Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 175 (2007) 135–151
www.elsevier.com/locate/entcs

Concurrent Logic and Automata Combined: A Semantics for Components[1]

J. K. F. Bowles[2] and S. Moschoyiannis[3]

School of Computer Science, The University of Birmingham, Edgbaston, Birmingham B15 2TT, UK

Abstract

In this paper, we describe a true-concurrent hierarchical logic interpreted over concurrent automata. Concurrent automata constitute a special kind of asynchronous transition system (ATS) used for modelling the behaviour of components as understood in component-based software development. Here, a component-based system consists of several interacting components whereby each component manages calls to and from the component using ports to ensure encapsulation. Further, a component can be complex and made of several simpler interacting components. When a complex component receives a request through one of its ports, the port delegates the request to an internal component. Our logic allows us to describe the different views we can have on the system. For example, the overall component interactions, whether they occur sequentially, simultaneously or in parallel, and how each component internally manages the received requests (possibly expressed at different levels of detail). Using concurrent automata as an underlying formalism we guarantee that the expressiveness of the logic is preserved in the model. In future work, we plan to integrate our *truly*-concurrent approach into the Edinburgh Concurrency Workbench.

Keywords: vector semantics, order structure, independence, concurrency

1 Introduction

Modern systems rely increasingly on combining concurrent, distributed, mobile, reconfigurable and heterogeneous components. New models, architectures, languages, and verification techniques are necessary to cope with the complexity induced by the demands of today's software development. In addition, there is an increased interest in theoretical models and foundations for component-based systems taking into account aspects such as component composition, concurrency, mobility, interaction, and emergent behaviour. We address some of these issues, by providing a mathematical framework for modelling and reasoning about component-based systems.

[1] This work was partially supported by the School of Computer Science Ramsay fund.

[2] Email:J.Bowles@cs.bham.ac.uk

[3] Email:S.Moschoyiannis@cs.bham.ac.uk

doi:10.1016/j.entcs.2007.03.008

We describe a class of automata, the so-called *concurrent automata*, which can be used for modelling component behaviour in terms of event occurrences on ports. Concurrent automata are a special kind of asynchronous transition system (ATS) [19] where concurrency is given in the form of an independence relation (whereby independent events that occur consecutively are concurrent).

We start by considering a language-based representation of component behaviour in which we can capture (i) the event occurrences on each port; (ii) the temporal relations between events occurring on distinct ports; and (iii) the contribution of each port in the overall behaviour of the component. The notion of concurrency in these languages takes up on ideas from Mazurkiewicz trace languages [10] and is again based on an independence relation: events occurring on distinct ports are independent and *can* happen concurrently; events associated with the same port are causally dependent and *must* be ordered in time.

We then describe concurrent automata whose transition structure is set up in such a way that it embodies the properties that ensure the well-formedness of the language-based behavioural description and allow us to express concurrency explicitly. In this way, we have a language-based description of component behaviour and a class of automata which generate such languages.

Given the semantic model presented for modelling components, we introduce a true-concurrent hierarchical logic for describing properties over such systems. The logic, here referred to as *concurrent logic*, is in fact based on an existing distributed temporal logic MDTL [8] but adapted to the new semantic model. This combination provides the right setting for a *truly-concurrent* extension of the Edinburgh Concurrency Workbench. In the context of component-based systems, our logic allows us to describe the different views we can have on these systems. For example, the overall component interactions, whether they occur sequentially, simultaneously or in parallel, and how each component internally manages the received requests (possibly expressed at different levels of detail). To the best of our knowledge, our logic is novel and provides a powerful and expressive means to describe properties over component-based systems.

The paper is structured as follows. In Section 2, we give a brief overview of two diagrams in UML 2.0 used for component-based modelling. In Section 3 we introduce our underlying semantic model, namely concurrent automata. In Section 4, we introduce our concurrent logic. The paper finishes with some concluding remarks and ideas for future work.

2 Components in UML 2.0

A component is a modular part of of a system design that provides a coherent set of services through well-defined interfaces. Hence, in addition to an internal implementation consisting of one or more classifiers that realise its behaviour, a component also has an external specification in the form of one or more provided and required interfaces.

In this section we give a very brief description of two UML 2.0 diagrams used

for component-based modelling: class diagrams and sequence diagrams.

Class diagrams are used in UML to describe the structural view of a system, i.e., the classes that make up a system and how they are related (through associations or dependencies). In UML 2.0 [18], a component is a special kind of a *structured classifier*, i.e., a class which has some internal structure made of parts. A class declares properties shared by all instances of the class, namely attributes and operations. Some of the operations of a class are public and constitute the services the class offers to its clients. The public operations form an interface to the class. Notice that in UML 2.0, an interface consists not only of a set of operations, but also a set of (virtual) attributes. These attributes allow us to specify the contracts for interface operations more accurately. A class can offer one or more interfaces, and require one or more interfaces from other classes. To avoid a client of a structured class being able to invoke public services of a part, the structured class 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 structured class via interfaces in a state-based way. This can be further exploited in providing a direct model of the state of the interaction with the component so that constraints on the call sequences can be expressed.

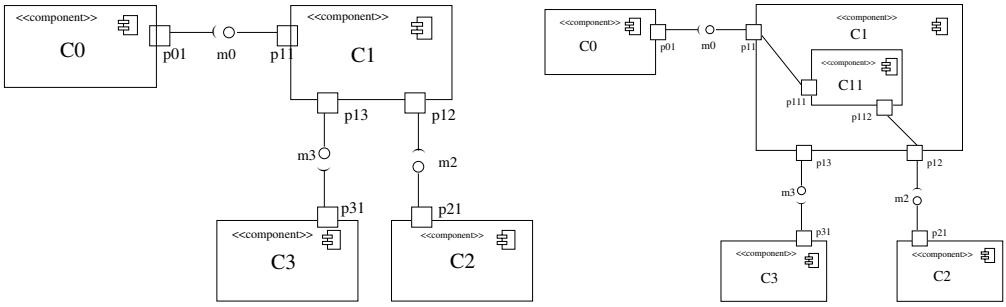

Fig. 1. Component architecture showing different levels of detail.

