On a Process Algebraic Representation of Sequence Diagrams

Jaco Jacobs and Andrew Simpson

Department of Computer Science, University of Oxford
Wolfson Building, Parks Road, Oxford OX1 3QD
United Kingdom

Abstract. Sequence diagrams depict the interaction between entities as a sequence of messages arranged in a temporal order. However, they lack a formal execution semantics: the *Unified Modeling Language* (UML) specification opts to use natural language to describe fundamental concepts such as interaction operators that alter the behaviour of a fragment. *Communicating Sequential Processes* (CSP) is a process-algebraic formalism that is suited to modelling patterns of behavioural interaction. Moreover, the associated refinement checker, *Failures-Divergence Refinement* (FDR), gives rise to a practical approach that enables us to reason about these interactions in a formal setting. In this paper, we show how CSP and FDR have been used to provide a process-algebraic representation of sequence diagrams that is amenable to refinement-checking.

1 Introduction

Sequence diagrams are used to depict the interactions between entities in a sequential, temporal order and have been applied in a wide range of contexts, including: the automatic generation of test cases [1]; the specification of interaction protocols in multi-agent systems [2]; and in technical documentation outlining the specification and design of a product [3]. In this paper, we give consideration to sequence diagrams within the context of the *Systems Modeling Language* (SysML),[1] an extension of a subset of the *Unified Modeling Language*.[2]

The UML specification makes use of meta-models in order to capture the abstract syntax of a diagram. While the benefits of this approach are significant, a drawback is that the execution semantics are expressed using natural language [4, 5]. The lack of sufficient formalism in the specification makes it problematic to interpret the precise meaning of a complex diagram [5]. In addition, the use of natural language may lead to ill-defined semantics, or induce further confusion with regards to how a diagram ought to be interpreted. Thus, approaches that translate UML diagrams into formal representations are advantageous. Our focus is the process algebra *Communicating Sequential Processes* (CSP) [6], with a view to establishing a formal framework that supports the automated reasoning about patterns of behaviour exhibited by sequence diagrams.

[1] `www.sysml.org`
[2] `www.uml.org`

One notable reference where sequence diagrams are translated into CSP (via a model-driven engineering approach) is that of Li and Li [7], where the emphasis is placed on the translation process, which is insightful in terms of a mechanised implementation approach. In contrast, we direct our efforts towards the definition of adequate and succinct CSP processes in an implementation-independent manner. Our objective therefore is to provide concise definitions — using the process algebra CSP — of the patterns of behaviour represented by the different interaction operators. This is done in the spirit of work undertaken by Ng and Butler for state machines [8], and Dong *et al.* for activity diagrams [9]. We view the mechanised translation, using, for example, model-driven techniques, as an implementation of our approach; as such, will address this separately. While these contributions have their benefits, none of them provide a satisfactory (for our purposes) behavioural semantics for sequence diagrams in terms of CSP.

2 Background

2.1 Communicating Sequential Processes

Events are at the heart of CSP, with an event being an indivisible communication or interaction. We denote by Σ the set of all possible events for a particular specification. We can also give consideration to the *alphabet* of a process — the events that it can perform. We write $\alpha\, P$ to denote the alphabet of a process P.

A communication takes place when two or more processes agree on an event. The communication can either be a primitive event, or can take a more structured, message-passing form, utilising channels. The message-passing mechanism is based on the principle of a rendezvous between a sending and a receiving process: if the communication takes place on channel c, and a sending process wants to output a value e, the receiving process has to allow for this (by inputting on c). Once this has happened, the event is abstracted as $c.e$.

CSP is compositional in that it provides operators that allow us to define a process in terms of other, constituent processes. The CSP syntax utilised in this paper can be defined thus (where P, P_1 and P_n denote processes, e denotes an event, X and Y denotes sets of events, and b denotes a Boolean condition):

$$\begin{aligned} P \mathrel{\widehat{=}} {} & P \mid \mathit{Stop} \mid \mathit{Skip} \mid e \rightarrow P \mid \\ & P \mathbin{\Box} P \mid \Box\, e : X \bullet e \rightarrow P \mid P \sqcap P \mid \bigsqcap e : X \bullet e \rightarrow P \mid \\ & P \setminus X \mid P \mathbin{;} P \mid \mathit{if}\ b \text{ then } P \text{ else } P \mid \\ & P \,[\, X \parallel Y \,]\, P \mid P \,[\mid X \mid]\, P \mid \parallel i \bullet [X_i] P_i \mid P \interleave P \mid \interleave\, i \bullet P_i \mid \end{aligned}$$

Stop is the deadlocked CSP process: it will refuse to participate in all events. *Skip* models successful termination: it performs the special internal event ✓, before behaving like *Stop*. The process $e \rightarrow P$, modelled using the *prefixing* operator, performs the event e and subsequently behaves as P.

