Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 203 (2009) 3–22

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

A True-Concurrent Interpretation of Behavioural Scenarios

Sotiris Moschoyiannis[1] , Paul Krause[2] and Michael W Shields[3]

Department of Computing, University of Surrey, Guildford, Surrey, GU2 7XH, UK

Abstract

We describe a translation of scenarios given in UML 2.0 sequence diagrams into a tuples-based behavioural model that considers multiple access points for a participating instance and exhibits true-concurrency. This is important in a component setting since different access points are connected to different instances, which have no knowledge of each other. Interactions specified in a scenario are modelled using tuples of sequences, one sequence for each access point. The proposed unfolding of the sequence diagram involves mapping each location (graphical position) onto the so-called *component vectors*. The various modes of interaction (sequential, alternative, concurrent) manifest themselves in the order structure of the resulting set of component vectors, which captures the dependencies between participating instances. In previous work, we have described how (sets of) vectors generate *concurrent automata*. The extension to our model with sequence diagrams in this paper provides a way to verify the diagram against the state-based model.

Keywords: interactions, multiple ports, concurrency, UML 2.0 sequence diagrams, vector semantics

1 Introduction

Modern software applications increasingly take a view of computation as something that happens by and through *interaction*. Software systems are architected using components, which provide a coherent set of services through well-defined interfaces typed by ports. The overall system functionality is the result of, often complex and highly concurrent, interactions between components of the system. A thorough understanding of the behaviour exhibited at the interfaces of a component can increase expectations of a successful outcome prior to deployment.

A component in pragmatic approaches to software design such as UML [12] or the Koala component model [15] is understood as having multiple access points (i/o

[1] Email:s.moschoyiannis@surrey.ac.uk

[2] Email:p.krause@surrey.ac.uk

[3] Email:m.shields@surrey.ac.uk

doi:10.1016/j.entcs.2009.03.023

ports) through which it provides and requires services, by interacting with other components. UML2.0 sequence diagrams, Message Sequence Charts (MSc) [5] and Live Sequence Charts (LSCs) [2] are the mainstay of industrial software specifications describing interactions within the context of a given scenario. The (informal) semantics of a sequence diagram defined in any of the above notations is given in terms of a partial order on the events appearing in the interaction described in the diagram. Additional constructs can be superimposed on the diagram to denote concurrent interactions or alternative scenarios. Although useful, the graphical notation alone cannot be relied upon for rigorous analysis and formal verification. Scenarios need to be translated into some other, more formal, notation with a well-defined behavioural semantics.

There is an increasing interest (e.g. [8,7,14,6,4]) in the formal translation of scenarios, but current approaches seem to be geared towards an interleaving semantics and also do not consider participating instances (components) as having multiple access points (ports). In a component setting, different ports of a component are connected to different components, which have no knowledge of each other, and thus the case that services are requested/offered at the same time cannot be reasonably excluded. A non-interleaving interpretation of concurrency [16] considers such cases and can faithfully describe concurrent interactions.

In previous work [10,9,11] we have been concerned with a formal model of components exhibiting true-concurrency [13]. Component interactions are modelled using tuples of sequences, one for each access point. These so-called *component vectors* are expressive enough to model the component offering or requiring services concurrently through its multiple ports. Such vectors, formed over a given component signature Σ (cf Definition 3.1), give rise to a certain class of automata [11] in which concurrency is expressed as an explicit structural property. Of course, we are interested in *that* subset of all possible component vectors over Σ that describes intended behaviour only.

In this paper we describe a formal translation of scenarios given in UML2.0 sequence diagrams into component vectors. This is used for restricting to an appropriate subset of all possible vectors that describes constraints on the order in which the services of the component can or should be called. We give a mathematical construction for the unfolding of scenarios into *vector languages*, based on a non-interleaving semantics for sequence diagrams.

The formalisation of the information conveyed by a UML sequence diagram, including the constructs for expressing parallel and alternative scenarios, is based on that found in [8], which includes notation for moving down the diagram and identifying any particular constructs. In the context of components, we are interested in the intended behaviour of a component and thus we adopt the formal definition of a sequence diagram given in [8] to a single lifeline rather than the diagram as a whole. In our formal model of components, the system behaviour exhibited during the execution of the scenario as a whole is then given in terms of composition of the components involved, as described in [10].

This paper is structured as follows. Section 2 gives a brief account of UML models used for describing configurations of components and their interactions. In Section 3 we outline the formal description of a component in our approach. Interactions are modelled by component languages, essentially sets of component vectors. In Section 4 we show how UML sequence diagrams can be translated into component languages, by mapping interactions onto component vectors. Section 5 contains some concluding remarks, comments on related work and sketches some ideas for future work.

2 Components in UML 2.0

In UML 2.0 [12], a component is a special kind of a *structured classifier*; a class which has some internal structure made of parts. A component can offer one or more interfaces, and require one or more interfaces from other components. To avoid a client of the component being able to invoke public services of a part, the component can (and should) be encapsulated using ports. Ports have a name and type. Interfaces are associated to ports, which are instantiable and can thus mediate the handling of call requests to and from the component via interfaces in a state-based way. This can be exploited in providing a direct model of the state of the interaction, e.g. see [11].

Fig. 1 on the left shows a configuration of components using the UML 2.0 notation. Component `c1` provides services to components `c0` and `c3` through ports `p11` and `p13`, respectively, and requires services from `c2` through port `p12`. Sequence

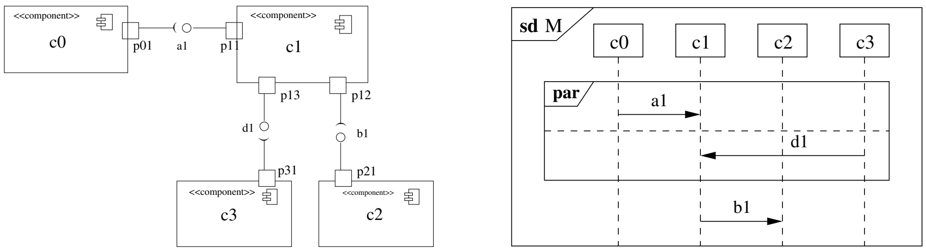


Fig. 1. Component architecture and component interactions

diagrams describe a behavioural view of a system showing the interactions between objects or components in the system. In a nutshell, a sequence diagram displays participating instances as lifelines running down the page (dashed vertical lines in Fig. 1) and their interactions over time are represented as messages drawn as horizontal arrows between lifelines. Each sending/receiving of a message is associated to an event of the sender/receiver.

Sequence diagrams in UML 2.0 [12] have been considerably revised in relation to those in previous versions of UML. They have been extended to include features from MSCs [5] and, to a lesser extent, from LSCs [2] and as a result they are more expressive and fundamentally better structured. Fig. 1 on the right shows an example UML 2.0 sequence diagram describing the interactions between the components on the left.

One of the major changes has to do with the introduction of sub-interactions called *interaction fragments* which can be combined using *interaction operators*. Interaction fragments may comprise one or more *operands* (compartments) depending on the corresponding interaction operator. For example, the **alt** operator designates that the corresponding interaction fragment represents a choice between alternative scenarios. The **seq** interaction operator (default for sequence diagrams) designates that the interaction fragment specifies a sequencing between the corresponding event occurrences. The **par** interaction operator is used to represent parallel execution of the behaviours from different operands and the resulting **par** interaction fragment models concurrent interactions. In Fig. 1, component `c1` receives requests $a1$ and $d1$ concurrently from `c0` and `c3`.