Fig. 1 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`. Fig. 1(right) shows the internal structure of `C1` using constructs from composite structure diagrams in UML 2.0. It can be seen that port `p11` delegates the received request `m0` to the internal or *local* port `p111` and, similarly, port `p112` requires `m2` through port `p12`, which in turn issues the request to the component providing this service (in this case `C2`).

Sequence diagrams describe a behavioural view of a system showing the interactions between objects or components in the system. More details on this model as well as a true concurrent semantics based on prime event structures can be found in [9,3]. For this paper, it suffices to understand that in a sequence diagram we describe the messages sent synchronously or asynchronously between objects/components. Each sending/receiving of a message is associated to an event of the sender/receiver. From such a diagram, we can furthermore infer whether such events are ordered, concurrent or in conflict (belong to different operands in an alternative fragment -

not shown here).

Consider Fig. 2 showing an example of (asynchronous) interactions between components on their ports. Diagram M on the left shows the interaction between

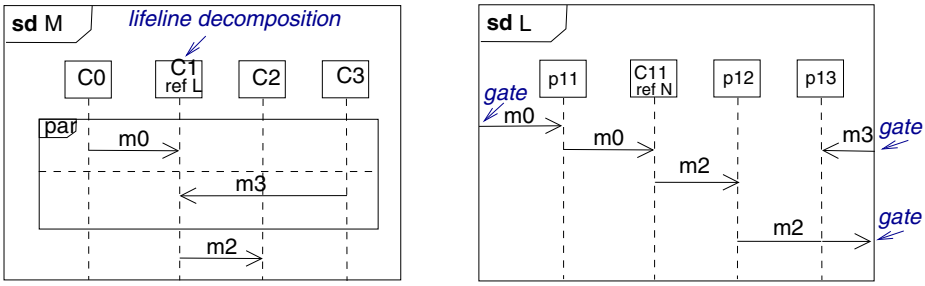

Fig. 2. Component interactions at different levels.

all components at a higher level. The details of component C1, i.e., which port of this component deals with the send/receipt of a message is unspecified, and further refined in the diagram L shown on the right. This form of decomposition is called lifeline decomposition in UML terminology. Diagram L shows the detailed interaction for component C1 only, leaving the sender of m0 and m3 as well as the receiver of m2 unspecified (called gates). The repetition of messages m0 and m2 in diagram L shows that the ports are simply relegating the messages. The component C11 is the internal component of C1 as shown in Fig. 1 on the right, which can be further detailed in diagram N (not shown).

These two kinds of diagrams are linked in the sense that a message used in a sequence diagram implies that the sender (component in our structural diagram) has a required interface containing an operation with the name of the message, and the receiver has a provided interface containing an operation with the same name. In our formalism, we refer to messages and operations interchangeably. Our mathematical model as defined in the next section can be obtained directly from the UML 2.0 models.

3 Concurrent Automata

In describing the externally visible behaviour of a component we need to be able to talk about what happens on its interaction points (ports). 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 [23]. The following definition merely formalises the picture of a component.

We shall assume a (countably infinite) set of port names $\mathcal{P}$ and a (countably infinite) set of names for interface operations Ω. Let Op denote port operations, written as $p.o$ where $p \in \mathcal{P}$ and $o \in \Omega$.

Component Signature. A *component signature* is a tuple $\Sigma = (c, I, L, O, \beta)$ where

- c is a component identifier

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

and we require that I, O, L are pairwise disjoint. Define $P_\Sigma = I \cup O \cup L$ and $Op_\Sigma = \bigcup_{p \in P_\Sigma} \beta(p)$.

This definition captures the static characteristics that serve to identify a component. The elements of L are 'internal' ports concerned with internal actions of a complex component. These are the result of either composition between matching input and output ports of different components, or connecting input/output ports of a complex component with those of its internal components. The latter is required, for example, when the complex component delegates requests to its internal components.

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.

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

By $\beta(p)^*$ we denote the set of finite sequences over $\beta(p)$. 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.

Mathematically, the set V_Σ is the Cartesian product of the sets $\beta(p)^*$, for each p. When an event (or, as we will see, a set of simultaneously occurring events) occurs over a port, it appears on a new vector on the appropriate coordinate. As a result, 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.

Component. 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.

In what follows we describe the basic order-theoretic properties of component

vectors and show how the order structure of the corresponding language expresses ordering constraints on requests over the ports of a component.

We have seen that component vectors are essentially tuples of sequences. We may thus define operations on components vectors in terms of well known operations on sequences. For $\underline{u}, \underline{v} \in V_\Sigma$, we define,

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

Note that $\underline{u} \sqcup \underline{v}$ is defined only when $max(\underline{u}(p), \underline{v}(p))$ exists, for each p. It is easy to see that V_Σ is a monoid with binary operation '$\cdot$' and identity $\underline{\Lambda}_\Sigma$, where $\underline{\Lambda}_\Sigma$ 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 $\underline{\Lambda}_\Sigma$. The operations '$\sqcup$' and '$\sqcap$' give the greatest lower bound and the least upper bound, respectively, of $\underline{u}, \underline{v} \in V_\Sigma$, in the usual sense of lattices and domain theory [5,24]. The right-cancellation operator says that if $\underline{u}$ is an initial part of behaviour $\underline{v}$ so that $\underline{u} \leq \underline{v}$, then $\underline{v}/\underline{u}$ is the 'continuation' of $\underline{u}$ that extends it to $\underline{v}$.