CSP provides two choice operators: the *external* or *deterministic choice* operator, $\Box$, offers the environment the choice between the initial events of its

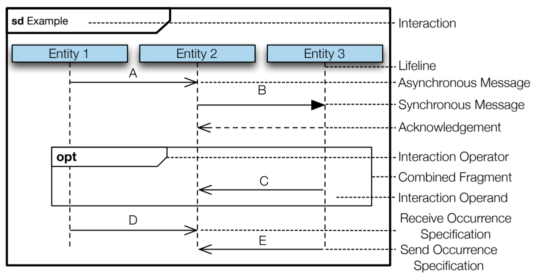

Fig. 1. Relevant constructs of the sequence diagram.

argument processes; conversely, the *internal* or *nondeterministic choice* operator, $\sqcap$, offers no such choice and the observed behaviour may be that of either process. Indexed versions exist for both operators.

The *hiding* operator, $\setminus$, conceals the events of X from the view of the external environment of P. The process $P_1 \fatsemi P_2$ represents the *sequential composition* of P_1 and P_2. This process behaves as P_1 until it terminates successfully, after which it behaves as P_2. A conditional choice construct is available in the form *if* b then P_1 else P_2, where a process behaves as P_1 if b is true and P_2 otherwise.

The process $P_1 \; [| \; X \; |] \; P_2$ uses the *generalised parallel* operator to define an interface on which P_1 and P_2 must synchronise. Events outside X may occur independently in either process. The process $P_1 \, [\, X \parallel Y \,] \, P_2$ denotes *alphabetised parallel*, where synchronisation takes place on events in the set $X \cap Y$. The *interleaving* operator, $|||$, expresses the unsynchronised concurrent interleaving of the events of its constituent processes. Indexed forms exist for each.

The refinement checker Failures-Divergence Refinement (FDR) — which uses the machine-readable dialect of CSP, CSP$_M$ [10] — employs CSP's theory of refinement to investigate whether a potential design meets its specification. If such a test fails, a counter-example is returned to indicate why this is so. We write $P \sqsubseteq_T Q$ when the process Q is a traces-refinement of the process P. While other forms of refinement exist, traces-refinement is sufficient for our purposes.

2.2 Sequence Diagrams

Sequence diagrams facilitate the modelling of interactions between structural constructs as sequences of temporal occurrences. These *interaction occurrences*, or *occurrence observations*, can be broadly categorised into three classes: the sending or receiving of a message; the creation or destruction of an instance; and the start or end of another behaviour. In the interests of brevity, we restrict our treatment to the first class of occurrence observations.

Messages can be exchanged either *synchronously* or *asynchronously*. If the communication is synchronous, the sender blocks until the arrival of a response. Conversely, during an asynchronous exchange, the sender does not block; rather, it continues execution after sending the message. In SysML, for example, an interaction executes within the context of its owning block, and specifies the interaction between parts or references [11]. A sequence diagram depicts this interaction graphically.

Figure 1 shows the notation of interest. On the diagram, lifelines correspond to the parts (or references). A lifeline is represented as a dashed line with the name of the reference or part enclosed in a rectangle. A synchronous message exchange is indicated using a solid line with a filled arrowhead from the sending lifeline to the receiving lifeline; the return message, unblocking the sender, is a dashed line with opposite direction. An asynchronous message is represented using a solid line from the sending lifeline to the receiving lifeline; there is no associated return message as the interaction does not block. When an interaction executes, it produces a sequence of interaction occurrences, termed a *trace*.

Several interaction operators exist. An operator either alters the behaviour of the prescribed sequence, or alters our interpretation of the trace. Examples include the optional interaction operator, *opt*, and the assertion operator, *assert*.

3 Formalisation Using CSP

An interaction, I, is a quintuple of the form $I \mathrel{\widehat{=}} (L_I, E_I, M_I^S, M_I^O, O_I)$, where:

- L_I denotes the set of lifelines of the sequence diagram, I;
- E_I denotes the set of event types (partitioned by disjoint sets for the signals, E_I^S, or operations, E_I^O, that type messages);
- $M_I^S : \mathit{ID} \pinj E_I^S$ uniquely identifies the asynchronous messages of an interaction and associates a message with the signal that typed it;
- $M_I^O : \mathit{ID} \pinj E_I^O$ uniquely identifies the synchronous messages of an interaction and associates a message with the operation that typed it; and
- $O_I \subseteq L_I \times \mathit{seq}\,(\mathit{ID} \times \{\mathit{snd}, \mathit{rcv}, \mathit{ack}\})$ describes all interaction occurrences as a set of pairs, with the first element being the lifeline and the second being a sequence of occurrence observations.

We partition E_I into two disjoint sets, E_I^S and E_I^O, representing *signal events* and *operation events*, respectively. An instance of a signal event corresponds to the sending and receiving of an asynchronous message in the interaction; similarly, an *operation event* types an operation call and can be either synchronous or asynchronous. For the purposes of this paper, we will treat all call operations as synchronous (asynchronous call operations are similar to signals).

To provide each message (we view the acknowledgement message as part of the synchronous message) with a unique identifier, we require that the domains of the functions M_I^S and M_I^O be pairwise disjoint: $\operatorname{dom}(M_I^S) \cap \operatorname{dom}(M_I^O) = \emptyset$.

