Component-Based Design: Towards Guided Composition

S. Moschoyiannis and M. W. Shields
Department of Computing
University of Surrey
Guildford, Surrey
GU2 7XH
United Kingdom
{s.moschoyiannis, m.shields}@eim.surrey.ac.uk

Abstract

In this paper, we present a mathematical model for the composition of software components, at a semantic modelling level. We describe a mathematical concept of a single software component and identify properties that ensure its potential behaviour can be captured. Based on that, we give a formal definition of composition and examine its effect on the individual components. We argue that properties of the individual components can, under certain conditions, be preserved in the composite. The proposed model can be used for guiding the composition of components as it advocates formal reasoning about the composite before the actual composition takes place.

1. Introduction

The development of large-scale, evolvable software systems in a timely and affordable manner can, potentially, be realised by assembling systems from pre-fabricated software components. The component-based approach to software engineering is emerging as the key development method, as it advocates the (re)use of existing (independent) software components in producing the final system.

Inevitably, in the context of component-based software engineering (CBSE) emphasis is placed on *composition*. It can be argued that software systems built by assembling together independently developed and delivered components sometimes exhibit pathological behaviour. Part of the problem seems to be that developers of such systems do not have a precise way of expressing the behaviour of components at their interfaces, where the inconsistencies occur. Graphical notations such as Koala [30] and the widely used UML [21, 2] attempt to capture behavioural aspects of a system, but lack an associated formalism to aid designers in precisely describing dynamic properties of components. Components may be developed at different times and by different developers with, possibly, different uses in mind. Their different internal assumptions, further exposed by concurrent execution, may give rise to emergent behaviour when these components are used in concert, e.g. race conditions, deadlocking behaviour, etc.

Current efforts to address the technical problems are directed at providing support for predicting properties of the assemblies of components before the actual composition takes place. Yet, this requires prior knowledge of the individual components' properties. We argue in favour of an *a priori reasoning* [14] approach to CBSE, in which reasoning about composition is based on properties of the individual components. First, it must be shown that the components adhere to their own specifications. Based on correctness of individual components, their composition can then be guided to meet the specification of a larger system as well as predict the behaviour of the composite. In order to prove that a software component meets its own specification, and even more importantly, will continue to do so when fitted together with other components, we need a well-grounded mathematical framework. The ability to formally describe the concurrent behaviour of interacting components is a key aspect in component-based design.

In this paper, we describe a formal model for software components, at a semantic modelling level, which can be used to describe and reason about generic issues related to components and their composition. In particular, we formally specify a single software component, identifying conditions that ensure it is 'well-behaved'. We also give a formal definition of the composition of components. We argue that when we put two well-behaved components together, the resulting system is also well-behaved. (i.e. the conditions hold for the composite).

The proposed mathematical model is based on a fairly simple idea. The static structure of a component is described by a sort (see Definition 2.1) while its dynamic characteristics are captured by tuples of sequences which model calls to operations on interfaces of the component. Putting together such sequences, one for each interface, we form sets of vectors of sequences. Each coordinate corresponds to an interface of the component. It represents the behaviour of the component at that interface, during the period of operation in question. We restrict the component model by imposing certain conditions; that is, discreteness and local left closure. Each component defined in this way, can be associated with an event structure -like object, called a behavioural presentation [24]. Thus, the component model can be related to a general theory of non-interleaving representation of behaviour [25].

As for composition, it takes place via complementary interfaces with the restriction that each interface corresponds to a unique (input or output) port of the component. The static structure of the composite is formed by those of the components. The dynamic characteristics comprise behaviours of each component and these must agree on connected interfaces.

The use of tuples of sequences to model concurrent behaviour was first introduced by one of the authors [22]. However, the vector language used to describe the input / output behaviour of a software component in the proposed model, differs in important respects from that in [22] and rather, is reminiscent of the use of streams in [6] to represent messages communicated along the channels of a component. In fact, the setout of our model is quite similar to the algebraic specification model of Broy. It is worth mentioning though that in [6] both finite and infinite sequences of messages are considered whereas we only work with finite sequences of calls to operations.