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

Independence. Let $\underline{u}, \underline{v}$ be component vectors in V_Σ. We define $\underline{u}$ and $\underline{v}$ 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. This should become more clear when we consider the transition structure of the corresponding automata.

Component vectors are obtained by coordinatewise concatenation, for example, $(x_1, x_2, x_3) \cdot (y_1, y_2, y_3) = (x_1y_1, x_2y_2, x_3y_3)$. 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.

Column Vectors. 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, $\underline{e} = (\Lambda, m2)$ represents an operation call $m2$ on the port corresponding to the second coordinate. If $m2$ is intended to occur only after both $m0$ and $m3$ have, then this is described in a component vector $\underline{v} = (m0, m3m2)$ which is obtained as $\underline{u} \cdot \underline{e} = (mo, m3) \cdot (\Lambda, m2) = (m0, m3m2) = \underline{v}$.

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 are defined as follows.

Discreteness. Let $V \subseteq V_\Sigma$, then V is *discrete* iff, $\underline{\Lambda}_\Sigma \in V$ and whenever $\underline{u}, \underline{v}, \underline{w} \in V$ such that $\underline{u}, \underline{v} \leq \underline{w}$, then (i) $\underline{u} \sqcup \underline{v} \in V$ and (ii) $\underline{u} \sqcap \underline{v} \in V$.

Note that $\underline{u} \sqcup \underline{v} \in V$ is understood as asserting that $\underline{u} \sqcup \underline{u}$ is defined. Discreteness captures the fact that a system's computations always have a starting point and imposes a finiteness constraint in the sense that it excludes infinite ascending or descending chains of events with respect to time ordering. In order to obtain a precise description of discrete behaviour, we further require that every occurrence of an event (e.g. operation call) is 'recorded' in the component language V. This guarantees that any earlier part of behaviour is itself a behaviour (at the port level) and motivates the local left-closure property.

Local left-closure. Let $V \subseteq V_\Sigma$, $p \in P_\Sigma$ and $x \in \beta(p)^*$. Then, V is *locally left-closed* iff, whenever $\underline{v} \in V$ and $\Lambda < x < \underline{v}(p)$, then there exists $\underline{u} \in V$ such that $\underline{u} \leq \underline{v}$ and $\underline{u}(p) = x$.

We say that a component language V is *normal* iff it is locally left-closed and discrete. This reflects the fact that the guarantees that accrue from discreteness and local left-closure are 'embedded' in the behaviour of the corresponding component. We note that normality is preserved under composition of component languages, as shown in [12].

More importantly, the normality property has as a consequence that component vectors in the language are built up from the empty vector by repeatedly concatenating column vectors to it (cf Proposition 1). This is based on an ordering among component vectors in which one is 'immediately beneath' the other.

Covers. Let $V \subseteq V_\Sigma$ and $\underline{u}, \underline{v} \in V$, then $\underline{v}$ *covers* $\underline{u}$ in V, and we write $\underline{u} \lhd \underline{v}$ iff (i) $\underline{u} \leq \underline{v}$ and $\underline{u} \neq \underline{v}$ and (ii) if $\underline{z} \in V$ such that $\underline{u} \leq \underline{z} \leq \underline{v}$, then $\underline{z} = \underline{u} \vee \underline{z} = \underline{v}$.

The relation '$\lhd$' determines immediate predecessors / successors in a component language and combined with the corresponding column vectors that extend a predecessor to its immediate successor, we may talk about *immediate causality* in the sense of [9].

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. 3 which uses Hasse diagrams to depict the order structure of component languages in which $m0, m3$ are sequential in (i), concurrent in (ii), mutually exclusive in (iii), and simultaneous in (iv). Notice that $m0$ and $m3$ are represented by independent column vectors in the first three

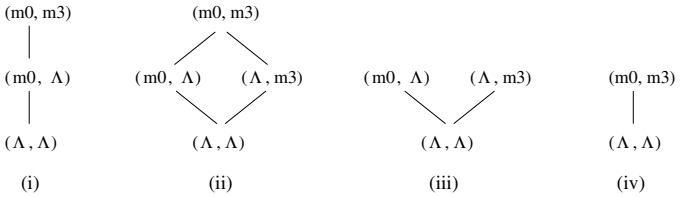

Fig. 3. Order structure of component languages

cases. They are both offered after (Λ, Λ) and occur consecutively in (ii), resulting in concurrent execution. 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 wrt discreteness 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 the issue of restricting to an appropriate subset V of V_Σ is of particular importance. Considerations about obtaining a component language are however beyond the scope of the present paper and have been studied elsewhere. In [14], we have seen how to obtain component languages from scenario-based specifications, more specifically UML2.0 sequence diagrams. In this way, we have also provided a true-concurrent vector semantics to the core of scenario-based notations.

By examining the order-theoretic properties of component vectors, and '$\lhd$' in particular, we may observe that, as the set of ports P_Σ of a component is finite, for any $\underline{u} \in V_\Sigma$, there can only be a finite number of vectors $\underline{v}$ such that $\underline{v} \leq \underline{u}$ (that describe earlier behaviour than $\underline{u}$). From this it follows that if $\underline{u} \neq \underline{\Lambda}_\Sigma$, then there exists a finite sequence $\underline{u}_1, ..., \underline{u}_n$ such that $\underline{\Lambda}_\Sigma \lhd \underline{u}_1 \lhd ... \lhd \underline{u}_n = \underline{u}$. Further, component vectors in a normal language V decompose into products of column vectors. This is established in the following proposition.