It can be seen that components in UML 2.0 (and elsewhere, eg. Koala [15]) are understood as having multiple access points (ports, interfaces) through which they interact with other components. An intrinsic aspect of component-based design is the ability to capture the ordering relations between event occurrences on component ports and the dependencies that arise (whether they are ordered, concurrent or in conflict).

3 Component Model

In this section, we discuss the formal description of a component in our approach. We restrict the discussion to the basic concepts necessary to explain the context in which we will later use UML 2.0 sequence diagrams. A more detailed account of the component model can be found in [9,10].

We have seen in Section 2 that a component in UML is pictured with a number of provided and required interfaces which are associated with ports. This is also in line with the way components are rendered in Koala [15]. The following definition captures the static characteristics that serve to identify a component. We shall assume a (countably infinite) set of port names $\mathcal{P}$ and a (countably infinite) set of names for interface operations Op.

Definition 3.1 A *component signature* is a tuple $\Sigma = (I, O, \beta)$ where

- $I \subseteq \mathcal{P}$ is a set of *input* ports, each typed by one or more *provided* interfaces
- $O \subseteq \mathcal{P}$ is a set of *output* ports, each typed by one or more *required* interfaces
- $\beta : I \cup O \to \wp(Op)$ returns the set of operations associated with a port p

and we require that $I \cap O = \emptyset$. Define $P_\Sigma = I \cup O$ and $Op_\Sigma = \bigcup_{p \in P_\Sigma} \beta(p)$.

In any behaviour of the system, each port will experience sequences of events (calls to interface operations) formed over the corresponding set $\beta(p)$. We simply describe the behaviour of the component as a whole by assigning such sequences to each of its ports.

Definition 3.2 Suppose that Σ is a component signature. We define V_Σ to be the set of all functions $v : P_\Sigma \to Op_\Sigma^*$ such that for each $p \in P_\Sigma, v(p) \in \beta(p)^*$. We shall refer to elements of V_Σ as *component vectors*.

By $\beta(p)^*$ we denote the set of finite sequences over $\beta(p)$. Mathematically, the set V_Σ is the Cartesian product of the sets $\beta(p)^*$, for each p. A function $\underline{v}$ of the definition maps each port to a finite sequence of events formed over the corresponding set $\beta(p)$. Effectively, component vectors are n-tuples of sequences where each coordinate corresponds to a port (hence, n is the number of ports) and contains a finite sequence of events (calls to operations) that have occurred over that port.

The set of vectors V_Σ describes *all possible* behaviours of a component, given its signature Σ. In describing component behaviour however, we are mostly interested in what the component is *intended* to do. Within our approach this amounts to restricting to an appropriate subset of V_Σ comprising component vectors that describe *intended* or *permitted* behaviour only.

Definition 3.3 A component is a pair (Σ, V), where Σ is the signature and $V \subseteq V_\Sigma$ is the *component language*.

Thus, a component consists of the static structure described by a signature Σ together with a 'language' V of component vectors, formed over Σ. Intuitively, the idea is that the component language describes the intended behaviour in that it indicates possible constraints on the order in which the operations of the component can or should be called.

This is done by exploiting the basic order-theoretic properties of component vectors. We have seen that component vectors are essentially tuples of sequences. We may thus define operations on component vectors *coordinate-wise* in terms of well known operations on sequences. For $\underline{u}, \underline{v} \in V_\Sigma$, we define,

- $\underline{u}.\underline{v}$ to be the unique vector w such that $w(p) = \underline{u}(p).\underline{v}(p)$, for each $p \in P_\Sigma$ (*concatenation*)
- $\underline{u} \leq \underline{v}$ iff $\underline{u}(p) \leq \underline{v}(p)$, for each $p \in P_\Sigma$ (*prefix ordering*)
- $\underline{u} \sqcap \underline{v}$ to be the vector w which satisfies $w(p) = min(\underline{u}(p), \underline{v}(p))$, for each p
- $u \sqcup \underline{v}$ (if it exists) to be the vector w which satisfies $w(p) = max(u(p), \underline{v}(p))$
- if $u \leq \underline{v}$, then we define $\underline{v}/u$ to be the unique element $\underline{z} \in V_\Sigma$ such that $u.\underline{z} = \underline{v}$ (*right-cancellation*)

Note that $u \sqcup v$ is defined only when $max(u(p), v(p))$ exists, for each p. It is easy to see that V_Σ is a monoid with binary operation '.' and identity Λ_Σ, where Λ_Σ is the empty vector. The empty vector assigns the empty sequence, denoted by Λ, to each interface of the component. Furthermore, V_Σ is a partially ordered set (poset) with partial order '$\leq$' and bottom element Λ_Σ. The operations '$\sqcup$' and '$\sqcap$' give the greatest lower bound and the least upper bound, respectively, of $u, v \in V_\Sigma$, in the usual sense of lattices and domain theory [3,16]. The right-cancellation operator says that if u is an initial part of the behaviour v, then v/u is the 'continuation' of u that extends it to v.

We may readily consider an independence relation on component vectors, which is central to expressing true-concurrency within component languages.

Definition 3.4 Let u, v be component vectors in V_Σ. We define u and v to be *independent*, and we write $\underline{u}\ ind\ \underline{v}$, iff $\forall p \in P_\Sigma : \underline{u}(p) > \Lambda \Rightarrow \underline{v}(p) = \Lambda$.

Effectively, the independence relation implies that behaviours which may happen concurrently engage distinct ports of the component. In component-based development, different ports of the component will be connected to different components which have no knowledge of each other and thus cannot be expected to respect any particular ordering in issuing requests over their allocated port. It is important to note that independence alone does not guarantee concurrency - there is the additional requirement that the events concerned are both offered after some behaviour and occur consecutively.

Component vectors are obtained by coordinatewise concatenation, for example, $(x_1, x_2, x_3).(y_1, y_2, y_3) = (x_1y_1, x_2y_2, x_3y_3)$. In describing component interactions, we are interested in event occurrences over ports of the component. These are captured in our formalism using a specific kind of component vectors, termed *column vectors*, which have at most one event per coordinate.

Definition 3.5 Suppose that Σ is a component signature. Define $E_\Sigma = \{\underline{e} \in V_\Sigma \setminus \{\underline{\Lambda}_\Sigma\} : p \in P_\Sigma \Rightarrow |\underline{e}(p)| \leq 1\}$ where $|x|$ denotes the length of sequence x. We also define $E_\Sigma^\perp = E_\Sigma \cup \{\underline{\Lambda}_\Sigma\}$.

For example, $\underline{e} = (\Lambda, d1)$ represents an operation call $d1$ on the port corresponding to the second coordinate. If $d1$ is intended to occur only after both $a1$ and $d2$ have, then this is described in a component vector $\underline{v} = (a1, d2d1)$ which is obtained as $\underline{u}.\underline{e} = (a1, d2).(\Lambda, d1) = (a1, d2d1) = \underline{v}$. Hence, every event occurrence appears on a new vector on the appropriate coordinate.