Common ground between the two models can be found in the mathematical concept of a software component and particularly, in describing the static characteristics of a component. The difference lies with the use of the notion of sort. In [6] it is considered to be the set of messages associated with each channel of the component while in our model the notion of sort (see Definition 2.1) is used in a more abstract sense and refers to the static picture of a component as a whole. Semantically, a component in [6] is represented by a predicate defining a set of behaviours where each behaviour is represented by a stream processing function. In this respect, the two models diverge since our model is mostly based on the order theoretic structure of the set of behaviours of a component and is then related to behavioural presentations, which provide an operational semantics expressive enough to model non-determinacy, concurrency and simultaneity as distinct phenomena.

This paper is structured as follows. The next section describes a formalism for specifying a single software component, including component properties that allow us to characterise a component as *normal* (see Definition 2.6). In Section 3, we outline a mathematical framework for the composition of components. We return to the idea of normality in Section 4 where the effect of composition, in terms of preservation of property normality, is examined. Finally, Section 5 includes some concluding remarks and a discussion on future work.

2. Formal Specification of a Component

A software component can be understood as an encapsulated software entity with an explicit interface to its environment which can be used in a variety of configurations. At a specification level, a component provides services to other components and possibly requires services from other components in order to deliver those promised. The offered services are made available via a set of *provides* interfaces while the reciprocal obligations are to be satisfied via a set of *requires* interfaces.

In line with the object-oriented paradigm, we take a *black box* view of a component [9, 28]. Its functionality is made available to the rest of the system only through its interfaces. Pictorially [30], a component is a square box with a number of input and output ports. Each port is associated with an interface and communication with other components is established via the operations of each interface. We shall assume a countable infinite set I of interface names and a countable infinite set Op of operations of those interfaces, both sets remaining fixed for the remainder of this paper. The following definition merely formalises the picture of a typical component.

Definition 2.1 *We define a (component) sort to be a tuple* $\Sigma = (P_\Sigma, R_\Sigma, \beta_\Sigma)$ *where*

- $P_\Sigma \subseteq I$ *is a set of provides interfaces*
- $R_\Sigma \subseteq I$ *is a set of requires interfaces*
- $\beta_\Sigma : P_\Sigma \cup R_\Sigma \to \wp(Op); \beta_\Sigma(i)$ *is the set of calls to operations associated with interface* i

and we require that $P_\Sigma \cap R_\Sigma = \emptyset$. *Define* $I_\Sigma = P_\Sigma \cup R_\Sigma$.

These sets and this function comprise the static structure of a typical component. As for its dynamic characteristics we introduce the notion of behaviour vectors in our model.

Definition 2.2 *Suppose that* Σ *is a sort. We define* V_Σ *to be the set of all functions* $\underline{v} : I_\Sigma \to Op^*$ *such that for each* $i \in I_\Sigma, \underline{v}(i) \in \beta_\Sigma(i)^*$. *We shall refer to the behaviour vectors of* V_Σ *as* β_Σ*-vectors.*

By $\beta_\Sigma(i)^*$ we denote the finite sequences over $\beta_\Sigma(i)$. Thus, the function $\underline{v}$ returns the finite sequences of calls to operations made at and by interface i, for each interface i of the component.

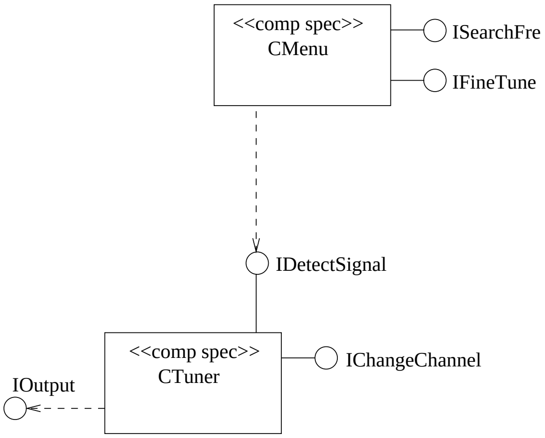

Figure 1. Component specification architecture

Based on the above definitions, we obtain a mathematical concept of a component. We shall define a component c to consist of the static structure described by a sort Σ together with a language of β_Σ*-vectors*.