Proposition 1. Let $V \subseteq V_\Sigma$ be normal and let $\underline{u}, \underline{v} \in V$. If $\underline{u} \lhd \underline{v}$, then $\underline{v}/\underline{u} \in E_\Sigma$.

To anticipate this allows us to define a transition structure on V which leads to the definition of a class of automata that generate well-behaved components.

We have seen that in a normal component language V, a vector $\underline{z}$ extends a vector $\underline{u}$ to a vector $\underline{v}$ if $\underline{v} = \underline{u} \cdot \underline{z}$ and there is no other vector in V that lies strictly between $\underline{u}$ and $\underline{v}$. The latter requirement can be expressed by saying that $\underline{v}$ *covers* $\underline{u}$, in the sense of the $\lhd$ relation. The continuation $\underline{z}$ which extends $\underline{u}$ to $\underline{v}$ is defined using the right-cancellation operator, i.e. $\underline{v}/\underline{u} = \underline{z}$. In a normal component language such continuations turn out to be elements of E_Σ (by Proposition 1). This observation gives a transition relation which leads to the definition of a type of transition systems.

Σ-machines. Let Σ be a component signature. We define a Σ-*machine* to be a pair $M = (Q, \succ)$ where Q is a set of states and $\succ \subseteq Q \times E_\Sigma \times Q$ is the *transition* relation, and we write $q \succ^{\underline{e}} q'$ for $(q, \underline{e}, q') \in \succ$, which satisfies:

(i) $q \succ^{\underline{e}} q_1 \wedge q \succ^{\underline{e}'} q_2 \wedge \underline{e} \leq \underline{e}' \Rightarrow \underline{e} = \underline{e}' \wedge q_1 = q_2$

(ii) $q \succ^{\underline{e}} q' \wedge q \succ^{\underline{e}'} q' \Rightarrow \underline{e} = \underline{e}'$

We also define a *rooted* Σ-machine to be a pair $M^* = (M, q)$ where $M = (Q, \succ)$ is a Σ-machine and $q \in Q$.

We will write $q \succ \underline{e}$ to denote that there exists $q' \in Q$ such that $q \succ \underline{e}\ q'$. Note that condition (i) includes the case that $\underline{e} = \underline{e}'$ in which case the condition can be rewritten as $q \succ \underline{e}\ q_1 \wedge q \succ \underline{e}\ q_2 \Rightarrow q_1 = q_2$. This condition excludes the case that an event is offered both on its own and simultaneously with other events, and thus guarantees unambiguity in any case.

Rooted Σ-machines determine languages of vectors in the usual way.

Execution Vectors. Let $M = (Q, \succ)$ be a Σ-machine. Define $q \rightarrow \underline{u}\ q'$ if

(i) $q = q'$ and $\underline{u} = \underline{\Lambda}_\Sigma$

(ii) $\underline{u} = \underline{v} \cdot \underline{e}, \underline{e} \in E_\Sigma$, such that $q \rightarrow \underline{v}\ \hat{q} \succ \underline{e}\ q'$, some $\hat{q} \in Q$

We also define $V(M, q) = \{\underline{u} \in V_\Sigma : \exists q' \in Q, q \rightarrow \underline{u}\ q'\}$.

The execution vectors of a Σ-machine can be understood as describing sequences of individual transitions; precisely those out of which $\underline{u}$ is formed. Before introducing Σ-automata we discuss how a Σ-machine can be derived from the language V of a normal component. This is done by taking component vectors in V as states and defining a transition relation in a way that reflects the observation that behaviours may be seen to be built up from the empty vector by repeatedly concatenating column vectors to it.

Suppose that $V \subseteq V_\Sigma$ is normal, then we define $M_c = (V, \succ_V)$ where

$$\underline{u} \succ \frac{e}{V}\ \underline{v} \ \Leftrightarrow\ \underline{u} \lhd \underline{v} \wedge \underline{v}/\underline{u} = \underline{e}$$

We also define $M_c^* = (M_c, \underline{\Lambda}_\Sigma)$.

The subscript V will be dropped when the language is clear from context. Note that $\underline{v}/\underline{u} \in E_\Sigma$, whenever $\underline{u} \lhd \underline{v}$, by Proposition 1 so the $\succ \frac{e}{V}$ relation makes sense.

This construction gives a Σ-machine as shown in the following proposition. Moreover, it can be shown that the vector language generated by M_c from initial state $\underline{\Lambda}_\Sigma$ determines the same language of the same component (Σ, V) using the execution vectors.

Proposition 2. Suppose that $V \subseteq V_\Sigma$ is normal, then (1) $M_c = (V, \succ)$ is a Σ-machine, and (2) $V(M_c^*) = V$.

We are now set to consider true-concurrency in a component language and the corresponding Σ-machine. This builds on ATSs [19] where transitions are thought of as occurrences of events which bear a relation of independence. We have seen that two column vectors are independent when the events they describe engage distinct ports of the component. The minimal requirement for concurrency at state $q \in Q$ is depicted in Fig. 4. Both independent transitions must be enabled at state q, and both must occur between states q and q'. The following definition formulates the property in terms of the transition structure of Σ-machines.