In order to ensure that vectors in a component language are the result of concatenations with column vectors only, the language must satisfy certain properties, namely *discreteness* and *local left-closure*. These properties lead to the characterisation of *normal* component languages and give rise to a class of automata [11] that provide a state-based description of component interactions in which concurrency is captured as an explicit structural property. Due to space limitations we do not discuss the state-based model further in this paper.

We have now set up the necessary machinery for expressing behavioural dependencies between ports of a component. Such dependencies are manifested in the order structure of component languages which is dependent on context - on what other vectors are included. This is illustrated in Fig. 2 which uses Hasse diagrams to depict the order structure of component languages in which $a1, d1$ are sequential in (i), concurrent in (ii), and in conflict in (iii). The events $a1$ and $d1$, represented

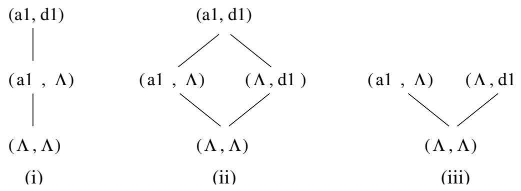


Fig. 2. Order structure of component languages

by the independent column vectors $\underline{e}_1 = (a1, \Lambda)$ and $\underline{e}_2 = (\Lambda, d1)$, are both offered after (Λ, Λ) in (ii) and occur consecutively leading to the behaviour described by $(a1, d1)$, in which $a1$ and $d1$ have happened concurrently. They occur consecutively but are not both available after (Λ, Λ) in (i). They are both offered but do not occur consecutively in (iii). Finally, it might be worth noting that the two incomparable vectors sitting at the middle of the lozenge in (ii) represent concurrent execution and have their greatest lower bound (at the bottom of the lozenge) and their least upper bound (on the top of the lozenge) in the language.

It transpires that restricting to an appropriate subset V of V_Σ is of particular importance. This is where we turn our attention to more pragmatic approaches to software design and, in particular, scenario-based specifications.

4 From UML to Component Languages

In this section, we describe the unfolding of scenario-based specifications into component vectors. Our pirmary objective is to capture the *intended* behavior of the component, in terms of its interactions over ports as specifed in sequence diagrams, and use that to restrict to an appropriate subset V of all possible component vectors V_Σ formed over a given component signature Σ.

The approach we advocate uses UML2.0 sequence diagrams - with a non-interleaving semantics, together with a flavour of LSCs - for specifying the *allowed* sequences of events over the multiple ports of a component. The idea is to capture the observed behaviours at each point (graphical position) in a sequence diagram by mapping them onto component vectors.

The embedding of individual components into their environment is described in sequence diagrams by the corresponding lifelines (and associated constructs appearing along a lifeline, e.g. interaction fragments). We are interested in obtaining the vector language part of a component (recall Definition 3.3) from a sequence diagram and therefore we will be focusing on a single lifeline rather than considering all lifelines in the diagram.

In practice a component will appear in several sequence diagrams. UML 2.0 [12] includes Interaction Overview Diagrams (IODs) which generate a single sequence diagram. We therefore assume a single sequence diagram for a component c. (Similarly, this can be achieved in MSCs using *hierarchical* MSCs (hMSCs) [5].)

There are graphical positions along the lifeline of an instance in a sequence diagram that are of particular significance, especially when the diagram is considered in a formal setting. For example, the various event occurrences along the lifeline of a component is what gives rise to its observable behaviours. Similarly to [8], we need to borrow the concept of a *location* from LSCs [2] to capture these significant graphical positions in a UML sequence diagram. *Locations* in LSCs are the points along a lifeline of an instance which correspond to the occurrence of some event (sending/receiving a message). We will always consider along a lifeline (i) an initial location, corresponding to the start of the lifeline, (ii) a final location, corresponding to the end of the lifeline, and (iii) the starting and the ending points of an interaction fragment.

We start by formalising the interaction described in a sequence diagram. For this purpose, we use the notion of a *signature* (including the functions *scope* and *time*) found in [8]. The only difference is that the definition here is adopted to a single lifeline rather than the diagram as a whole, and the extracted information is related to the component signature (Definition 3.1).

Definition 4.1 Given a sequence diagram and a component c participating in the interaction with signature $\Sigma = (P, R, \beta)$, the *component lifeline* is formally given by the tuple

$$Cline = (c, Loc, l_0, Op_\Sigma, SE, RE, Path)$$

where

- c is a component identifier
- Loc is a set of locations on the lifeline corresponding to c
- $l_0 \in Loc$ is the initial location
- Op_Σ is the set of all operations defined on the interfaces of component c, i.e. $Op_\Sigma = \bigcup_{i \in I_\Sigma} \beta(i)$
- $SE \subseteq Loc \times \beta(i)$, $i \in R$ is the set of send events of component c, experienced at its required interfaces
- $RE \subseteq \beta(i) \times Loc$, $i \in P$ is the set of receive events of component c, experienced at its provided interfaces
- $Path$ is a given set of well-formed path terms for the diagram (we will have more to say about $Path$ when we define the function *scope* below)

Further, two auxilliary functions are defined over $Cline$. The first has to do with the timing of locations along a component lifeline and allows us to identify the immediately preceding/succeeding location fo a given location. We do this by defining an injective function

$$time : Loc \rightarrow \mathcal{N}_0$$

which associates each location with a natural number according to its position along the lifeline in the diagram, and is assumed given.

There are certain conditions on this function that formulate our intuitive requirements with respect to timing of locations: (i) the initial location has associated time value 0, i.e. $time(l_0) = 0, l_0 \in Loc$, (ii) since $time$ is injective, all locations along a component lifeline have necessarily different associated time values: $\forall l_1, l_2 \in Loc : l_1 \neq l_2 \;\Rightarrow\; time(l_1) \neq time(l_2)$.

Note that *time* here does not necessarily mean *occurrence time* (though within a **seq** interaction fragment it does), but rather refers to an implicit *visual time* value according to the layout of the diagram. That is to say, locations with different visual time values may still have the same occurrence times, if they belong to a **par** interaction fragment (concurrent locations), or mutually exclusive occurrence times, if they belong to an **alt** interaction fragment (alternative locations).

Consequently, it is important to know in which part of the diagram (in what interaction fragment, if any) a location belongs to. Further concepts introduced in [8], and which we need for our purposes, allow us to talk about the scope of a location. This is done by defining a second auxilliary function,

$$scope : Loc \rightarrow Path$$

which associates each location along a component lifeline *Cline* with a path term. The path term identifies the various compartments of a sequence diagram. We do not define here the grammar for generating path terms. It suffices to understand that path terms are encoded in such a way that it is possible to distinguish between a location that is:

- inside the main diagram (i.e. does not belong to any interaction fragment). Here a path term has the form $\alpha.name$ where α is a path term, possibly the empty term ϵ, and *name* is the name of the sequence diagram, given after the keyword **sd** on the top left corner of the diagram.
- marking the start of an interaction fragment. Here a path term has the form $\alpha.alt(n)$ for an interaction fragment **alt** with $n \in \mathcal{N}^+$ operands, where $\mathcal{N}^+$ denotes the set of natural numbers excluding zero. Similarly, for **par**.
- inside an operand of an interaction fragment. Here a path term has the form $\alpha.par(n)\sharp k$ where $k = 1..n$ indicates that the location is within the k-th operand of a **par** fragment with n operands. Similarly, for **alt**.
- marking the end of an interaction fragment. Here a path term has the form $\alpha.alt(n)$ for an interaction fragment **alt** with $n \in \mathcal{N}^+$ operands. Similarly, for **par**.