Definition 2.3 *A component c is a pair* (Σ, B)*, where*

- Σ *is the sort of c*
- $B \subseteq V_\Sigma$ *and* $B \neq \emptyset$ *is the set of behaviours of c.*

The main concept behind employing β_Σ*-vectors* is that the behaviour of the component as a whole may be described by assigning to each interface i a sequence of calls to operations of an interface. Being focused on fundamental principles, we base our model on abstract component concepts where calls to operations of an interface correspond to events, that is, arrivals or departures of signals at ports of the component, and component behaviour is represented by tuples of sequences of signals entering or leaving the component through its ports.

Example 2.1 Consider a small and simplified extract of a TV platform, related to the MENU fuctionality of TV set. The MANUAL STORE options are provided by the interaction of the components of Figure 1 which depicts the component specification architecture using the notation of [21, 7]. The stereotype <<comp spec>> is introduced to describe component specifications and the UML lollipop notation is used for interfaces. The component architecture of Figure 1 comprises a set of application-level components together with their structural relationships and behavioural dependencies [12].

The CMenu component requires services through interface IDetectSignal in order to implement the promised services via interfaces ISearchFre and IFineTune that it provides. The ISearchFre interface has operations *highlightItem* and *startSearch*. Calls to these operations shall be denoted by a_1, a_2 respectively, for abbreviation. The IFineTune interface has operations *highlightItem*, *incrementFre* and *decrementFre*, abbreviated by b_1, b_2 and b_3 respectively. The CMenu component establishes communication with users via its provided interfaces ISearchFre and IFineTune. A user requests to search the available frequency for a program via the ISearchFre interface. The CMenu component cannot satisfy the requested operation itself and requires a component providing the IDetectSignal interface to conduct the frequency search on its behalf. This is done by invocation of an operation *detectingSignal* (abbreviated by c_1) on its required interface IDetectSignal, which is implemented by the CTuner component.

In what follows, we apply the mathematical theory presented earlier to model the CMenu component. Its provided interfaces ISearchFre and IFineTune will be denoted by $p1$ and $p2$ and the required interface IDetectSignal by $r1$. Thus, $P_\Sigma = \{p1, p2\}$, $R_\Sigma = \{r1\}$. Hence, $I_\Sigma = P_\Sigma \cup R_\Sigma = \{p1, p2, r1\}$ and of course, $P_\Sigma \cap R_\Sigma = \emptyset$. Function β_Σ as defined in Definition 2.1 provides the set of calls to operations associated with each interface. Hence, $\beta_\Sigma(p1) = \{a_1, a_2\}$, $\beta_\Sigma(p2) = \{b_1, b_2, b_3\}$ and $\beta_\Sigma(r1) = \{c_1\}$.

It can be seen that $\Sigma = (P_\Sigma, R_\Sigma, \beta_\Sigma)$ is a sort. And if we write (x, y, z) for the function $\underline{v}$ of Definition 2.2 with $\underline{v}(p1) = x$, $\underline{v}(p2) = y$ and $\underline{v}(r1) = z$ we can define the set of behaviours for the CMenu component as,

$$B = \{(\Lambda,\Lambda,\Lambda), (a_1,\Lambda,\Lambda), (\Lambda,b_1,\Lambda), (a_1a_2,\Lambda,\Lambda), (\Lambda,b_1b_2,\Lambda), (\Lambda,b_1b_3,\Lambda), (a_1,b_1b_2,\Lambda), (a_1a_2,\Lambda,c_1), (a_1a_2,b_1,\Lambda), (a_1,b_1,\Lambda), (a_1,b_1b_2b_3,\Lambda), (a_1a_2,b_1,c_1)\}$$

It turns out that $c = (\Sigma, B)$ is a component (recall Definition 2.3) where $\Sigma = (P_\Sigma, R_\Sigma, \beta_\Sigma)$ is a component sort and B is a subset of all behaviour vectors V_Σ.