As an additional constraint, we assume that each synchronous message has an associated acknowledgement (with opposite direction). This acknowledgement is not a message in the conventional sense — it merely exists in order to unblock the sender. We can think of the acknowledgement as a rendezvous between the communicating lifelines in order to unblock the sender. As such, we do not associate it with its own identifier (it uses that of the corresponding synchronous message); nor do we associate with it *snd* or *rcv* occurrence observations. In order for the communicating lifelines to synchronise on this event, both observe it as an *ack*.

In addition, we define the following auxiliary functions:

– $sd : ID \nrightarrow L_I$ returns, for a message identifier, the sending lifeline;
– $rv : ID \nrightarrow L_I$ returns, for a message identifier, the receiving lifeline; and
– $occurrences : L_I \nrightarrow seq\,(ID \times \{snd, rcv, ack\})$ denotes the sequence of event occurrences on the argument lifeline in temporal order.

Interaction occurrences appear in temporal order on a lifeline, with time progressing downwards. An interaction implicitly imposes an order on the messages sent between lifelines. This *weak sequencing* implies that the order of interaction occurrences on a particular lifeline is significant, but that ordering between occurrences on different lifelines can be interleaved. An additional (and seemingly obvious) constraint is that, for a particular message, the send occurrence must happen before the receive occurrence. For example, consider again Figure 1. Message A (and all other messages) must be sent before it can be received. Additionally, for entity 2, A must be received before B can be sent. However, there are no direct constraints between the send occurrences of messages D and E.

Our approach for translating sequence diagrams to CSP is based on mirroring the structure of the corresponding diagram. Broadly, each lifeline is mapped to a process and each occurrence observation is mapped to a CSP event. The process then enforces weak sequencing by insisting that the occurrence observations appear in the temporal order specified on the corresponding lifeline. The acts of sending and receiving a message are completely detached; as such, we require an additional constraint process to enforce the fact that a message cannot be received before it was sent.

We treat the various interaction operators of sequence diagrams using template processes that describe their respective patterns of behaviour. These are defined formally in Section 4.

Consider an interaction, I, with a corresponding sequence diagram. Our approach can be outlined as follows.

– With each lifeline, $l \in L_I$, we associate a sequence of events of the same temporal order. The sequence of events is given by $occurrences\,(l)$. An element of this sequence is a pair of the form (id, obs), where $id \in ID$ and $obs \in \{snd, rcv, ack\}$.
– We model each occurrence observation with a corresponding CSP event. The unique identifier is communicated as part of the event due to the finer nuances of weak sequencing semantics. Let $obs' \in \{snd, rcv\}$. Recall that for acknowledgements we use the same *id* as that of the associated synchronous message. Depending on the observation and the nature of the message, the event takes the following form:
  • for asynchronous messages, $msg.asynch.id.obs'.sd(id).rv(id).M_I^S(id)$
  • for synchronous messages, $msg.synch.id.obs'.sd(id).rv(id).M_I^O(id)$
  • for acknowledgements, $msg.synch.id.ack.rv(id).sd(id).M_I^O(id)$

– Each lifeline has a corresponding CSP process that communicates the events in the required order (defined in the template process).
– For each message in an interaction, we associate a triple, $(\mathit{from}, \mathit{to}, \mathit{name})$, where $\{\mathit{from}, \mathit{to}\} \subseteq L_I$ and $\mathit{name} \in E_I$.
– Each message has an associated process with send and receive occurrence events that synchronise with the appropriate sending and receiving lifelines (defined in the template process).
– Depending on the interaction, we instantiate the correct template process (as defined in the next section) to describe the behaviour.
– A sequence diagram that consists of more than one interaction operator is subsequently defined as the sequential composition of the CSP template processes that describe the respective interaction operators.

The approach does not require fixed sized buffers to model asynchrony, as the sending and receiving lifelines do not synchronise on a message. This allows for a uniform treatment of synchronous and asynchronous messages: in an asynchronous exchange neither the sending nor the receiving lifelines are blocked; conversely, for a synchronous exchange, the sending lifeline blocks until the receiving lifeline communicates the acknowledgement.

In order to simplify the CSP presented here, we do not model passing arguments for call operations or signals; however, these can be readily incorporated via the use of CSP channels.

4 Complex Interactions

Combined fragments allow for the description of complex patterns of interaction in a concise and compact manner. UML (and, therefore, SysML) defines different *interaction operators*, each enabling the specification of different rules with regards to the ordering of messages (and their associated occurrence observations). A combined fragment is an *interaction* operator with associated *operands*. Figure 1 gives an example of the use of the *opt* interaction operator.

The operands of an interaction operator is dependant upon the type of the operator: the alternative and parallel operator each "have multiple horizontal partitions, separated by dashed lines that correspond to their operands. Others have just a single partition" [11]. For single partition operators, their operands correspond to the messages enclosed in the combined fragment. In addition, the operands of the interaction operators follow weak sequencing semantics (unless it is the strict operator): "During execution of an interaction, all operands use weak sequencing semantics on their contents" [11].

The weak sequencing interaction operator, *seq*, is the default. The operator imposes a weak sequencing semantics on its operands, with the operands of the weak sequencing operator being the messages contained within the combined fragment. The UML specification [4] defines weak sequencing as follows.

1. "The ordering of occurrence specifications within each of the operands [messages] are maintained in the result."

