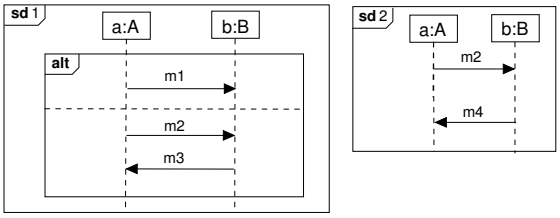

Fig. 6. Two sequence diagrams with shared message m_2.

The sequence diagrams can be seen as $SD_1 = alt(\varphi_0, seq(\emptyset, m2, \varphi_1))$ and $SD_2 = seq(\emptyset, m2, \varphi_1{}'))$, with φ_0 corresponding to a simple interaction with m_1, and similarly for φ_1 and message m_3, and $\varphi_1{}'$ and message m_4. The composition $SD_1 \parallel_G SD_2$ as outlined above is given by $alt(\varphi_0, seq(\emptyset, m_2, par(\varphi_1, \varphi_1{}')))$. The composed diagram is our first sequence diagram from Fig. 1.

Given the syntactic composition of two sequence diagrams we derive the model (a labelled event structure) as described before.

4 Model Transformation to Alloy

We implement our composition method with the help of MDA techniques [7]. Due to space restrictions, in this paper we only discuss some of the transformation rules from sequence diagrams to Alloy. These rules can be implemented via any MDA transformation engine. Our approach is such that if an Alloy model can be solved, it generates all possible solutions each of which corresponds to a run of the original sequence diagram and in accordance to the formal semantics defined in the previous sections. The following transformation rules illustrate the transformation for `sd1` from Fig. 6.

4.1 Lifeline and message

Each lifeline in a sequence diagram, which corresponds to an object with a name and of a given class (type), is transformed into Alloy code as follows.

```
1 abstract sig Lifeline {}
2 one sig A {}  //Lifeline Class
3 one sig a {} //Lifeline name
4 one sig sd1_L_1 extends Lifeline {name: a, type: A }
5 one sig sd1_L_2 extends Lifeline {name: b, type: B }
```

A `Lifeline` corresponds to an abstract signature refined further by concrete lifelines from a sequence diagram. The code above shows the transformation of `sd1` lifelines in Alloy. Lines 4 and 5 give concrete lifeline declarations `sd1_L_1` and `sd1_L_2`. The keyword `one` in the declaration indicates that there is exactly one instance of the signature. Furthermore, a lifeline signature has fields `name` to specify the object name and `type` to specify its class.

A message has two message ends, a send and a receive events, which cover a lifeline. A receive event cannot occur unless its corresponding send event has happened before. An event is either a send or a receive event.

```
7  abstract sig Event { cover :one Lifeline , next :set Event }
8  abstract sig Message { send :one Event , receive :one Event }
9  //a message send event always occurs before the associated receive event
10 fact MessageEventsOrder {all m: Message |
11         m.receive in m.send.next }
12 // all events correspond to send or receive events of one message
13 fact {all e: Event | one m: Message |
14         e = m.send or e = m.receive }
```

The rule above creates the domains Event and Message. In both cases these are abstract signatures with two fields. The `Event` signature has a field `cover` corresponding to a relationship with the lifeline it belongs to, and a field `next` denoting 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. The facts on lines 10-14 describe two constraints over the elements in the domain as mentioned before and are straightforward.

A message also has a name which is introduced when creating a concrete message.

```
15 lone sig sd1_m1 extends Message { name : m1}
16 lone sig sd1_m2 extends Message { name : m2}
17 lone sig sd1_m3 extends Message { name : m3}
```

In diagram `sd1` of Fig. 6 we have three messages `m1`, `m2` and `m3`. The lines above show the declaration of these messages. The messages are declared as `lone` (a multiplicity keyword in Alloy meaning 0 or 1). 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 in the transformation rule for the alternative fragment.

```
18 lone sig sd1_e1 extends Event {}
19 lone sig sd1_e2 extends Event {}
20 lone sig sd1_e3 extends Event {}
21 lone sig sd1_e4 extends Event {}
22 lone sig sd1_e5 extends Event {}
23 lone sig sd1_e6 extends Event {}
24
25  // assigning events to messages
26 fact { sd1_m1.send = sd1_e1 and sd1_m1.receive = sd1_e2 and
27 sd1_m2.send = sd1_e3 and sd1_m2.receive = sd1_e4 and
28 sd1_m3.send = sd1_e5 and sd1_m3.receive = sd1_e6 }
29
30  // assigning events to lifelines
31 fact EventToLifeline {
32 sd1_e1. cover =sd1_L_1 and sd1_e2. cover =sd1_L_2
33 ...
34 sd1_e5. cover =sd1_L_1 and sd1_g6. cover =sd1_L_2 }
```