The mathematics of β_Σ*-vectors* is given in [26] and is very similar to that of [25]. The main difference is that while vectors in [25] describe behaviour of systems of sequential processes combined using something like the parallel composition operator || of CSP [10], β_Σ*-vectors* describe behaviour of systems using something like the interleaving operator ||| of CSP.

In this paper, we present the fairly basic properties of β_Σ*-vectors*. If v and u are sequences we write $v.u$ for the concatenation of v and u. As is well known, this operation is associative with identity Λ, where Λ denotes the empty sequence. We also have a partial order on sequences given by $v \leq w$ if and only if there exists u such that $v.u = w$, and this partial order has a bottom element Λ. It is also well known that concatenation is cancellative, thus u is unique.

Further, the set of behaviour vectors V_Σ is a monoid with binary operation '.' and identity Λ. It is also a partially ordered set with partial order '$\leq$' and bottom element Λ. The interested reader is referred to [26] where the order theoretic properties of V_Σ are established.

We shall now introduce two basic operations on the set of behaviours of a component, based on the order theoretic properties of the set V_Σ.

Definition 2.4 *Let $\underline{u}$ and $\underline{v}$ be behaviour vectors in $B \subseteq V_\Sigma$. Then,*

1. *$\underline{u} \sqcap \underline{v}$ is defined to be the vector $\underline{w}$ which satisfies $\underline{w}(i) = min(\underline{u}(i), \underline{v}(i))$, each i*
2. *$\underline{u} \sqcup \underline{v}$ is defined to be the vector $\underline{w}$ which satisfies $\underline{w}(i) = max(\underline{u}(i), \underline{v}(i))$, each i*

In terms of partial orders the above operations essentially give the greatest lower bound and the least upper bound of $\underline{u}, \underline{v} \in B$, in the usual sense of lattices and domain theory [8, 31]. Recall that if $(X, \leq)$ is a partially ordered set [8] then the least upper bound of $x_1, x_2 \in X$, if it exists, is the least element $x \in X$ such that $x_1, x_2 \leq x$. We denote it by $x_1 \sqcup x_2$. The greatest lower bound, denoted by $x_1 \sqcap x_2$, is the largest element $x \in X$ such that $x \leq x_1, x_2$. Notice that these are computed coordinate-wise for the behaviour vectors of our model.

A key property of the sets V_Σ is that they possibly contain discrete subsets. Before introducing discreteness, we also need to define consistent completeness. We shall say that B is *consistently complete* if and only if i) $\underline{\Lambda}_{\beta_\Sigma} \in B$ and ii) whenever $\underline{v}_1, \underline{v}_2, \underline{w} \in B$ such that $\underline{v}_1, \underline{v}_2 \leq \underline{w}$, then $\underline{v}_1 \sqcup \underline{v}_2 \in B$. In short, the notion of consistent completeness for a poset dictates that whenever two of its elements are less or equal than a third in the set, their least upper bound not only exists but is also in the poset.

Now, we can impose the first condition on a software component.

Definition 2.5 *Let I_Σ and Op be sets with I_Σ finite, and $\beta_\Sigma : I_\Sigma \to \wp(Op)$, and suppose that $B \subseteq V_\Sigma$, then we shall say that B is discrete iff*

1. *B is consistently complete*
2. *If $\underline{u}_1, \underline{u}_2 \in B$, then $\underline{u}_1 \sqcup \underline{u}_2 \in B \Rightarrow \underline{u}_1 \sqcap \underline{u}_2 \in B$*

Let $c = (\Sigma, B)$ be a component; if B is discrete, then c is discrete.

Informally, the above definition refers to vectors in the set of behaviours B of the component which have at least two distinct immediate predecessors and says that both the least upper bound and the greatest lower bound of these predecessors must exist and also belong to the set of behaviours B. In short, such vectors together with their predecessors must constitute finite lattices.