2. "Occurrence specifications on different lifelines from different operands [messages] may come in any order."
3. "Occurrence specifications on the same lifeline from different operands [messages] are ordered such that an occurrence specifications of the first operand [message] comes before that of the second operand [message]."

Thus: a message needs to be sent before it can be received; occurrence specifications between different lifelines (also between different messages) impose no additional ordering constraints upon each other; and the temporal order of the occurrence specifications on each lifeline must be honoured.

The process *Message* asserts that the sending of a message necessarily occurs before its reception, as per condition 1. The parameters *type* and *id* correspond to the type (synchronous or asynchronous) and unique identifier, respectively; *from* and *to* model the sending and receiving lifelines; and *name* corresponds to the signal or operation (an instance of an event type).

$$\begin{aligned}
&\mathit{Message}\,(\mathit{type}, \mathit{id}, \mathit{from}, \mathit{to}, \mathit{name}) = \\
&\quad \mathit{msg.type.id.snd.from.to.name} \to \mathit{msg.type.id.rcv.from.to.name} \to \mathit{Skip} \\
&\alpha\,\mathit{Message}\,(\mathit{type}, \mathit{id}, \mathit{from}, \mathit{to}, \mathit{name}) = \\
&\quad \{\!|\ \mathit{msg.type.id.snd.from.to.name}, \mathit{msg.type.id.rcv.from.to.name}\ |\!\}
\end{aligned}$$

PrefixComposition, if supplied a sequence as input, is the process that communicates the events in order and then behaves as *Skip*. Given a temporal sequence of interaction occurrences for a lifeline, we use *PrefixComposition* to enforce condition 3:

$$\begin{aligned}
&\mathit{PrefixComposition}\,(s) = \\
&\quad \text{if } \mathit{null}\,(s) \text{ then } \mathit{Skip} \text{ else } \mathit{head}\,(s) \to \mathit{PrefixComposition}\,(\mathit{tail}\,(s))
\end{aligned}$$

The process *Lifelines* models the parallel composition of a set of lifelines. The process takes as input a set of sequences, where each sequence describes occurrence specifications for a lifeline in temporal order. Each lifeline in the composition synchronises on its entire alphabet. (In the following, the function *set* converts a sequence to a set.)

$$\begin{aligned}
&\mathit{Lifelines}\,(l) = \big\| \ \mathit{line} : l \bullet [\mathit{set}\,(\mathit{line})]\mathit{PrefixComposition}\,(\mathit{line}) \\
&\alpha\,\mathit{Lifelines}\,(l) = \bigcup \{\mathit{line} : l \bullet \mathit{set}\,(\mathit{line})\}
\end{aligned}$$

The process *Messages* is the parallel composition of the *Message* processes, with each taking a quintuple of the form $(\mathit{type}, \mathit{id}, \mathit{from}, \mathit{to}, \mathit{name})$ as input.

$$\begin{aligned}
&\mathit{Messages}\,(m) = \\
&\quad \big\| (t, \mathit{id}, \mathit{from}, \mathit{to}, n) : m \bullet [\mathit{Message}\,(t, \mathit{id}, \mathit{from}, \mathit{to}, n)] \\
&\alpha\,\mathit{Messages}\,(m) = \\
&\quad \bigcup \{(t, \mathit{id}, \mathit{from}, \mathit{to}, n) : m \bullet \alpha\,\mathit{Message}\,(t, \mathit{id}, \mathit{from}, \mathit{to}, n)\}
\end{aligned}$$

We can now model weak sequencing behaviour. By placing *Messages* and *PrefixComposition* in parallel, we restrict the traces to adhere to the behaviours

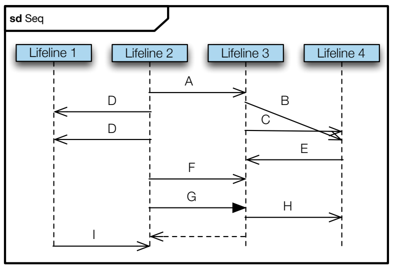

Fig. 2. The default seq operator (adapted from [11]).

imposed by the first and last condition. Condition 2 places no further restrictions on the behaviour, and, as the interaction occurrences between different lifelines do not have any shared events in common, we require no process to model this behaviour. *Seq*, which models weak sequencing, is defined thus (for brevity, we write $\alpha\, \mathit{Lifelines}\,(l)$ and $\alpha\, \mathit{Messages}\,(m)$ as L_α and M_α, respectively):

$$\mathit{Seq}\,(l, m) = \mathit{Lifelines}\,(l)\; [\, L_\alpha \parallel M_\alpha \,]\; \mathit{Messages}\,(m)$$

The operands of the strict sequencing operator, *strict*, are the messages contained within the combined fragment: "the semantics of strict sequencing defines a strict ordering of the operands [messages]" [4].

Strict sequencing semantics therefore impose an additional constraint upon weak sequencing, in that the operands (messages) must be sequenced across all participating lifelines [11]. This implies that, for a particular message, the send and receive occurrences must occur in strict succession.

We can subsequently define strict sequencing by placing another process (*Enforce*) in parallel to constrain the behaviour of weak sequencing.