Six events are declared in lines 18-23 above. Events are always associated to the sending or receiving of messages. How these events are associated to the messages declared in lines 15-17 is given in the fact of lines 26-28, and which lifeline they cover is given in the fact of lines 31-34. All events are declared as lone as the corresponding messages fall within the scope of an alternative fragment and may therefore not occur.

In the case of sequential messages without interaction fragments, or messages within the same operand (e.g., `m2` and `m3`), this implies a total order among the events of the lifeline of an object. This is specified in Alloy by another logical constraint called `GeneralOrder` shown below for the events underlying messages `m2` and `m3`.

```
36  fact GeneralOrder {
37    sd1_e6 in sd1_e3.^next and sd1_e5 in sd1_e4.^next
38    }
```

The fact above specifies the expected ordering between events `e3` (`m2.send`) and `e6` (`m3.receive`), and between `e4` (`m2.receive`) and `e5` (`m3.send`) . No statement is made about the relation between these events and those underlying `m1` because they belong to a different operand and are hence not related by causality.

4.2 Alternative Combined Fragment

For the alternative interaction fragment (aka combined fragment in UML's metamodel [4]), the transformation generates a set of abstract signatures as follows.

```
39 abstract sig Combinedfragment {
40           operand:set Operand}
41 abstract sig Operand{cover:set Event+Combinedfragment}
42
43 fact {all e: Event | lone op: Operand |
```

```
44  e in op. cover }
45
46 fact {all cf: Combinedfragment |
47  lone op: Operand | cf in op. cover }
48
49 fact {all op: Operand |
50  one cf: Combinedfragment | op in cf. operand }
51
52 // alt: exactly one operand will be executed
53 fact Alt-Execution {all CF: Combinedfragment |
54  ( CF.TYPE = CF_TYPE_ALT) => # CF.operand = 1}
```

These abstract signatures represent the main elements of combined fragments and how interactions are defined at the metamodel level in UML[4]. The abstract signature for `CombinedFragment` consists of one or more operands whereby operands contain events (i.e., cover the send and receive events of the messages defined inside it) and/or combined fragments nested within the operand. In addition, three facts impose further constraints on the elements of these domains. Fact on line 43 states that every event `e` belongs to at most one operand, and fact on line 46 states that every combined fragment `cf` belongs to at most one interaction operand (indicating fragment nesting). Fact in line 49 states that all interaction operands are operands of only one combined fragment. Finally, the `fact` in line 53 defines that at most one operand is executed. This implies that a different set of events may occur for each possible run of the code preserving the semantics of alternative combined fragments.

```
56 one sig CF_TYPE_ALT {}//Combinedfragment Type
57 one sig sd1_CF extends Combinedfragment{TYPE = CF_TYPE_ALT}
58 lone sig sd1_Operand_1 extends Operand{}
59 lone sig sd1_Operand_2 extends Operand{}
60
61 // Covering: Combined Fragment->Operands
62 fact{
63   sd1_Operand_1 in CF.operand
64   sd1_Operand_2 in CF.operand}
65 // Connect events to Operands
66 fact EventToOp{
67 sd1_e1 in sd1_Operand_1.cover and sd1_e2 in sd1_Operand_1.cover and
68 sd1_e3 in sd1_Operand_2.cover and sd1_e4 in sd1_Operand_2.cover and
69 sd1_e5 in sd1_Operand_2.cover and sd1_e6 in sd1_Operand_2.cover}
```

In line 56, the signature `CF_TYPE_ALT` declares the type of the combined fragment, in this case an `ALT`. Following this, in lines 57-59, three signatures define the combined fragment and the number of operands used, in this case `Operand_1` and `Operand_2`. The operands (lines 58-59) are declared as `lone` which allows the previous fact in line 53 to execute only one operand. Moreover, the facts in line 62 and 66 establish a connection between the combined fragment and its operands, and between the operands and their events, respectively.