These two additional functions, *scope* and *time*, together with Definition 4.1 is what we need in order to formally capture what is described in a sequence diagram. In the remaining sections, we describe how interactions between components specified in a sequence diagram are translated into component languages. This involves unfolding the diagram into component vectors. The basic idea is to map all locations along the lifeline of the component in question onto (a set of) component vectors. This is done by introducing a function *vec_map* from the set of locations to the set of component vectors V_Σ.

For readability, we introduce the function *vec_map* incrementally.

4.1 Sequential scenarios

We start by considering how we can move down a sequence diagram, from one location to the next along the component lifeline in question, whilst mapping each location to (a set of) component vectors.

The rationale behind doing this is the following. The component vectors associated with each location are obtained from the vectors of the *immediately preceding* location, by concatenating the event (if any) associated with the location being considered with the sequence of events appearing on the corresponding coordinate of the component vectors of the immediately preceding location. By convention, the initial location of a component lifeline is mapped onto the empty vector $\underline{\Lambda}_\Sigma$.

There are some cases however, in which this central idea does not apply. In particular, the end location of a **par** interaction fragment as well as the end location of an **alt** interaction fragment need to be treated differently. This is because we have to take into account the various execution sequences that may arise when encountering these interaction fragments. Furthermore, the first location of each operand of an **alt** or **par** interaction fragment has to be considered in relation to the start location of the **alt** or **par** fragment rather than its immediately preceding location. This is due to the fact that the *visual* time does not correspond to the *occurrence* time for the locations of these interaction fragments. We will see exactly how these special cases are addressed in the following sections.

At this stage it suffices to understand that the basic definition for moving down the sequence diagram does not apply to the following locations:

- the end location of an **alt** interaction fragment with n operands,
$$X_1 = \{l \in Loc | scope(l) = \alpha.alt(n)\}$$
Similarly, for **par** we have $X_2 = \{l \in Loc | scope(l) = \alpha.par(n)\}$
- the first location of each operand in an **alt** interaction fragment with n operands,
$$Y_1 = \{l_k \in Loc, 1 \leq k \leq n \mid scope(l_k) = \alpha.alt(n)\sharp k \wedge time(l_k) = time(l') + 1 \wedge \\ \wedge (scope(l') = \alpha.alt(n)\sharp(k-1) \vee scope(l') = \alpha.alt(n))\}$$
In further explanation of the notation, l_k is the first location of the k-th operand if the previous location, l', is a location of the $(k-1)$ operand or the start location of the fragment. Similarly, for **par** we have
$$Y_2 = \{l_k \in Loc, 1 \leq k \leq n \mid scope(l_k) = \alpha.par(n)\sharp k \wedge time(l_k) = time(l') + 1 \wedge \\ \wedge (scope(l') = \alpha.par(n)\sharp(k-1) \vee scope(l') = \alpha.par(n))\}$$

Let Z denote the union of the sets X_1, X_2, Y_1, Y_2, hence $Z = X_1 \cup X_2 \cup Y_1 \cup Y_2$. Then, we may capture the rest of the locations along a lifeline in the set, $Loc' = Loc \setminus Z$. We may now give a basic definition that describes how locations in Loc' are mapped onto component vectors. $|Y|$ is used to denote the cardinality of the set Y.

Definition 4.2 Suppose that Σ is the signature of a component c represented in a sequence diagram by a lifeline $Cline = (c, Loc, l_0, Op_\Sigma, SE, RE, Path)$. We define an injective function,

$$vec\_map : Loc' \rightarrow \wp(V_\Sigma)$$

given by

- $vec\_map(l_0) = \Lambda_\Sigma$
- $vec\_map(l) = \{v_l^{(1)}, v_l^{(2)}, ..., v_l^{(m)}\}$, where $m = |vec\_map(\tilde{l})|$ and $\tilde{l} \in Loc$ such that $time(\tilde{l}) = time(l) - 1$ and for each j, $1 \leq j \leq m$,
$$v_l^{(j)} = (v_{l_1}^{(j)}, v_{l_2}^{(j)}, ..., v_{l_n}^{(j)})$$

where n is the number of interfaces of c and each coordinate is given by

$$v_{l_i}^{(j)} = \begin{cases} v_{\tilde{l}_i}^{(j)}.e & , \quad ((l,e) \in SE \vee (e,l) \in RE) \wedge e \in \beta(i) \\ v_{\tilde{l}_i}^{(j)} & , \quad \text{otherwise} \end{cases}$$

where $1 \leq i \leq n$.

It can be seen that the component vectors associated with l are obtained by the m component vectors of the previous location $\tilde{l}$ (identified using the function *time*) and the coordinates of each vector are obtained from those of $\tilde{l}$ by concatenating the event associated with l, on the appropriate coordinate (on the coordinate corresponding to the port on which e occured).

Notice that $\underline{v}_l^{(k)}$ denotes the k-th (out of m) component vector associated with location l while $v_{l_i}^{(k)}$ denotes the i-th (out of n) coordinate of the k-th component vector of location l. It might also be worth pointing out that each location l is mapped onto *a set of* component vectors. The cardinality of the set is that of the set of component vectors associated with its previous location. This might seem somewhat counter-intuitive at this stage, but is necessary because the previous location might have been mapped onto more than one vector. This will be the case when any of the locations preceding l is the end location of a **par** or an **alt** interaction fragment.

Example 4.3 Consider the component $c1$ of Fig. 1. Its signature is given by $\Sigma_{c1} = (I_{c1}, O_{c1}, \beta_{c1})$ where $I_{c1} = \{p_{11}, p_{13}\}$, $O_{c1} = \{p_{12}\}$ and let $\beta_{c1}(p_{11}) = \{a1, a2\}$, $\beta_{c1}(p_{12}) = \{b1\}$, $\beta_{c1}(p_{13}) = \{d1, d2\}$. We demonstrate how the locations appearing along its lifeline in the sequence diagram of Fig. 3 are mapped onto component vectors. We write (x, y, z) for $\underline{v} \in V_{\Sigma_{c1}}$ where the first coordinate is allocated to port p_{11}, the second to p_{12} and the third to p_{13}.

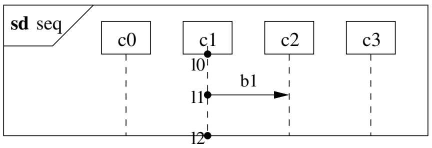


Fig. 3. Sequential locations

$l0$ is the initial location and by definition is mapped onto the empty vector $\Lambda_{\Sigma_{c1}}$. Thus, $vec\_map(l0) = (\Lambda, \Lambda, \Lambda)$.