The process *Strict* is defined as follows:

$$\begin{aligned}
\mathit{Strict}\,(l, m) &= (\mathit{Lifelines}\,(l)\; [\, L_\alpha \parallel M_\alpha \,]\; \mathit{Messages}\,(m))\; [|\; M_\alpha \;|]\; \mathit{Enforce}\,(m) \\
\mathit{Enforce}\,(m) &= \square\, (msg.m.i.snd.f.t.n) : M_\alpha \bullet \\
&\qquad msg.m.i.snd.f.t.n \rightarrow msg.m.i.rcv.f.t.n \rightarrow \mathit{Enforce}\,(m) \\
&\quad \square\; \mathit{Skip}
\end{aligned}$$

Our approach allows for detecting when the operands of an interaction operator are not well-defined. For example, when we try and enforce strict semantics on the sequence of Figure 2, FDR detects a *deadlock* and returns a counter-example — message overtaking is not possible using strict semantics.

The parallel operator, *par*, designates an interleaving between its operands. The horizontal partitions (within the combined fragment) correspond to the operands. The *interleaving* operator of CSP models this pattern of behaviour perfectly. We therefore define the *par* interaction operator as the interleaved behaviour of sequentially interleaved processes. For readability, the definition below assumes that there are only two partitions within the combined fragment;

we can, however, easily extend this to cover more partitions, or even generalise the definition to cover an arbitrary number of horizontal partitions.

$$Par\,(l_1, m_1, l_2, m_2) = Seq\,(l_1, m_1) \mathbin{|||} Seq\,(l_2, m_2)$$

The alternative operator, *alt*, offers the choice between the behaviours of its operands, based on the guard associated with each partition. Recall that the horizontal partitions (within the combined fragment) correspond to the operands. In a scenario in which more than one guard evaluates to true, the choice is nondeterministic; if none evaluate to true, an optional else partition is selected [11]. We can use the *nondeterministic* and *conditional choice* constructs to model this behaviour. Below we provide a definition for a combined fragment consisting of two conditionally guarded partitions and one else clause. This definition can be generalised to handle an arbitrary number of conditional clauses, but a simplified version is presented here to illustrate the concepts.

$$\begin{array}{l} Alt\,(l_1, m_1, g_1, l_2, m_2, g_2, l_3, m_3) = \\ \quad \text{if } (g_1 \land g_2) \text{ then } Seq\,(l_1, m_1) \sqcap Seq\,(l_2, m_2) \\ \quad \text{else if } g_1 \text{ then } Seq\,(l_1, m_1) \\ \quad\quad \text{else if } g_2 \text{ then } Seq\,(l_2, m_2) \text{ else } Seq\,(l_3, m_3) \end{array}$$

The operator *opt* models optional behaviour. The operand (messages contained within the combined fragment) is only executed if the guard condition is true. This behaviour is precisely that of an *alt* operator with a single operand.

$$Opt\,(l, m, g) = \text{if } (g) \text{ then } Seq\,(l, m) \text{ else } Skip$$

The *break* interaction operator is used to model a breaking scenario from another enclosing fragment. The behavioural semantics is such that if the guard associated with the break evaluates to true, then its operand is executed (rather than the remainder of the enclosing fragment). For example, consider a break nested within an enclosing *seq* fragment, which we model in terms of the process *Break*. The first two parameters (l_{pre} and m_{pre}) describe the lifelines and messages of the enclosing fragment preceding the break; the final two parameters (l_{post} and m_{post}) model the remainder of the enclosing behaviour. The l, m and g parameters correspond to the operands of the break fragment.

$$\begin{array}{l} Break\,(l_{pre}, m_{pre}, l, m, g, l_{post}, m_{post}) = \\ \quad Seq\,(l_{pre}, m_{pre}) \fatsemi (\text{if } g \text{ then } Seq\,(l, m) \text{ else } Seq\,(l_{post}, m_{post})) \end{array}$$

The *loop* operator repeats its operand (the messages contained within the combined fragment) until the termination condition imposed upon it is satisfied. The semantics of the loop operator allows for the termination condition to be expressed as either: an iteration bound (of the form (*lower*, *upper*) or (*exact*)); a Boolean condition; or a combination of both. (In practice, however, one would use one or the other, rather than a combination.)

The UML specification is ambiguous with regards to the semantics when the termination condition is expressed as a combination of an iteration bound and

Boolean guard: it is unclear what happens if the Boolean condition evaluates to false before the minimum number of iterations have executed. This ambiguity arises as a result the following two quotes from the UML specification: “after the minimum number of iterations have executed and the Boolean expression is false the loop will terminate” [4], and “the loop will only continue if that specification evaluates to true during execution regardless of the minimum number of iterations specified in the loop” [4]. As such, we consider in our treatment only the cases where either an iteration bound or Boolean guard is specified.