4.3 Parallel Combined Fragment

In Alloy the representation of a parallel combined fragment (not present in Fig. 6) is similar to that of an alternative combined fragment, but without fact `Alt-Execution`. The following is an example.

```
1 one sig CF_TYPE_PAR {}//Combinedfragment Type
2 one sig sd1_CF extends Combinedfragment{
```

```
3     TYPE = CF_TYPE_PAR}
4  one sig Operand_1 extends Operand{}
5  one sig Operand_2 extends Operand{}
6  // Covering: Combined Fragment->Operands
7  fact{
8    Operand_1 in CF.operand
9    Operand_2 in CF.operand}
10   // Connect events to Operands
11 fact EventToOp{
12 sd1_e1 in sd1_Operand_1.cover and sd1_e2 in sd1_Operand_1.cover and
13 .........
14 sd1_e5 in sd1_Operand_2.cover and sd1_e6 in sd1_Operand_2.cover}
15
16 fact{all CF: Combinedfragment, OP1:  CF.operand,  OP2: CF.operand,
17    E1: OP1.cover, E2: OP2.cover, E3: OP1.cover | no  E4: OP2.cover | OP1 != OP2
18    and E2 in E1.next and E3 in E4.next  }
```

All operands of a parallel fragment are declared as `one` since they are necessarily executed. The Alloy model that contains a parallel combined fragment must show a parallel execution of `Operand_1` and `Operand_2`, i.e., the events covered by each operand are not related and can thus occur in an arbitrary order. This is given in the fact of line 16, and is in accordance to the labelled event structure semantics given earlier. It implies a relation of concurrency between events in different operands whilst the events within an operand remain ordered in the usual way. Therefore, this fact guarantees the preservation of the correct and intended order of events in a parallel fragment.

5 Composition via Alloy

In order to compose Alloy models that have been obtained by transformation from sequence diagrams, two fundamental conditions must be satisfied:

- *Matching* elements must indicate correspondence between equivalent elements of the source. The purpose of matching is to uncover how two models correspond to each other.
- *Merging* of equivalent elements identified earlier producing a composed version of the models.

In Alloy, these conditions can be encoded by adding facts that must be satisfied to match and merge equivalent elements. For example, consider two Alloy models `A1` and `A2` each with two lifelines, where these lifelines have the same name and type. In order to compose the lifelines with the same name from each one of the models we have to specify the following fact.

```
fact lifelineEquality {
all L1: A1_Lifeline_1 , L2: A2_lifeline_1 |
(L1.type=L2.type && L1.name=L2.name) =>#  L2 =0}
```

The Alloy code above shows that if the matching condition is satisfied, then lifelines will be merged into one given by `L1` and `L2` will be hidden. The same is true of messages. For example, if the two Alloy models `A1` and `A2` have two messages with the same name and involving the same objects (lifelines) then Alloy will compose these messages into one.

The idea of the procedure of merging entered models in Alloy is as follows. First we generate a new Alloy model `A3` representing the result of merging the original models. Second, we copy all the elements of `A1` to `A3`. Third, we copy all elements of `A2` except the duplication elements such as abstract signatures that are shared in the two models. Fourth, for any pair of equal elements, one of the signatures keyword has to be changed from `one` to `lone` to be able to merge it and then add the merging facts mentioned above. Finally, in terms of merging messages, the merged message events (send and receive) are replaced with their equivalent message events to apply the behaviour environment of both models into this message.

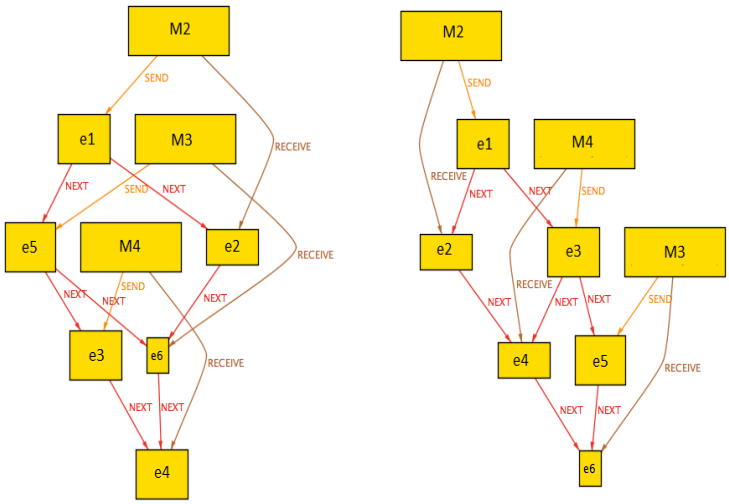

Fig. 7. Alloy instance obtained from merging sd1 and sd2 of Fig. 6.