Concurrent Transitions. Suppose that $M = (Q, \succ)$ is a Σ-machine. Define a relation $I^M \subseteq Q \times E_\Sigma \times E_\Sigma$, and we write $\underline{e}_1\ I_q^M\ \underline{e}_2$ for $(q, \underline{e}_1, \underline{e}_2) \in I^M$, by

$$\underline{e}_1\ I_q^M\ \underline{e}_2 \Leftrightarrow \underline{e}_1\ ind\ \underline{e}_2 \wedge (\exists q_1, q_2, q' \in Q : q \succ \underline{e}_1\ q_1 \wedge q \succ \underline{e}_2\ q_2 \succ \underline{e}_1\ q')$$

We shall drop the superscript M when it is clear from context. It can be seen that

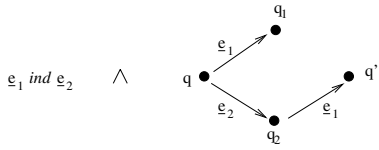

Fig. 4. Concurrent transitions $\underline{e}_1, \underline{e}_2$

I_q defines *local* concurrency in the sense that column vectors $\underline{e}_1, \underline{e}_2$ are concurrent *at state* q of the machine.

Apart from concurrency, we also required discreteness and local left-closure in our characterisation of normal component languages. Consequently, we are interested in a class of automata that determine such languages only. This requires additional constraints on the transition structure, as described in [13] and in more detail in [14]. The following definition refines Σ-machines to Σ-automata, by adding conditions that allow to express concurrency explicitly and ensure the generated language is discrete and locally left-closed.

Σ-automata. Let Σ be a signature. A Σ-*automaton* M is a Σ-machine $M = (Q, \succ)$ satisfying

(i) If $\underline{e}_1\ I_q^M\ \underline{e}_2$ and $q \succ^{\underline{e}_1} q_1 \succ^{\underline{e}_2} \hat{q}$, then $q \succ^{\underline{e}_2} q_2 \succ^{\underline{e}_1} \hat{q}$, some $q_2 \in Q$

(ii) If $q_1 \succ^{\underline{e}_1} \hat{q}$ and $q_2 \succ^{\underline{e}_2} \hat{q}$ and $q_1 \neq q_2$, then $\underline{e}_1\ ind\ \underline{e}_2$ and there exists $q \in Q$ such that $q \succ^{\underline{e}_2} q_1$ and $q \succ^{\underline{e}_1} q_2$

(iii) If $\underline{u}, \underline{v} \in V_\Sigma$ and $q \rightarrow^{\underline{u}\cdot\underline{v}} q''$, then $\exists q' \in Q$ such that $q \rightarrow^{\underline{u}} q' \Leftrightarrow q' \rightarrow^{\underline{v}} q''$

(iv) If $\underline{e}_1, \underline{e}_2 \in E_\Sigma$ s.t. $q \succ^{\underline{e}_1, \underline{e}_2}$ and $\underline{x} \in V(M, q)$ with $\underline{e}_1, \underline{e}_2 \leq \underline{x}$, then $\underline{e}_1 I_q^M \underline{e}_2$

We also define a *rooted* Σ-automaton to be a rooted Σ-machine $M^* = (M, q)$ where M is a Σ-automaton.

Note that by Definition of I_q and condition (1) above we have that I_q is symmetric and irreflexive. Symmetry reflects the fact that concurrency is always mutual while irreflexivity prohibits considering an event as being concurrent with itself.

Condition (1) is characteristic of automata for non-interleaving representation of behaviour and is sometimes called the *lozenge rule* [19,17]. It is depicted in Fig. 5. Condition (2) relates to discreteness of the generated language. This property

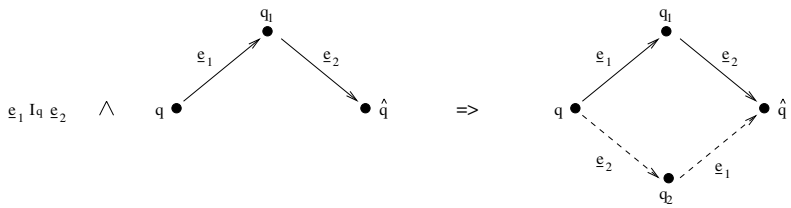

Fig. 5. Condition (1)

requires that elements bounded above in the language have their least upper bound and greatest lower bound in it. Subsequent analysis in [22] shows this to be the case when the generated language satisfies the *lower diamond property*, which says that whenever we have the upper half of a diamond, then we have the whole diamond. Condition (3) excludes the possibility that an execution vector may be produced in

two different ways from sequences of individual transitions. Condition (4) says that if two distinct transitions can start off the same behaviour from q, i.e. be part of the same execution vector from q, then they must do so concurrently.

The composition of Σ-automata has been described in [14,13]. In a fashion similar to that of composition of components [12], the key idea is that a component vector $\underline{v}$ represents behaviour of the product $M_1||M_2$ provided it results from component vectors $\underline{v}_1$ of M_1 and $\underline{v}_2$ of M_2 which agree on *complementary* interfaces. Complementary interfaces are the interfaces where composition takes place (in other words where components can be connected), i.e., interfaces provided by one component and required by the other component. For example, Fig 1 showed three cases of complementary interfaces connecting components `C0` and `C1`, `C1` and `C2`, and `C1` and `C3`.

4 Concurrent Logic

In this section, we describe our concurrent logic, a distributed temporal logic addressing concurrency explicitly, and how it can be used to specify component properties at different levels of abstraction for component-based systems. This logic is derived from MDTL (see e.g. [9]) with some differences to allow us to specify component properties more adequately. The logic here is interpreted over concurrent automata as opposed to MDTL which is defined over labelled prime event structures. Notice that our only gain over MDTL concerns practical reasons, that is, with an interpretation over concurrent automata we are closer to the framework underlying the Edinburgh Concurrency Workbench.

A component c specified by a signature Σ and a component language V as defined in the previous section, has a *home* logic to describe internal properties or describe interactions from a local point of view. Further, a *communication* logic describes interactions between several components. To some extent, the communication logic describes an *observer* of component interactions in a (possibly partial) view of the system. We assume *shallow* views of (a part of) a system only, i.e., at a particular level of detail a view of the (sub)system consists of the top most components on that level. For our example system of Fig. 1 (left), an observer is only able to see the four interacting components but not internal details of individual components. Moving to a lower level (Fig. 1 right), an observer only sees the top most internal details of the component `C1`, that is, interactions between the component ports and internal component `C11`.