The *sequencing* operator of CSP is used to express behaviour as a sequence of process executions. We can convey the desired behaviour of the loop operator through successive application of the sequencing operator (to the CSP process modelling the behaviour of the operand) in accordance with the stated termination condition. Consider the case where there is a single integer iteration bound is specified as the termination condition. The process *Loop* models this:

$$\mathit{Loop}\,(l, m, e) = \text{if } (e \geq 1) \text{ then } (\mathit{Seq}\,(l, m) \mathbin{;} \mathit{Loop}\,(l, m, e-1)) \text{ else } \mathit{Skip}$$

5 Interaction Interpretation

The interaction operators described in the previous section allowed us to model different forms of control flow — alternative or parallel behaviour, for example. In this section, we introduce the three operators that change our interpretation of a particular interaction sequence. We discuss these in the context of how they might possibly be used in a refinement check. In addition, we motivate why it is inappropriate to define process definitions in the spirit of the preceding section.

The *ignore* interaction operator provides, as part of the combined fragment, a set of messages that are to be ignored. Consequently, the messages are not allowed within the interaction fragment. The interpretation is that the messages are insignificant and irrelevant and are to be ignored if they appear in the interaction. An alternative interpretation is that the ignored messages can appear anywhere in a trace and still be considered valid.

It is possible to model this as a template process, where the ignored traces are interleaved with those of the interaction (assuming we followed the second interpretation, and *ignore* contained all the valid observations of the ignored events between participating lifelines):[3]

$$\mathit{Ignore}\,(l, m, \mathit{ignore}) = \mathit{Seq}\,(l, m) \mathbin{|||} \mathit{Run}\,(\mathit{ignore})$$

A more elegant solution can be achieved via the hiding operator and the first interpretation: in a refinement, we simply hide the ignored events from any behaviour we are comparing against. For example, $\mathit{StateMachines} \setminus \mathit{ignore} \sqsubseteq_T \mathit{Seq}\,(\mathit{lifelines}, \mathit{messages})$ would test if an interaction is valid for a pair of communicating state machines, *StateMachines*.

The *consider* interaction operator specifies a set of messages that are to be considered as part of this combined fragment; all other messages are ignored.

[3] Here, $\mathit{Run}\,(E) = \Box\, e : E \bullet e \rightarrow \mathit{Run}\,(E)$.

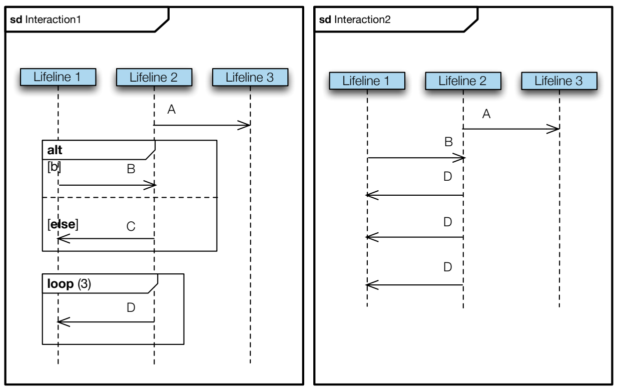

Fig. 3. Example 1

Consequently, the combined fragment can only contain the considered messages. The semantics is interpreted to mean that other messages might occur as part of the interaction, but that these are irrelevant and ought to be ignored. The *consider* operator can be defined in terms of *ignore*: ignore all other messages not considered. As was the case for *ignore*, there exists an alternative interpretation, where all messages that are not considered may appear anywhere in the traces. (In the interests of brevity, we do not expand further on the *consider* operator.)

The assertion operator, *assert*, declares that the interaction fragment models the only valid continuations; any other eventuality is considered invalid. In this case, we need the refinement relation to hold in both directions.

6 Examples

Having defined a process-algebraic formal semantics for sequence diagrams, we can test whether the behaviour of one interaction sequence is contained within another by considering trace semantics. Consider Figure 3. If we regard the behaviour (in terms of traces) of $I_2 = \mathit{Seq}\,(L_2, M_2)$ as the valid behaviours (a *safety specification*), and we want to test whether another interaction sequence, $I_1 = \mathit{Seq}\,(L_{11}, M_{11}) \mathbin{⨟} \mathit{Alt}\,(L_{12}, M_{12}, b, L_{13}, M_{13}) \mathbin{⨟} \mathit{Loop}\,(L_{14}, M_{14}, 3)$, does not deviate from this, we can use a traces-refinement ($I_2 \sqsubseteq_T I_1$) to confirm this.

As another example, we might want to be sure that interaction diagrams at different levels of the specification are consistent (see Figure 4). Such *vertical consistency* problems are induced by a development process where models are iteratively refined: we start with an interaction sequence at a higher level and add more detail as we move closer to the implementation level specification. Assuming $\mathit{Higher} = \mathit{Seq}\,(L_h, M_h)$ and $\mathit{Lower} = \mathit{Seq}\,(L_l, M_l)$, we can check whether $\mathit{Higher} \sqsubseteq_T \mathit{Lower} \setminus \mathit{hidden}$ (where *hidden* denotes those occurrence observations present at the lower level, but not at the higher level).