The next location visited is $l1$, since $time(l1) = time(l0) + 1$. By Definition 4.2 we have that $vec\_map(l1) = v_{l1}$ (since $m = |vec\_map(l0)|$ and $l0$ is the immediately preceding location). The vector v_{l1} is given by $v_{l1} = (v_{l1_1}, v_{l1_2}, v_{l1_3})$ where

- $v_{l1_1} = v_{l0_1} = \Lambda$ since $(l1, b1) \in SE_{c1}$ but $b1 \notin \beta_{c1}(p_{11})$
- $v_{l1_2} = v_{l0_2}.b1 = b1$ since $(l1, b1) \in SE_{c1} \wedge b1 \in \beta_{c1}(p_{12})$
- $v_{l1_3} = v_{l0_3} = \Lambda$ since $(l1, b1) \in SE_{c1}$ but $b1 \notin \beta_{c1}(p_{13})$

Hence, $vec\_map(l1) = (\Lambda, b1, \Lambda)$. $\square$

4.2 Alternative scenarios

In this section, we extend Definition 4.2 to address (a) locations that mark the end of an **alt** fragment, i.e. $l \in X_1$, and (b) the first location of each operand in an **alt** fragment, i.e. $l \in Y_1$. We motive these extensions as follows.

An **alt** interaction fragment in a sequence diagram represents choice of behaviour, the choice being between the behaviours described by each of its operands. Recall that at most one of the operands executes [12]. However, the set of execution sequences of the choice is the union of the execution sequences of the operands. Thus, at the end of an **alt** fragment with n operands in a diagram there are n different behaviours, one for each operand. Each of these behaviours or execution sequences arises as a continuation of the start location of the **alt** fragment.

The location marking the start of an **alt** interaction fragment is identified using the function *scope* and the corresponding path term (e.g. $scope(l) = \alpha.alt(n)$). This use of *scope* allows us to determine when we come across an **alt** fragment along a lifeline $Cline$ in a sequence diagram.

The n different scenarios the corresponding component may engage in are continuations of the start location of the **alt** fragment. To reflect this, the component vectors of the first location of each operand are obtained based on those of the start location instead of their immediately preceding location.

The first location of each operand is identified by the combined use of the functions *time* and *scope* as follows. l is the first location of the k-th operand in an **alt** interaction fragment with n operands iff $time(l) = time(\tilde{l}) + 1$ where $\tilde{l}$ is such that $scope(\tilde{l}) = \alpha.alt(n)\sharp(k-1) \vee scope(\tilde{l}) = \alpha.alt(n)$. Informally, this says that l is the first location of the k-th operand if its previous location $\tilde{l}$ (given by function *time*) belongs to the previous operand (given by function *scope*) or is the start location of the **alt** fragment (identified again using function *scope*).

The first location of each operand is mapped onto component vectors based on the construction given in the following definition.

Definition 4.4 Let c be a component, with signature $\Sigma = (P, R, \beta)$, represented in a sequence diagram by a lifeline $Cline = (c, Loc, l_0, Op_\Sigma, SE, RE, Path)$ and let $l \in Loc$ be the first location of the k-th operand of an **alt** interaction fragment with n operands on $Cline$. Then, $vec\_map(l) = \{v_l^{(1)}, v_l^{(2)}, ..., v_l^{(m)}\}$ where $m = |vec\_map(\tilde{l})|$ and $\tilde{l} \in Loc$ such that $scope(\tilde{l}) = \alpha.alt(n)$ in which case, for each j, $j = 1..m$,

$$v_l^{(j)} = (v_{l_1}^{(j)}, v_{l_2}^{(j)}, ..., v_{l_n}^{(j)})$$

where n is the number of interfaces of c and each coordinate is given by

$$v_{l_i}^{(j)} = \begin{cases} v_{\tilde{l}_i}^{(j)}.e & , \quad ((l,e) \in SE \vee (e,l) \in RE) \wedge e \in \beta(i) \\ v_{\tilde{l}_i}^{(j)} & , \quad \text{otherwise} \end{cases}$$

where $1 \leq i \leq n$.

This definition applies to locations $l \in Y_1$ with respect to the discussion prior to Definition 4.2. Note that the only difference with Definition 4.2 is that in an

alt interaction fragment the component vectors associated with the first location of each operand are considered in relation to the start location of the **alt** fragment instead of its preceding location. (The first location of the first operand is still considered in relation to its preceding location, but only because this happens to be the start location (of the **alt** fragment).)

At the end of an **alt** fragment we need to capture the fact that there are n alternative scenarios the component may have engaged in. We do this by associating the end location of an **alt** with the component vectors of the last location of each operand. This is formally put in the following definition.

Definition 4.5 Let c be a component, with signature $\Sigma = (P, R, \beta)$, represented in a sequence diagram by a lifeline $Cline = (c, Loc, l_0, Op_\Sigma, SE, RE, Path)$ and let $l \in Loc$ be the end location of an **alt** interaction fragment with n operands (i.e. $scope(l) = \alpha.alt(n)$). Then, the component vectors associated with l are given by

$$vec\_map(l) = \bigcup_{k=1}^{n} vec\_map(\hat{l}_k)$$

where $\hat{l}_k$, each k, is such that $scope(\hat{l}_k) = \alpha.alt(n)\sharp k \wedge time(\hat{l}_k) = time(\tilde{l}) - 1$ where $\tilde{l} \in Loc$ is such that $scope(\tilde{l}) = \alpha.alt(n)\sharp(k+1) \vee scope(\tilde{l}) = \alpha.alt(n)$

This definition applies to location $l \in X_1$ with respect to the discussion prior to Definition 4.2. Notice that the last location $\hat{l}_k$, each k, of each operand is identified in similar fashion to that for the first location of each operand.[4]

Example 4.6 We extend the previous example (Example 4.3) with the interactions shown in Fig. 4 to demonstrate the treatment of alternative locations in our approach.

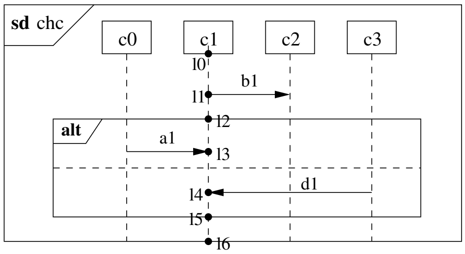


Fig. 4. Alternative locations

The location $l2$ has been mapped onto $v_{l2} = (v_{l2_1}, v_{l2_2}, v_{l2_3}) = (\Lambda, b1, \Lambda)$. This is the start location of an **alt** fragment with 2 operands, since $scope(l2) = chc.alt(2)$.

The next location considered (not necessarily the one to be visited next, as explained before) is $l3$. This is the first location of the 1st operand, since $time(l3) = time(l2) + 1$ and $scope(l2) = chc.alt(2)$ while $scope(l3) = chc.alt(2)\#1$. This may

[4] Note that there might be some duplication (duplicate component vectors between last location per operand and end location of **alt**), but there is good reason for it. The component vectors of the last location per operand feed into the resulting component language V (cf Definition 4.10) while the component vectors of the end location are used for obtaining the component vectors associated with the location below it (the one visited next).

be expressed more succinctly as $l3 \in Y_1$. Thus, its associated vectors are given by Definition 4.4 as follows.