Let $\mathcal{C}$ be the set of component names in our system, let v be a view of the system such that $\mathcal{C}_v$ is the subset of components from the system known for view v. The abstract syntax of the concurrent logic CL defined over view v, where $c \in \mathcal{C}_v$ and $d \in \mathcal{C}_v \cup \{g\}$ are components, g stands for an external component or gate, and obs_v stands for an observer of view v, is given as follows:

$$\mathrm{CL}_v ::= \{c.H_c\}_{c \in \mathcal{C}_v} \mid C_v$$
$$H_c ::= \mathrm{ATOM}_c \mid \neg H_c \mid H_c \Rightarrow H_c \mid H_c \; U \; H_c \mid H_c \; \Delta \; H_c$$

$C_v ::= c.Mes!d \leftrightarrow d.Mes?c \mid c.Mes!d \rightarrow d.Mes?c \mid H_{obs_v}$

$\text{ATOM}_c ::= true \mid Att\ \theta\ t \mid \triangleright Mes!d \mid \triangleright Mes?d \mid Mes!d \mid Mes?d$

The home logic H_c is basically an extension of temporal logic with a binary concurrency operator Δ. From $\neg$ and $\Rightarrow$ we can derive the other usual connectives (e.g., $\varphi \wedge \psi \equiv \neg(\varphi \Rightarrow \neg\psi)$). Similarly, further temporal operators can be derived from the *weak* until. The intuition of a formula $c.(\varphi_1 \Delta \varphi_2)$ is that from the point of view of component c, φ_1 and φ_2 hold concurrently. The set of message labels *Mes* is used to capture message terms where $Mes!d$ denotes sending a message to d and $Mes?d$ denotes receiving a message from d. Further, $\triangleright Mes$ uses the enabled predicate ($\triangleright$) applied to a message term to indicate that the sending or receiving of a message is enabled (*may* happen next). The set of attribute symbols *Att* is used to denote attribute terms where θ is a comparison predicate (e.g., $<, \leq, =, \geq, >, \ldots$) and t is a data term of the same type. The atomic formulae obtained with attribute terms can be used to describe constraints in alternative fragments (not treated in this paper).

In the communication logic C_v, $\leftrightarrow$ is used for synchronous communication and $\rightarrow$ for asynchronous communication. We always assume that if a message is sent it must be received. Notice that the communication logic can refer to H_{obs_v} where obs_v can be understood as a placeholder for any component in $\mathcal{C}_v$ which the observer can see, and the observer can thus see *beyond* communication, for example, observe concurrent executions, and so on. CL is indexed over views, and in this way we can obtain a hierarchical view of the system, i.e., describe properties of the component-based system at different levels of detail. We will illustrate this with our example.

We can use the logic CL to describe general properties of components as well as the interactions between components (at different levels of abstraction) in the system. Recall the example introduced earlier in Fig. 1 and Fig. 2. Let us consider Fig. 2 on the left as our view v_m, and on the right as our view v_l. In v_m, $\text{CL}_{v_m} ::= c.H_{c \in \{C_0, C_1, C_2, C_3\}} \mid C_{v_m}$. An observer of v_m, sees the interaction between `C0`, `C1`, `C2` and `C3`, and can observe, for example, the following properties written as formulae in the communication logic C_{v_m}:

$$C_0.m_0!C_1 \rightarrow C_1.m_0?C_0$$

denoting the asynchronous communication between `C0` and `C1` on message `m0`,

$$C_0.m_0!C_1\ \Delta\ C_3.m_3!C_1$$

denoting the concurrent send of messages `m0` and `m3`, and

$$C_1.m_0?C_0\ \Delta\ C_1.m_3?C_3$$

denoting the concurrent receipt of the same messages `m0` and `m3`. These last two formulae are examples of formulae in $H_{obs_{v_m}}$, i.e., observations that can be made by an observer of v_m which go beyond communication.

In the home logic of component `C1`, H_{C_1} of CL_{v_m}, we can also express the formula:

$$C_1.(m_0?C_0\ \Delta\ m_3?C_3)$$

which similarly to the last formula in C_{v_m} expresses the concurrent receipt of the

messages `m0` and `m3`.

This naturally means that for every local formulae in H_c, for example of the form $c.(\varphi_1 \Delta\ \varphi_2)$, there is a global formulae in H_{obs} of the form $c.\varphi_1 \Delta\ c.\varphi_2$. There are also equivalent ways of expressing properties in C. For example, a communication formulae can always be expressed in H_{obs} in an equivalent way. We do not formalise such equivalences here for space reasons.

We now move to view v_l, a lower level of interactions within `C1`, that is, we look at the interactions between components (and their ports) known internally to `C1`. Here $\text{CL}_{v_l} ::= c.H_{c \in \{C_{11}, C_1::p_{11}, C_1::p_{12}, C_1::p_{13}\}} \mid C_{v_l}$. As seen on the right diagram of Fig. 2, we also must consider the ports of component `C1` since at this level we show interactions between these ports and component `C11`. By convention we use $c :: p$ to indicate a port p of component c. To simplify the formulae, we often omit the name of the component. Examples of formulae in C_{v_l} are:

$$g.m_0!p_{11} \rightarrow p_{11}.m_0?g$$

denoting the asynchronous communication between a gate and port `p11` on message `m0`, and

$$C_1 :: p_{11}.m_0?g\ \Delta\ C_1 :: p_{13}.m_3?g$$

denoting the independent receipt of messages `m0` and `m3` by the different ports `p11` and `p13`.