To validate our approach, we implemented the example of Fig. 6 in Alloy. After solving the merged model, we obtained three Alloy solutions (also referred to as instances). These instances show exactly the expected behaviour underlying Fig. 1 with possible traces of execution: only $m1$ occurs, or $m2 \cdot (m3\ co\ m4)$ occur. Fig. 7 shows two Alloy instances, one for each of the possible executions of the second operand of the alternative fragment. These instances show in particular that m_2 is always before $m3$ and $m4$, and $m3$ and $m4$ are in parallel. The complete Alloy code for the example used as well as the composed model is available from our research webpage[3].

6 The *SD2Alloy* Tool

Our approach relies on Model Driven Development (MDD) techniques that aim to enhance the role of modelling in software development [12]. It allows the developer to

[3] http://www.cs.bham.ac.uk/∼bxb/research/matching-merging/alloy-example

model the required functionality and the overall architecture of the system instead of calling on developers to spell out every detail of the system's implementation using a programming language. Hence, MDD results in reduced development cycles and lower cost of software production.

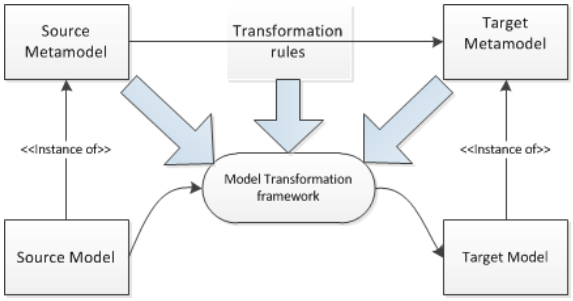

Fig. 8. An overview of MDA.

To ensure that the methods developed can be adopted by the software industry, it is crucial to follow standards set by the Model Driven Architecture (MDA) framework [7]. MDA is a framework for software development proposed by the Object Management Group (OMG). It provides a set of guidelines for the structuring of models and their specifications. It also defines a standard for application design and implementation. Central to MDA is the notion of metamodels [13]. A metamodel defines all elements that are available for a designer to use when modelling with a language. In MDA, a model transformation is defined by mapping the meta-elements, i.e., the constructs of the source metamodel (e.g., sequence diagrams) are mapped onto constructs in the target metamodel (e.g., Alloy) as Fig. 8 shows. Subsequently, every model arising from the source metamodel can be transformed automatically to an instance of the destination metamodel with the help of a model transformation framework such as SiTra [14].

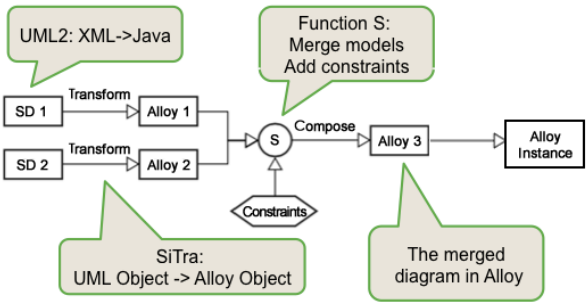

Fig. 9. The SD2Alloy architecture.

We now give a brief description of our composition tool *SD2Alloy* which was built in accordance to MDA. The transformation rules have been implemented as an Eclipse plugin. Fig. 9 depicts the *SD2Alloy* architecture. The tool includes a modified open source tool called Papyrus [16], which allows the user to generate any number of sequence diagrams, and exports these diagrams as XMI files so they can be parsed. *SD2Alloy* parses the XMI files generated by Papyrus into Java objects using the UML2 library. SiTra is used to transform the UML sequence diagrams and create the Alloy Java object that produces the Alloy code.