$vec\_map(l3) = \underline{v}_{l3}$ since $m = |vec\_map(l2)|$ and $scope(l2) = chc.alt(2)$. The component vector $\underline{v}_{l3}$ is given by $\underline{v}_{l3} = (v_{l3_1}, v_{l3_2}, v_{l3_3})$ where

- $v_{l3_1} = v_{l2_1}.a1 = a1$ since $(a1, l3) \in RE_{c1} \wedge a1 \in \beta_{c1}(p_{11})$
- $v_{l3_2} = v_{l2_2} = b1$ since $(a1, l3) \in RE_{c1}$ but $a1 \notin \beta_{c1}(p_{12})$
- $v_{l3_3} = v_{l2_3} = \Lambda$ since $(a1, l3) \in RE_{c1}$ but $a1 \notin \beta_{c1}(p_{13})$

Hence, $vec\_map(l3) = (v_{l3_1}, v_{l3_2}, v_{l3_3}) = (a1, b1, \Lambda)$.

The next location considered is $l4$ which is the first location of the 2nd operand. Hence, again we apply definition 4.4 and we have $vec\_map(l4) = \underline{v}_{l4}$ since $m = |vec\_map(l2)|$ and $scope(l2) = chc.alt(2)$. The component vector $\underline{v}_{l4}$ is given by $\underline{v}_{l4} = (v_{l4_1}, v_{l4_2}, v_{l4_3})$ where

- $v_{l4_1} = v_{l2_1} = \Lambda$ since $(d1, l3) \in RE_{c1}$ but $d1 \notin \beta_{c1}(p_{11})$
- $v_{l4_2} = v_{l2_2} = b1$ since $(d1, l3) \in RE_{c1}$ but $d1 \notin \beta_{c1}(p_{12})$
- $v_{l3_3} = v_{l2_3}.d1 = d1$ since $(d1, l3) \in RE_{c1} \wedge d1 \in \beta_{c1}(p_{13})$

Hence, $vec\_map(l4) = (v_{l4_1}, v_{l4_2}, v_{l4_3}) = (\Lambda, b1, d1)$.

The next location is $l5$ which is the end location of the **alt** fragment, since $scope(l5) = chc.\overline{alt}(2)$. Thus, its component vectors are given by Definition 4.5 as follows.

$vec\_map(l5) = vec\_map(l3) \cup vec\_map(l4)$ since $l3$ is the last location of the 1st operand and $l4$ of the 2nd. Hence, $vec\_map(l5) = \{\underline{v}_{l5}^{(1)}, \underline{v}_{l5}^{(2)}\} = \{(a1, b1, \Lambda), (\Lambda, b1, d1)\}$. $\square$

4.3 Parallel scenarios

In this section, we extend Definition 4.2 to address locations that mark the end of a **par** fragment, i.e. $l \in X_2$, and the first location of each operand in a **par** fragment, i.e $l \in Y_2$. We motive these extensions and give an account of our formal semantics for the **par** interaction fragment in UML 2.0.

Simplifying somewhat, instead of considering that locations from different operands are reached in either order, which one would find in an interleaving approach, we consider three cases: a) one location is reached first, b) the other location is reached first and c) both locations are reached at exactly the same time. Another way of expressing this is by saying that locations from different operands are reached in *no particular* order. This perception of parallelism is rooted in the formal treatment of concurrency, via an independence relation [13], within our theoretical framework. Independent events that are enabled, and occur consecutively, are concurrent.

Since the event occurrences of different operands are independent of each other, the resulting behaviours of each operand arise as continuations of the start location of the **par** fragment. In a fashion similar to the **alt** fragment, we thus need to identify the start location of **par** and ensure that the first location of each operand is mapped onto component vectors based on the vectors of the start location of **par** rather than the previous location.

The start location of **par** is identified using the function *scope* while the first location of each operand is identified through the combined use of *scope* and *time*, as before (Section 4.2). The following definition addresses such locations and is essentially an adaptation of Definition 4.4 for **par**.

Definition 4.7 Let c be a component, with signature $\Sigma = (P, R, \beta)$, represented in a sequence diagram by a lifeline $Cline = (c, Loc, l_0, Op_\Sigma, SE, RE, Path)$ and let $l \in Loc$ be the first location of the k-th operand of a **par** interaction fragment with n operands on $Cline$. Then, $vec\_map(l) = \{\underline{v}_l^{(1)}, \underline{v}_l^{(2)}, ..., \underline{v}_l^{(m)}\}$ where $m = |vec\_map(\tilde{l})|$ and $\tilde{l} \in Loc$ such that $scope(\tilde{l}) = \alpha.par(n)$ in which case, for each j, $j = 1..m$,

$$\underline{v}_l^{(j)} = (v_{l_1}^{(j)}, v_{l_2}^{(j)}, ..., v_{l_n}^{(j)})$$

where n is the number of interfaces of c and each coordinate is given by

$$v_{l_i}^{(j)} = \begin{cases} v_{\tilde{l}_i}^{(j)}.e & , \quad ((l,e) \in SE \vee (e,l) \in RE) \wedge e \in \beta(i) \\ v_{\tilde{l}_i}^{(j)} & , \quad \text{otherwise} \end{cases}$$

where $i = 1..n$.

This definition applies to locations $l \in Y_2$ with respect to the discussion prior to Definition 4.2.

All locations that appear in a **par** fragment other than the first of each operand and the end location belong to the set Loc' and they are mapped onto component vectors following Definition 4.2. It remains to address the end location of **par**.

Once the end location of **par** is reached, we need to consider the three cases, discussed earlier, for each location appearing within **par**. This is to reflect the fact that event occurrences appearing in different operands of **par** are effectively *unordered* (in parallel). We formulate each case below.

The following set comprises locations appearing within a **par** fragment.

$$Loc_{par} = \{l \in Loc \mid scope(l) = \alpha.par(n)\sharp k, k = 1..n\}$$

For each location $l \in Loc_{par}$ such that $scope(l) \neq \alpha.par(n)\sharp n$ determine,

- $vec\_map(l)^I = vec\_map(l)$.
  This gives the component vectors associated with l, when l is reached first.
- $vec\_map(l)^{II} = \bigcup_{s=1}^{|X|} vec\_map(\tilde{l}_s)$ where
  $X = \{\tilde{l} \in Loc_{par} \mid scope(\tilde{l}) = \alpha.par(n)\sharp j,\ (k+1) \leq j \leq n\}$
  This gives the component vectors associated with l when all other locations $\tilde{l}$ of the succeeding operands are reached first (before reaching l).
- $vec\_map(l)^{III} = \{v_{l,\tilde{l}}^{(j)} = v_l^{(j)} \sqcup v_{\tilde{l}}^{(j)} \mid \forall \tilde{l} : scope(\tilde{l}) = \alpha.par(n)\sharp j, (k+1) \leq j \leq n\}$
  where $1 \leq j \leq m$ and $m = |vec\_map(\hat{l})|$ where $\hat{l}$ is such that $scope(\hat{l}) = \alpha.par(n)$.
  This gives the component vectors associated with location l of the k-th operand when it is reached at exactly the same time with another location $\tilde{l}$ from a different (and succeeding) operand. The superscript j runs through the m component vectors associated with location l, where m is as before (Definition 4.2).