The justification for this constraint is as follows. A set of behaviours of a software component may be translated into an object called a behavioural presentation, introduced in [25], which generalises the event structures of [20] in allowing time ordering of events to be a pre-order rather than a partial order, thereby allowing the representation of simultaneity as well as concurrency. A behavioural presentation is a quadraple (O, Π, E, λ) - where O is a set of occurrences, $\Pi \subseteq \wp(O)$ is a set of points, E is a set of events and $\lambda : O \to E$ is the occurrence function that associates occurrences of events with the events of which they are occurrences - which satisfies $\bigcup_{\pi \in \Pi} \pi = O$. The intuition is that each $\pi \in \Pi$ represents that point in time reached after all occurrences that constitute it have taken place. $\lambda(o) = e$ is to be read 'o is an occurrence of e'.

A software component can be associated with a behavioural presentation by exploiting a basic order theoretic property of behavioural presentations related to primes. In this context, the notion of prime refers to vectors which have a unique other vector immediately below them (see also [19]). The ordering among elements of the set of behaviours of the component is based on the relation $\lhd_B$ defined as follows. For $\underline{u}, \underline{v} \in B$, $\underline{u} \lhd_B \underline{v}$ implies that $\underline{u} < \underline{v}$ and if $\underline{w} \in B$ is such that $\underline{u} \leq \underline{w} < \underline{v}$, then $\underline{w} = \underline{u}$. All $\underline{v}$ in B which have a unique $\underline{u}$ in B immediately below them, are considered to be primes in B.

As far as component-based design is concerned, we wish to constrain components in such a way that they can be associated with a subclass of behavioural presentations, namely those that are discrete. This gurantees that i) there are no infinite ascending or descending chains of occurrences of events, with respect to time ordering, which would give rise to Zeno type paradoxes, ii) there are no 'gaps' in the time continuum and iii) there is an initial point where nothing has happened. We also wish to ensure that the behavioural presentation for each component contains one occurrence for each call to an operation to one of its interfaces. This can be guaranteed by a property called local left closure, which we now define.

Definition 2.6 *Suppose that $c = (\Sigma, B)$ is a component. We shall say that c is locally left closed iff whenever $\underline{u} \in B$ and $i \in I_\Sigma$ and $x \in \beta_\Sigma(i)$ such that $\Lambda < x < \underline{u}(i)$, then there exists $\underline{v} \in B$ such that $\underline{v} \leq \underline{u}$ and $\underline{v}(i) = x$.*

If c is discrete and locally left closed, then we shall say that c is normal.

Effectively, the local left closure property ensures that there will be a distinct prime in B for each simultaneity class of calls to operations received or transmitted, during the time of this behaviour. This resolves ambiguities that may arise from not having enough points to describe the course of the behaviour in question; not the start or the

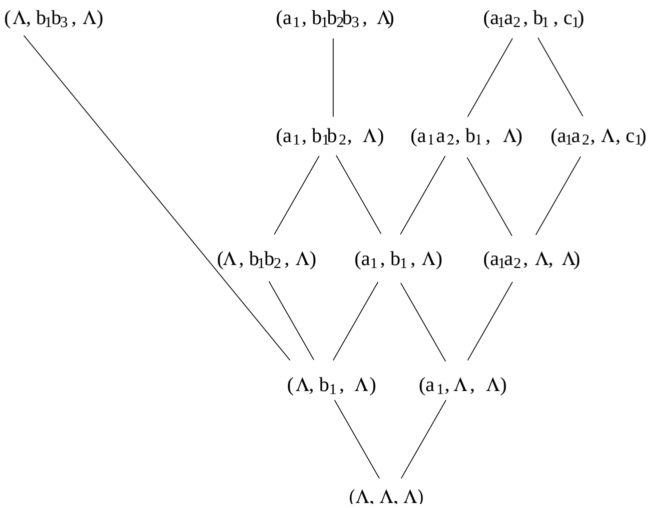
$(\Lambda, b_1b_3, \Lambda)$
$(a_1, b_1b_2b_3, \Lambda)$
(a_1a_2, b_1, c_1)
(a_1, b_1b_2, Λ)
(a_1a_2, b_1, Λ)
(a_1a_2, Λ, c_1)
$(\Lambda, b_1b_2, \Lambda)$
(a_1, b_1, Λ)
$(a_1a_2, \Lambda, \Lambda)$
(Λ, b_1, Λ)
(a_1, Λ, Λ)
$(\Lambda, \Lambda, \Lambda)$