In Fig. 9, two sequence diagrams SD1 and SD2 are transformed individually to Alloy generating Alloy 1 and Alloy 2 respectively. Moreover, this tool allows the user to specify composition constraints required in Alloy to merge the entered models, shown as S and denoting the syntactic matching of model elements. The composition model obtained is shown as Alloy 3 which corresponds to the union of all constraints associated to the individual models Alloy 1, Alloy 2 and the glue contraints in S. If there are no conflicts in Alloy 3 then the Alloy Analyzer produces an Alloy instance (in fact as many as there are possible traces of execution in Alloy 3). If no solution can be produced, Alloy highlights the constraints that are causing a conflict. More details on the tool can be found in [15].

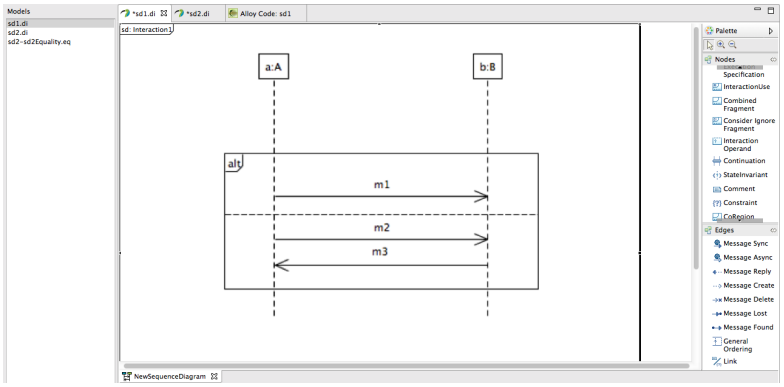

Fig. 10. A snapshot of the SD2Alloy interface.

Fig. 10 and Fig. 11 show two snapshots of the interface of *SD2Alloy*. In both cases, to the left we have a list of the current sequence diagrams used (here sd1.di and sd2.di where di is the extension name given by Papyrus) as well as the syntactic matching declarations of model elements from the different diagrams (here sd1-sd2Equality.eq). Different levels of detail can be shown on different panes in the tool with the editor in the middle showing the current diagram or code that is being edited. For example, in Fig. 10 we show a diagram and in Fig. 11 we look at the generated Alloy code for the same diagram. Properties of elements being edited can also be seen and changed on a separate pane at the bottom right of the tool.

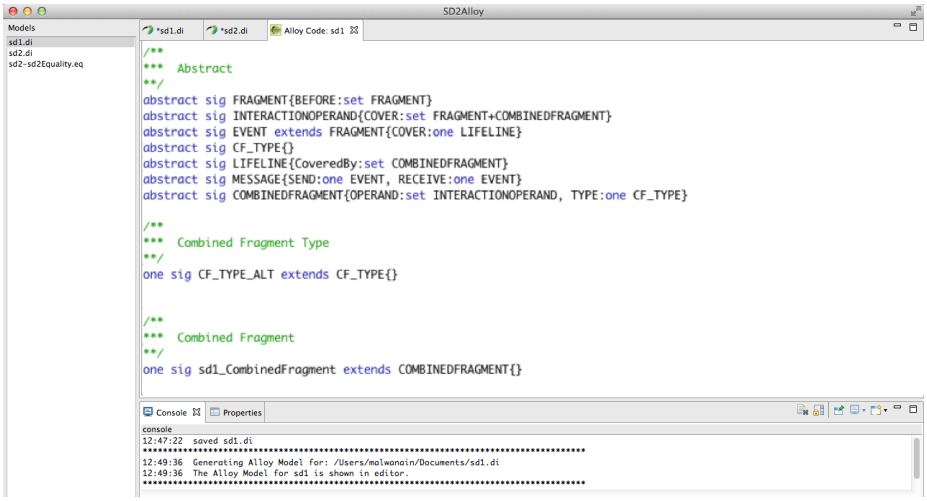

Fig. 11. Showing Alloy code in SD2Alloy.

7 Related work

Over the last decade, a number of software tools and algorithms have been designed and implemented to compose behavioural models. Liang et al. [17], have presented a method of integrating sequence diagrams based on the formalisation of sequence diagrams as typed graphs. Rubin et al. [2], illustrate the use of the Alloy Analyzer to compose class diagrams based on syntactic properties of metamodels and the primary model. This approach uses UML2Alloy [18] to transform UML class diagrams into Alloy and the Alloy Analyzer to compose these classes. However, their method only composes static models and the compositional code produced is generated manually.