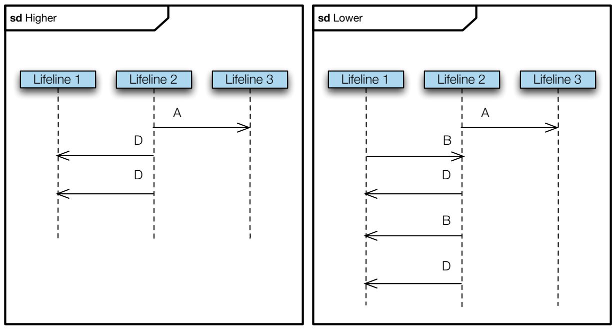

Fig. 4. Example 2

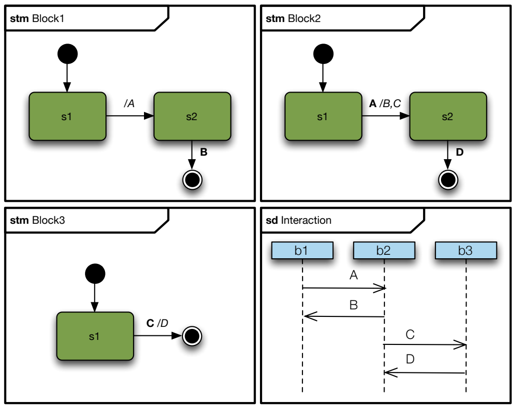

Fig. 5. Example 3

Finally, we might make use of sequence diagrams to check the validity of communicating state machines, as described in [12, 13]. We can, for example, test whether a particular sequence of events is possible when we consider the combined behaviour of a set of communicating state machines. We can check $\mathit{Blocks} \sqsubseteq_T \mathit{SEQ}$, where $\mathit{SEQ} = \mathit{Seq}\,(L, M)$. Here, *Blocks* denotes the compositional process describing the combined behaviour of the communicating state machines. We would also expect to make use of the CSP renaming operator in order to consolidate the events of our interaction semantics with the events of the state machine semantics, as proposed in [13].

7 Related Work

State machine diagrams were given a CSP semantics by Ng and Butler in [8]; activity diagrams were formalised by Dong *et al.* [9]. To the best of our knowledge, there has been no such mapping done in the spirit of the aforementioned papers for sequence diagrams. Both [8] and [9] focus on the provision of a CSP semantics in an implementation-independent fashion; this was our goal for sequence diagrams. Other examples where state-based graphical models have been given a formal CSP semantics include [14] and [15].

Li and Li [7] considered the automatic translation of sequence diagrams to CSP using a model-driven approach. Sibertin-Blanc *et al.* [16] showed four possible semantic interpretations of sequence diagrams, partly due to the semi-formal nature of the UML specification. Rasch and Wehrheim [17] used sequence diagrams to check the validity of scenarios in a UML model. Our work differs, in that they define a semantics for sequence diagrams in terms of the messages communicated; in addition, they exclude the interaction operators from their analysis. Our work considers sequence diagrams in terms of occurrence observations, rather than messages, and extends to all operators. The checking of the validity of scenarios, using our semantics as a model of interaction, will be a focus of future research. Other notable works of reference can be found in [18] and [19].

8 Discussion

We have introduced patterns of behaviour to model the interaction operators as per the UML standard. In addition, we have provided a uniform treatment of synchronous and asynchronous messages. Furthermore, our approach does not rely on fixed size buffers in order to model asynchronous exchanges. Finally, we are able to deal with lost and found messages, as well as message overtaking.

The process-algebraic approach suggested enables us to compare the behaviour of a sequence of interactions against another interaction in a natural fashion. This is in contrast to approaches that rely on traditional model checking, such as the work of Lima *et al.* [20] — where such comparisons are not possible. Furthermore, the only other formalisation of the semantics of sequence diagrams that makes use of CSP that we are aware of is that of Li and Li [7]. Our approach differs in that we define our semantics for sequence diagrams in terms of templates that describe the patterns of behaviour for the various interaction operators. Additionally, we consider the *seq*, *strict*, *ignore*, *consider*, and *assert* operators. The advantage of our approach is that any implementation of an automated translation mechanism would only have to instantiate the proposed CSP processes in order to describe the behaviour of the desired interaction operator.

The work of Li and Li [7] models the sending of a message between lifelines L_1 and L_2 using the channel construct, with the lifelines synchronising on the message being exchanged. The problem here, from our perspective, is that we require the sending and receiving of a message to be modelled as two, separate, detached events (the sending and receiving occurrence specifications related to the message exchange). However, the suggested approach abstracts them into a single event. This might have been appropriate, for example, if we were only concerned with the act of exchanging a message. However, this is not our desire here. Instead, we wish to decompose the exchange into two separate events. In doing so, we will be able to operate our CSP models at a finer granularity.

Consider making use of sequence diagrams to check the validity of communicating state machines, as described by the present authors in [12] and [13]. Activities are considered in [21]. Using our model for sequence diagrams, we would be able to make use of events (like a state machine sending an asynchronous message) that correspond to interaction occurrences on the sequence diagram. Of course, the sending of an asynchronous message by one state machine does not guarantee that the message is received by another. Even if it is received immediately, it might still be placed in an event queue, so the receiving state machine might only process it later. If we operated at a coarser granularity, we would have to be content with only modelling the exchange of the message, making it impossible to distinguish between when it was sent and when it was received.