The operation $\sqcup$ on component vectors gives their least upper bound and was formally defined in in Section 3. The resulting component vector is obtained by comparing the coordinates pairwise and keeping the one whose sequence is of greater length. This is demonstrated in Example 4.9 below.

By considering the above three cases, each location l in a **par** fragment is mapped onto three (sets of) component vectors, $vec\_map(l)^{I}$, $vec\_map(l)^{II}$ and $vec\_map(l)^{III}$. Now the end location of **par** is associated with the component vectors (including all three cases) corresponding to each location appearing within the fragment. This is formally put in the following definition.

Definition 4.8 Let c be a component, with signature $\Sigma = (P, R, \beta)$, represented in a sequence diagram by a lifeline $Cline = (c, Loc, l_0, Op_\Sigma, SE, RE, Path)$ and let $l \in Loc$ be the end location of a **par** interaction fragment with n operands. Then, the component vectors associated with l are given by,

$$vec\_map(l) = \bigcup_{r=1}^{|Y|} vec\_map(l_r)^x, x = I, II, III$$

where $Y = \{l \in Loc_{par} \mid scope(l) = \alpha.par(n)\sharp k, 1 \leq k \leq (n-1)\}$.

The definition maps the end location of **par** onto the union of all component vectors associated with locations of the first $n-1$ operands, via cases I, II and III (x runs through I, II, III while r runs through all locations of the first $n-1$ operands). Note that we do not consider locations of the last operand in the definition because they will have already been considered in going through the preceding $n-1$ operands. This definition is an extension of Definiton 4.2 that applies to locations $l \in X_2$.

Example 4.9 We extend Example 4.3 with the interactions shown in Fig. 5 to demonstrate the treatment of concurrent locations in our approach.

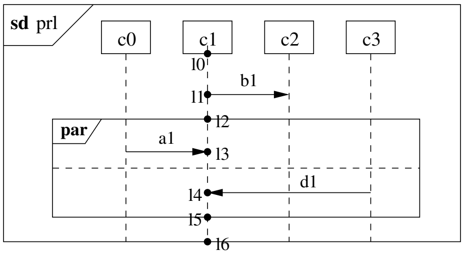


Fig. 5. Concurrent locations

The location $l2$ has been mapped onto $v_{l2} = (v_{l2_1}, v_{l2_2}, v_{l2_3}) = (\Lambda, b1, \Lambda)$. This is the start location of a **par** fragment with 2 operands, since $scope(l2) = prl.par(2)$.

The next location considered is $l3$. It is the first location of the 1st operand, since $time(l3) = time(l2) + 1$ and $scope(l2) = prl.par(2)$ while $scope(l3) = prl.par(2)\#1$. This may be expressed more succinctly as $l3 \in Y_2$. Thus, its associated vectors are given by Definition 4.7 as follows.

$vec\_map(l3) = v_{l3}$ since $m = |vec\_map(l2)|$ and $scope(l2) = prl.par(2)$. The component vector v_{l3} is given by $v_{l3} = (v_{l3_1}, v_{l3_2}, v_{l3_3})$ where

- $v_{l3_1} = v_{l2_1}.a1 = a1$ since $(a1, l3) \in RE_{c1} \wedge a1 \in \beta_{c1}(p_{11})$
- $v_{l3_2} = v_{l2_2} = b1$ since $(a1, l3) \in RE_{c1}$ but $a1 \notin \beta_{c1}(p_{12})$
- $v_{l3_3} = v_{l2_3} = \Lambda$ since $(a1, l3) \in RE_{c1}$ but $a1 \notin \beta_{c1}(p_{13})$

Hence, $vec\_map(l3) = (v_{l3_1}, v_{l3_2}, v_{l3_3}) = (a1, b1, \Lambda)$.

The next location considered is $l4$. It is the first location of the 2nd operand, since $time(l4) = time(l3) + 1$ and $scope(l3) = prl.par(2)\#1$ while $scope(l4) = prl.par(2)\#2$. This may be expressed more succinctly as $l4 \in Y_2$. Thus, its associated vectors are given by Definition 4.7 as follows.

$vec\_map(l4) = \underline{v}_{l4}$ since $m = |vec\_map(l2)|$ and $scope(l2) = prl.par(2)$. The component vector $\underline{v}_{l4}$ is given by $\underline{v}_{l4} = (v_{l4_1}, v_{l4_2}, v_{l4_3})$ where

- $v_{l4_1} = v_{l2_1} = \Lambda$ since $(d1, l4) \in RE_{c1}$ but $d1 \notin \beta_{c1}(p_{11})$
- $v_{l4_2} = v_{l2_2} = b1$ since $(d1, l4) \in RE_{c1}$ but $d1 \notin \beta_{c1}(p_{12})$
- $v_{l4_3} = v_{l2_3}.d1 = d1$ since $(d1, l4) \in RE_{c1} \wedge d1 \in \beta_{c1}(p_{13})$

Hence, $vec\_map(l4) = (v_{l4_1}, v_{l4_2}, v_{l4_3}) = (\Lambda, b1, d1)$.

The next location considered is $l5$. This is the end location of **par**, since $scope(l5) = prl.par(2)$. Thus, its component vectors are given by Definition 4.8. First, we determine the 3 cases for each location within **par**. In fact, we have seen that we do need to consider the locations of the last operand. So, in our example it suffices to determine the 3 cases for location $l3$ only.

- $vec\_map(l3)^{I} = vec\_map(l3) = (a1, b1, \Lambda)$
- $vec\_map(l3)^{II} = vec\_map(l4) = (\Lambda, b1, d1)$
- $vec\_map(l3)^{III} = \underline{v}_{l3,l4}$ since $m = |vec\_map(l2)| = 1$ where $l2$ is the start location of **par**, and $l4$ is the only location such that $scope(l4) = prl.par(2)\#2$. The corresponding component vector is given by

$$\underline{v}_{l3,l4} = \underline{v}_{l3} \sqcup \underline{v}_{l4} = (a1, b1, \Lambda) \sqcup (\Lambda, b1, d1) = (a1, b1, d1)$$

We may now obtain the vectors for **par** by applying Definition 4.8, $vec\_map(l5) = vec\_map(l3)^{I} \cup vec\_map(l3)^{II} \cup vec\_map(l3)^{III} = \{(a1, b1, \Lambda), (\Lambda, b1, d1), (a1, b1, d1)\}$

4.4 Obtaining the component language

So far we have seen how locations along a lifeline in a sequence diagram can be mapped onto component vectors. These provide a *snapshot* of component behaviour in that they show what events have occurred on the component's interfaces, starting from the top of the diagram and subsequently moving downwards. The component language V (of Definition 3.3) of a component c can thus be obtained in a straightforward manner by taking the union of all component vectors associated with a location along the corresponding lifeline.

The only locations that do not adhere to this rationale are locations within a **par** interaction fragment. We have already seen that the events associated with these locations are not ordered in any way (including simultaneity). This is captured in the vector mapping of the end location of this fragment. Thus, in obtaining the component language we include all vectors associated with a location along the corresponding lifeline, except for those appearing within a **par** fragment, for which we only include the vectors of its end location. This is formally put in the following definition.