Figure 2. Order structure of elements in B

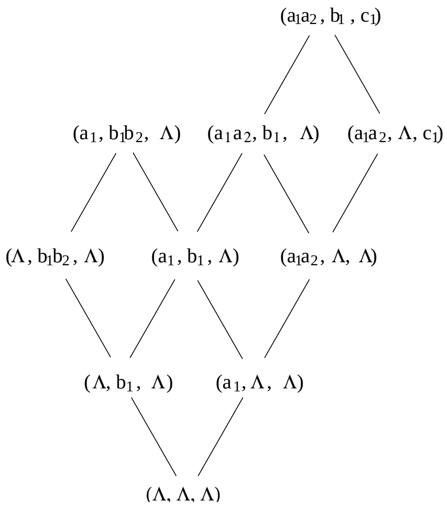
(a_1a_2, b_1, c_1)
(a_1, b_1b_2, Λ)
(a_1a_2, b_1, Λ)
(a_1a_2, Λ, c_1)
$(\Lambda, b_1b_2, \Lambda)$
(a_1, b_1, Λ)
$(a_1a_2, \Lambda, \Lambda)$
(Λ, b_1, Λ)
(a_1, Λ, Λ)
$(\Lambda, \Lambda, \Lambda)$

Figure 3. Discreteness of CMenu component

.

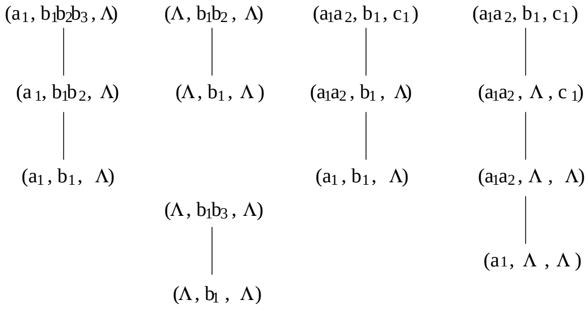
$(a_1, b_1b_2b_3, \Lambda)$
$(\Lambda, b_1b_2, \Lambda)$
(a_1a_2, b_1, c_1)
(a_1a_2, b_1, c_1)
(a_1, b_1b_2, Λ)
(Λ, b_1, Λ)
(a_1a_2, b_1, Λ)
(a_1a_2, Λ, c_1)
(a_1, b_1, Λ)
(a_1, b_1, Λ)
$(a_1a_2, \Lambda, \Lambda)$
$(\Lambda, b_1b_3, \Lambda)$
(a_1, Λ, Λ)
(Λ, b_1, Λ)

Figure 4. Local left closure of CMenu component

end, but the 'gaps' in between. In order to provide a precise description of a discrete behaviour we require that every occurrence of an event is 'recorded' in the set of behaviours of the component. This implies the presence of a distinct prime element in B for each simultaneity class of incidences, and for each appropriate interface.

Example 2.2 In this example, we examine discreteness and local left closure of the CMenu component of example 2.1. The ordering structure of the elements in B is shown in Figure 2 and we shall use it to illustrate the idea of normality for the CMenu component.

It can be seen in the Hasse diagram of Figure 2 that $(\Lambda, b_1b_3, \Lambda)$, $(a_1, b_1b_2b_3, \Lambda)$ and (a_1a_2, b_1, c_1) are the maximal behaviour vectors of the component, in the sense that they do not describe earlier behaviour than any other vector in B. Likewise, vector $(\Lambda, \Lambda, \Lambda)$ is the minimal behaviour vector representing behaviour of the component in which nothing has happened.