This formula interestingly relates to the last formulae we showed for C_{v_m}. In the former case, we knew that component `C1` received the messages concurrently, whereas in the refined view it becomes clear that these messages are actually received independently by the component at different ports. On the other side, in this refined view we lose the knowledge of the actual sender of both messages which is seen as a gate. As mentioned above, in a reasoning framework, we would need to formalise the translations between different views which we are not defining at present in this paper. We now describe the semantics of our logic in terms of concurrent automata.

The logic CL is interpreted over concurrent automata. In particular, a model for CL_v is given by $\mathcal{M}$ where $\mathcal{M}$ is a Σ_v-automaton, i.e., a concurrent automata for the composite of the components in $\mathcal{C}_v$. The satisfaction of a formula φ at a state $q \in Q$ is denoted by $\mathcal{M}, q \models \varphi$. In particular, $\mathcal{M}, q \models c.\varphi$ iff $\mathcal{M}_c, q \models_c \varphi$ for $c \in \mathcal{C}_v$, $q \in Q_c$ and $\mathcal{M}_c$ is a Σ_c-automaton. Here, we only give the satisfaction rules for the temporal operators and the concurrency operator as the others are standard.

(i) $\mathcal{M}_c, q \models_c \varphi\ U\ \psi$ holds iff there is some state q' with $q \rightarrow^{\underline{v}} q'$ for a vector $\underline{v} \neq \underline{\Delta}_\Sigma$ such that $M_c, q' \models_c \psi$ holds, and $M_c, q'', \models_c \varphi$ holds for each q'' where $q \rightarrow^{\underline{u}} q'' \rightarrow^{\underline{u}'} q'$ and $\underline{u} \cdot \underline{u}' = \underline{v}$.

(ii) $\mathcal{M}_c, q \models_c \varphi_1 \Delta \varphi_2$ holds iff there are states q' and q'', and column vectors $\underline{e}$ and $\underline{e}'$, such that $q \succ^{\underline{e}} q'$, $q \succ^{\underline{e}'} q''$, $\underline{e}\ ind\ \underline{e}'$ and both $\mathcal{M}_c, q' \models_c \varphi_1$ and $\mathcal{M}_c, q'' \models_c \varphi_2$ hold.

(iii) $\mathcal{M}_c, q \models_c \triangleright m!d$ holds iff there is a state q' such that $q \succ^{\underline{e}} q'$ and $\underline{e}(p) = m$ for some $p \in O \subset P_\Sigma$.

(iv) $\mathcal{M}_c, q \models_c m?d$ holds iff there is a state q' such that $q' \succ^{\underline{e}} q$ and $\underline{e}(p) = m$ for

some $p \in I \subset P_\Sigma$.

The first rule gives a weak definition of the until operator, i.e., if there is a state q' where ψ holds then there is a finite sequence of individual transitions leading to q' where φ must hold. From condition (3) of Σ-automata this sequence must be unique. The second rule talks about concurrency, and defines the semantics of the binary concurrency operator Δ. $\varphi_1 \Delta \varphi_2$ holds at state q if there are concurrent transitions from state q leading to states q' and q'' where φ_1 holds at q' and φ_2 holds at q'' respectively. The last two rules allow us to distinguish between a message being enabled (rule 3 shown only for a message being sent) and a message occurring (rule 4 shown only for a message being received).

5 Conclusions and Related Work

In this paper, we have described the first steps of a mathematical framework for modelling and subsequently reasoning about component-based systems given initially as UML 2.0 models. The framework consists of a true-concurrent logic for describing properties of the system at different levels of abstraction, and is interpreted over concurrent automata, a special kind of asynchronous transition system (ATS) [19]. Concurrent automata are generated by a language-based representation of behaviour, in our case of component behaviour.

The language-based representation of component behaviour described in this paper builds on the language-theoretic constructions in [21] which have been further studied in [14]. This work has been modified here to model ports of a component and extended to include the notion of *local* or internal ports. This further allows us to consider requests handled by the internal parts of a complex component.

Our model consists of a pair: a *component signature* which captures the static view of a component as depicted in UML 2.0 [18] or the Koala component model [23], and a 'language' of *component vectors* over this signature which describe the behaviour of the component. There is a strong similarity between the notion of a component signature and the static structure of interface automata [6]. The significant difference is that whereas in our model ports are associated to a set of operation calls/signals, thereby corresponding to channels in a process algebra such as CSP [7], ports in [6] correspond to individual operations/signals which are furthermore assumed to be sent or received sequentially. Notice that we deal with sets of operations instead and also allow concurrency between ports of the same component. Consequently, interface automata [6], as well as similar approaches such as constraint automata [2], may be considered as special cases of concurrent automata.

In our model, behaviour is represented by a set of tuples of sequences of events, the so-called component vectors, where each coordinate corresponds to a port and contains a sequence of events that may occur over that port (e.g. operation calls/signals arriving or departing or transmitted internally). This view of behaviour is very similar to that of Broy [4] whose components implement partial functions mapping input streams to output streams. Our approach, which seeks as much

generality as is consistent with plausibility, tends to give rise not to functions but to *relations* between input and output streams. This lays the foundation for examining the following problem: given a specification of the input output behaviour of a component, possibly in terms of UML 2.0 models, under what circumstances can we guarantee the existence of a normal component which exhibits such behaviour.

Our model can be associated with behavioural presentations [20] which are a mild generalisation of prime event structures [16] in that the time ordering of events is taken to be a pre-order (a reflexive and transitive relation) rather than a partial order, thereby allowing the representation of simultaneity as well as concurrency. The equivalence relation generated by the pre-order can be used to describe events that occur at *exactly the same time* (recall Fig. 3(iv)) and thus it is possible to express concurrency between simultaneity classes of event occurrences.