Definition 4.10 Let Σ be the signature of a component c represented in a sequence diagram by a lifeline $Cline$. Then, its corresponding component language V is given by

$$V = \{vec\_map(l) \mid l \in Loc \setminus Loc_{par}\}$$

This says that the component language V comprises the sets $vec\_map(l)$ for each location l along $Cline$, providing the location is not within a **par** interaction fragment.

Based on the postulate of the definition and the formal construction given for vec_map it may be shown that the resulting set V is a well-defined subset of the set of all possible component vectors V_Σ, formed over a signature Σ.

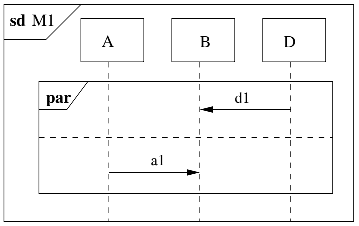
Proposition 4.11 *The set V obtained following the construction given in Definition 4.10 is a well-defined subset of V_Σ.*

5 Conclusions and Related Work

In this paper, we have described a formal translation of the interactions given in a UML sequence diagram into component languages. This tuples-based representation of interactive behaviour allows us to consider multiple access points for the participating instances (components here). Concurrency is handled in a non-interleaving manner [13]. Events occur sequentially on a single access point (same port) but they may do so concurrently on distinct access points (different ports). The true-concurrent semantics for the **par** interaction fragment allows us to faithfully express such behaviours.

[14] translate scenarios into an FSP specification and use that to generate a labelled transition system (LTS) model of each component. The LTS semantics of FSP impose an interleaving interpretation of concurrency. LTSs are also used in [4] where MSCs are used for verifying component properties.

Fundamentally, not all properties for concurrent systems can be expressed and verified without considering true-concurrency. Following an interleaving interpretation, it is not possible (to the best of our knowledge) to differentiate between the behaviours described in the sequence diagrams of Fig. 6. This may not be of concern



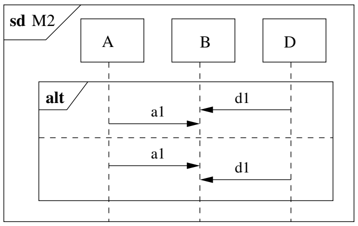


Fig. 6. The **par** construct and a possible interpretation

when all we are interested in is that both $d1$ and $a1$ have occurred at the end of the scenario - this incurs an implicit synchronisation point at the end of diagram M1. If, however, we want to include (or, indeed, cannot exclude) the possibility that $d1$ and $a1$ occur at exactly the same time then diagram M2 no longer describes the actual behaviour of component B.

[6] translate scenarios using timed Büchi automata. This approach considers the case that concurrent events may occur at exactly the same time. Concurrency is not expressed as an explicit property of the corresponding automata however, but rather in terms of their timing properties.

Our formal definition of (a lifeline of) a UML sequence diagram (recall Definition 4.1) is an adaptation of the approach taken in defining the notion of a *signature* of a sequence diagram (including *scope* and *time*) in [8], which first appeared in [7]. This work goes on to translate scenarios using another model of true-concurrency, namely labelled prime event structures. The true-concurrent semantics presented here is given in terms of vector languages and these, under certain conditions, have a transition structure that gives rise to a class of automata.

In [11] we have described how component languages in our approach generate *concurrent automata*, in which concurrency is expressed explicitly as a property of the underlying transition structure. The connection is based on the order-theoretic properties, namely *discreteness* and *local left-closure*, of component languages. In this paper we have described how component languages can be obtained directly from UML sequence diagrams. An interesting consequence of this extension is that we can verify the sequence diagram against the state-based model directly through model checking.

Furthermore, in checking the obtained language against discreteness and local left-closure we identify missing behaviours which may indicate emergent behaviour (e.g. race conditions, as shown in [9]) or were simply unthought in design. In this sense our approach supports the gradual elaboration of scenario-based specifications to more comprehensive ones. Note that discreteness and local left-closure are preserved under composition as shown in [10].

We are currently investigating the use of the language-based approach as a model for a distributed temporal logic addressing concurrency explicitly. Work is in progress on a hierarchical logic [1] interpreted over concurrent automata [11]. This extended framework can reflect properties of interactions from both the local viewpoint (e.g. delegation of requests to internal subcomponents, as a result of composition [10]) and the overall system viewpoint. These different viewpoints can be particularly useful in a purely distributed setting, such as long-running transactions over P2P networks, where there is no centralised control of the interaction.

Acknowledgement

We would like to thank Juliana Küster Filipe Bowles for valuable discussion, especially on the signature of a UML 2.0 sequence diagram.

References

[1] J. Küster-Filipe Bowles and S. Moschoyiannis. Concurrent Logic and Automata Combined: a Semantics for Components. In *CONCUR 2006 - FOCLASA'06*, ENTCS. Elsevier, 2007. To appear.

[2] W. Damm and D. Harel. LCSs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[3] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge Mathematical Textbooks, Cambridge University Press, 1990.

[4] B. Finkbeiner and I. Krüger. Using Message Sequence Charts for Component-Based Formal Verification. In *OOPSLA 2001 - SAVCBS'01*, Report ISU TR 01-09a, pages 32–41, 2001.

[5] ITU-T. *Recommendation Z.120: Message Sequence Chart (MSC)*. ITU-T, October 1996.

[6] J. Klose and H. Wittke. An Automata-based Interpretation of Live Sequence Charts. In *ETAPS 2001 - TACAS'01*, volume 2031 of *LNCS*, pages 512–527. Springer, 2001.

[7] J. Küster-Filipe. Modelling Concurrent Interactions. In *Proceedings of Algebraic Methodology and Software Technology (AMAST 2004)*, volume 3116 of *LNCS*, pages 304–318. Springer, 2004.

[8] J. Küster-Filipe. Modelling Concurrent Interactions. *Theoretical Computer Science*, 351(2):203–220, 2006.

[9] S. Moschoyiannis. *Specification and Analysis of Component-Based Software in a Concurrent Setting*. PhD thesis, University of Surrey, 2005.

[10] S. Moschoyiannis and M. W. Shields. A Set-Theoretic Framework for Component Composition. *Fundamenta Informaticae*, 59(4):373–396, 2004.

[11] S. Moschoyiannis, M. W. Shields, and P. J. Krause. Modelling Component Behaviour Using Concurrent Automata. In *ETAPS 2005 - FESCA'05*, volume 141 of *ENTCS*, pages 199–220. Elsevier, 2005.

[12] OMG. *Unified Modeling Language: Superstructure, version 2.0*. OMG document formal/05-07-04, available from http://www.omg.org, August 2005.

[13] M. W. Shields. *Semantics of Parallelism*. Springer-Verlag London, 1997.

[14] S. Uchitel, J. Kramer, and J. Magee. Synthesis of Behavioral Models from Scenarios. *IEEE Transactions on Software Engineering*, 29(2):99–115, 2003.

[15] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala Component Model for Consumer Electronics. *IEEE Transactions on Computers*, 33(3):78–85, 2000.

[16] G. Winskel and M. Nielsen. Models for Concurrency. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science, vol. 4, Semantic Modelling*, pages 1–148. Oxford Science Publications, 1995.