In addition, Widl et al. [3] present an approach for composing concurrently evolved sequence diagrams in accordance to the behaviour given in state machine models. They describe the problem of merging sequence diagrams formally using SAT solvers. However, similarly to [17], the approach does not merge complex sequence diagrams.

When looking at the integration of several model views or diagrams, [19] presents a method of mapping a design consisting of class diagrams, OCL constraints and sequence diagrams into a mathematical model for detecting and analysing inconsistencies. Finally, [20] propose a further approach to composition of sequence diagrams by composing sequence diagram operators directly. This approach is very different from ours and can be seen as a high-level composition strategy at the UML level.

8 Conclusions

In this paper, we have defined a new compositional semantics of sequence diagrams based on the true-concurrent model of labelled event structures, and presented an automated technique based on Alloy to generate a composed model in accordance to the true-concurrent semantics.

The underlying developed tool takes as input one or more sequence diagrams, and automatically constructs Alloy solutions for the composition. Each of the solutions corresponds to a run that can be derived from the underlying labelled event structure of the composed sequence diagram. Our approach has been evaluated through a series of examples and larger case studies.

References

1. Rosa, M.L., Dumas, M., Uba, R., Dijkman, R.: Merging business process models. On the Move to Meaningful Internet Systems: OTM 2010 (2010) 96–113
2. Rubin, J., Chechik, M., Easterbrook, S.: Declarative approach for model composition. In: MiSE'08, ACM (2008) 7–14
3. Widl, M., Biere, A., Brosch, P., Egly, U., Heule, M., Kappel, G., Seidl, M., Tompits, H.: Guided merging of sequence diagrams. In: SLE 2012. LNCS 7745, Springer (2013) 164–183
4. OMG: UML: Superstructure. Version 2.4.1. OMG, http://www.omg.org. (2011) Document id: formal/2011-08-06. [accessed 1-6-2012].
5. 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. Oxford Science Publications (1995) 1–148
6. Küster-Filipe, J.: Modelling concurrent interactions. Theoretical Computer Science **351** (2006) 203–220
7. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The model driven architecture: practice and promise. Addison-Wesley (2003)
8. Harel, D., Marelly, R.: Come, Let's Play: Scenario-based Programming Using LSCs and the Play-Engine. Springer (2003)
9. Micskei, Z., Waeselynck, H.: The many meanings of UML 2 sequence diagrams: a survey. Software and Systems Modeling **10** (2011) 489–514
10. Jackson, D.: Software Abstractions: logic, language and analysis. MIT Press (2006)
11. Berre, D.L., Parrain, A.: The SAT4j library, release 2.2 - system description. Journal on Satisfiability, Boolean Modeling and Computation **7** (2010) 59–64
12. Stahl, T., Völter, M.: Model-driven software development. Wiley (2006)
13. Gonzalez-Perez, C., Henderson-Sellers, B.: Metamodelling for Software Engineering. Wiley (2008)
14. Akehurst, D., Bordbar, B., Evans, M., Howells, W., McDonald-Maier, K.: SiTra: Simple transformations in Java. In: MoDELS'06. LNCS 4199, Springer (2006) 351–364
15. Chen, Y.: Automated synthesis of sequence diagrams. Master's thesis, University of Birmingham (2013)
16. Lanusse, A., Tanguy, Y., Espinoza, H. et al.: Papyrus UML: an open source toolset for MDA. In: ECMDA-FA 2009. (2009) 1–4
17. Liang, H., Diskin, Z., Dingel, J., Posse, E.: A general approach for scenario integration. In: MoDELS'08. LNCS 5301, Springer (2008) 204–218
18. Bordbar, B., Anastasakis, K.: Uml2alloy: A tool for lightweight modelling of discrete event systems. In: IADIS International Conference in Applied Computing. Vol. 1. (2005) 209–216
19. Bowles, J., Bordbar, B.: A formal model for integrating multiple views. In: ACSD 2007, IEEE (2007) 71–79
20. Araújo, J., Whittle, J., Kim, D.: Modeling and composing scenario-based requirements with aspects. In: RE 2004, IEEE (2004) 58–67