The connection between our model and behavioural presentations follows that described in [14]. We may impose an ordering on behaviours and associate occurrences with primes [5], based on the fairly standard techniques found in [16,21]. In this case, we have seen that vectors are related by coordinatewise prefix ordering, and a prime is a vector which *covers* precisely one other vector. A vector is then associated with the set of primes beneath it, and this gives a set which inherits the ordering between vectors and allows the formal treatment of phenomena such as concurrency and nondeterminism.

It is important to note that the assumption of sequential behaviour at each port is not as restrictive as it might appear. There is little advantage in allowing parallel access to individual ports. Concurrent access to the component is possible by having more than one input port - a similar observation applies to output. Parallel access to a port would make sense only if the port has multiplicity (which is possible in UML 2.0) and this is something to be explored in future work. Hence, components in our model are *not* treated as sequential, on the contrary, they exhibit true-concurrency. The explicit treatment of this temporal phenomenon distinguishes our approach, for example, from the interleaving approach of [6,7,11]. Broy's stream-based model [4] does exhibit concurrency, but this is lost once the equations representing the partial functions on streams are translated into automata.

Our formal approach for modelling components and subsequently reasoning about properties of component-based systems supports true-concurrency. This may yield more abstract specifications and allows us to consider concurrency at the level of individual components (between ports of the same component) in addition to concurrency that may arise through composition of components (via matching ports from each, as described in [12]).

Another strength of our approach concerns the true-concurrent hierarchical logic which allows us to express concurrency explicitly but also reflect properties of components at different levels of abstraction. As mentioned earlier in the paper, we have to clarify how and when formulae can be translated from one view to another. To the best of our knowledge, from the existing logics that address concurrency, none have the means to explore different levels of abstraction in quite the same way. This is a crucial feature to address complex component-based systems. Finally, the logic

in [1] defined for reasoning about mixed specifications with explicit gluing mechanisms compares to ours in that both allow us to express component-based systems at a high level, how components can be glued together (through interaction) and offer a way of dealing with refinement. The logic given in [1] is, however, based on first order logic, assumes component synchronisation only and does not address true-concurrency.

We are currently working on the integration of our concurrent logic and concurrent automata into the Edinburgh Concurrency Workbench (CWB). Since concurrent automata are an extension of ATS, we expect the CWB extension to be fairly straightforward given previous work on defining an ATS semantics for CCS (see e.g., [15]). Ultimately, our aim is to establish a connection between UML 2.0 and CWB enabling the verification of component-based systems.

References

[1] M. Aiguier, F. Barbier, and P. Poizat. A logic with temporal glue for mixed specifications. In *Foundations of Coordination Languages and Software Architectures 2003*, volume 97 of *ENTCS*, pages 155–174. Elsevier, 2004.

[2] C. Baier, M. Sirjani, F. Arbab, and J. Ruttens. Modeling Component Connectors in Reo by Constraint Automata. *Science of Computer Programming*, 61(2):75–113, 2006.

[3] J. K. F. Bowles. Decomposing interactions. In *Algebraic Methodology and Software Technology: 11th International Conference, Kuressaare, Estonia, 5-8 July 2006*, volume 4019 of *LNCS*, pages 189–203. Springer, 2006.

[4] M. Broy. Algebraic Specification of Reactive Systems. *Theoretical Computer Science*, 239(2000):3–40, 2000.

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

[6] L. de Alfaro and T. Henzinger. Interface Automata. In *Proceedings of 8th European Software Engineering Conference*, pages 109–120. ACM Press, 2001.

[7] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[8] J. Küster-Filipe. Fundamentals of a module logic for distributed object systems. *Journal of Functional and Logic Programming*, 2000(3), March 2000.

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

[10] A. Mazurkiewicz. Basic Notions of Trace Theory. In *Linear Time, Branching Time and Partial Orders in Logics and Models for Concurrency*, volume 354 of *LNCS*, pages 285–363. Springer Verlag, 1988.

[11] A. J. R. Milner. Calculus for Communicating Systems. volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, 1980.

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

[13] S. Moschoyiannis, M. W. Shields, and P. J. Krause. Modelling Component Behaviour Using Concurrent Automata. In *Formal Foundations of Embedded Software and Component-Based Software Architectures (FESCA'05)*, volume 141 of *ENTCS*, pages 199–220. Elsevier, 2005.

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

[15] M. Mukund and M. Nielsen. CCS, Locations and Asynchronous Transition Systems. In *Proceedings of the 12th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 652 of *LNCS*, pages 328–341. Springer, 1992.

[16] M. Nielsen, G. Plotkin, and G. Winskel. Petri Nets, Event Structures and Domains, part 1. *Theoretical Computer Science*, 13:85–108, 1981.

[17] M. Nielsen, V. Sassone, and G. Winskel. Relationships between Models of Concurrency. In *Proceedings of REX School/Symposium: A Decade of Concurrency, Reflections and Perspectives*, volume 803 of *LNCS*, pages 425–476. Springer-Verlag, 1994.

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

[19] M. W. Shields. Concurrent Machines. *Computer Journal*, 28:449–465, 1985.

[20] M. W. Shields. Behavioural Presentations. In *Linear Time, Branching Time and Partial Orders in Logics and Models for Concurrency*, volume 354 of *LNCS*, pages 671–689. Springer Verlag, 1988.

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

[22] M. W. Shields and S. Moschoyiannis. An Automata-Theoretic View of Software Components. Technical Report SCOMP-TC-02-04, Department of Computing, University of Surrey, 2004.

[23] 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.

[24] 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.