The approach presented here is novel as we give a detailed account of interaction operators. Moreover, due to the nature of a process-algebraic formalism like CSP, where the focus is on describing intricate patterns of behaviour, we are able to deal with interaction operators that alter our interpretation of an interaction sequence more naturally that in approaches that rely on traditional model checking using temporal logics [20]. In addition, the refinement checker, FDR, which allows the behaviour of one process to be compared against that of another in terms of a refinement hierarchy, provides a practical means of comparing behaviour of one sequence diagram against that of another (incorporating the operators that alter interaction interpretation, for example).

Possible areas of future work include checking the validity of scenarios. Other avenues that we will be pursuing include the development of a model-driven framework that automates the translation approach introduced in this paper. The resulting framework should ultimately enable us to verify the validity of interactions at a more fundamental level.

References

1. Swain, S.K., Mohapatra, D.P., Mall, R.: Test case generation based on use case and sequence diagram. International Journal of Software Engineering **3**(2) (2010) 21–52
2. Odell, J.J., Van Dyke Parunak, H., Bauer, B.: Representing agent interaction protocols in UML. In: Proceedings of the 1st International Workshop on Agent-Oriented Software Engineering (AOSE 2000). Volume 1957 of Lecture Notes in Computer Science. Springer (2001) 121–140
3. Bist, G., MacKinnon, N., Murphy, S.: Sequence diagram presentation in technical documentation. In: Proceedings of the 22nd International Conference on Design of Communication: The Engineering of Quality Documentation (SIGDOC 2004), ACM (2004) 128–133
4. Object Management Group: Unified Modeling Language Specification, version 2.4.1. (2011)
5. Kim, S.K., Carrington, D.A.: A formal model of the UML metamodel: the UML state machine and its integrity constraints. In: Proceedings of the 2nd International

Conference of B and Z Users on Formal Specification and Development in Z and B (ZB 2002). Volume 2272 of Lecture Notes in Computer Science. Springer (2002) 497–516

6. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall (1985)
7. Li, D., Li, D.: An approach to formalize UML sequence diagrams in CSP. International Proceedings of Computer Science and Information Technology **53**(2) (2010) 109–115
8. Ng, M.Y., Butler, M.: Towards formalizing UML state diagrams in CSP. In: Proceedings of the 1st International Conference on Software Engineering and Formal Methods (SEFM 2003), IEEE (2003) 138–147
9. Dong, X., Philbert, N., Zongtian, L., Wei, L.: Towards formalizing UML activity diagrams in CSP. In: Proceedings of the International Symposium on Computer Science and Computational Technology (ISCSCT 2008), IEEE (2008) 450–453
10. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice Hall (1997)
11. Friedenthal, S., Moore, A., Steiner, R.: A Practical Guide to SysML: The Systems Modeling Language. Morgan Kaufmann Publishers (2008)
12. Jacobs, J., Simpson, A.C.: A process algebraic approach to decomposition of communicating SysML blocks. International Journal of Modeling and Optimization **3**(2) (2013) 153–157
13. Jacobs, J., Simpson, A.C.: Towards a process algebra framework for supporting behavioural consistency and requirements traceability in SysML. In: Proceedings of the 15th International Conference on Formal Engineering Methods (ICFEM 2013). Volume 8144 of Lecture Notes in Computer Science. Springer (2013) 266–281
14. Yeung, W.L., Leung, K.R.P.H., Dong, W., Wang, J.: Improvements towards formalising UML state diagrams in CSP. In: Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC 2005), IEEE (2005) 176–182
15. Roscoe, A.W., Wu, Z.: Verifying Statemate statecharts using CSP and FDR. In: Proceedings of the 8th International Conference on Formal Engineering Methods (ICFEM 2006). Volume 4260 of Lecture Notes in Computer Science. Springer (2006) 324–341
16. Sibertin-Blanc, C., Hameurlain, N., Tahir, O.: Ambiguity and structural properties of basic sequence diagrams. Innovations in Systems and Software Engineering **4**(3) (2008) 275–284
17. Rasch, H., Wehrheim, H.: Checking the validity of scenarios in UML models. In: Proceedings of the 7th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2005). Volume 3535 of Lecture Notes in Computer Science. Springer (2005) 67–82
18. Sibertin-Blanc, C., Tahir, O., Cardoso, J.: Interpretation of UML Sequence Diagrams as Causality Flows. In: Proceedings of the 5th International School and Symposium (ISSADS 2005). Volume 3563 of Lecture Notes in Computer Science. Springer (2005) 126–140
19. Bernardi, S., Merseguer, J.: Performance evaluation of UML design with Stochastic Well-formed Nets . Journal of Systems and Software **80**(11) (2007) 1843–1865
20. Lima, V., Talhi, C., Mouheb, D., Debbabi, M., Wang, L., Pourzandi, M.: Formal verification and validation of UML 2.0 sequence diagrams using source and destination of messages. Electronic Notes in Theoretical Computer Science **254** (2009) 143–160
21. Jacobs, J., Simpson, A.C.: On the formal interpretation of SysML blocks using a safety critical case study. In: Proceedings of the 8th Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS 2014), IEEE (2014)