Based on Figure 2, we examine the discreteness property of the CMenu component. In order to do so, we concentrate on vectors $\underline{v}$ in B with at least two distinct incomparable immediate predecessors. They, together with their predecessors should constitute (finite) lattices, according to Definition 2.5 of discreteness. That this is so, is best illustrated diagrammatically. By inspection, we have the case depicted as a Hasse diagram in Figure 3, which exhibits the characteristic structure of a lattice.

It can be seen in the illustration of Figure 3 that we only include those vectors of B with at least two distinct immediate predecessors. To see that (a_1a_2, b_1, c_1), (a_1, b_1b_2, Λ), (a_1a_2, b_1, Λ) and (a_1, b_1, Λ) are such vectors, focus on the four rhombus-like shapes formed in Figure 2. The Hasse diagram of Figure 3 then, demonstrates that together with their predecessors they constitute lattices. Indeed, the least upper bound and the greatest lower bound of the distinct immediate predecessors exist and are in B, in all four cases. This implies that the CMenu component is discrete (in conformance with Definition 2.5).

For local left closure, we concentrate on those vectors in B with at least one component containing a coordinate with length greater than one and examine their predecessors. Again, we feel it is easier to demonstrate that the property holds diagrammatically.

Figure 4 demonstrates that for each vector in B with at least two events in one of its coordinates there is some other vector in B which has either the same sequence of events, at that specific coordinate, or the same reduced by one event. This implies that the CMenu component is locally left closed.

Having established both discreteness and local left closure for the CMenu component, we have shown that it is normal. Consequently, its set of behaviours can be associated with a behavioural presentation (see [19, 26] for this association), modelling the potential behaviour of the CMenu component.

In terms of justifying such a constraint in a clear and accessible way to the non-theoretical designer of component-based systems, we feel that the diagrammatic representation of the normality property could not be considered as a success factor on its own, although designers seem to be keen on working with diagrams rather than formalisms or mathematics.

The process of checking a software component against normality starts with forming the set B of behaviours. In doing so, a component designer should include only those vectors that describe desired behaviour. Then, while checking for discreteness and local left closure of the component, other vectors might need to be added in the set B (to satisfy the normality property). The component designer is to decide whether the additional vectors represent desired behaviour or not. If they do not, then the design should be refined to ensure that the component shall not exhibit instances of such behaviour, under any circumstances, in the course of achieving the desired behaviour.

In terms of our example, a component designer would most likely not include (a_1a_2, b_1, Λ) in the set of behaviours B as this vector does not represent desired behaviour. Recall that operation a_2 has as an 'immediate' consequence the invocation of c_1. A call to operation b_1 before c_1 actually occurs, is likely to cause the component to exhibit pathological behaviour (in that a search for a channel has not been completed while the user requests to fine tune the signal). While checking for discreteness, vector (a_1a_2, b_1, Λ) would be added to make the component discrete. Therefore, the designer would become aware that in achieving (a_1a_2, b_1, c_1) the component might experience pathological behaviour (i.e. (a_1a_2, b_1, Λ)) which might leave it in an inconsistent state. Based on this indication, the component design could then be refined accordingly.

3. Formalisation of Component Composition

In this section, we discuss the major theme of composition of components. First, we present a mathematical framework for combining components and then, we examine the effect of their composition.

Current component technologies such as the OMG's CORBA Component Model, Microsoft's COM+ and Sun's EJB support rapid assembly of systems from pre-fabricated software components. However, there is little, if any, support for reasoning about the resulting system until its parts have been combined, executed and tested. To address this issue and thus, facilitate predictable assembly of component-based systems there must be some way to formally reason about the behaviour of the composite based on properties of the individual components.

Naturally, composition takes place via complementary interfaces, that is, interfaces that are required by one component and provided by another. We assume disjoint sets of 'requires' and 'provides' interfaces for each of the components. As a result, a condition is required on the set of interfaces of a component; its elements must be pairwise consistent.

Definition 3.1 *Suppose that Σ_1, Σ_2 are sorts. We say that Σ_1 and Σ_2 are consistent and we write $\Sigma_1 \downarrow \Sigma_2$ if and only if*

- $P_{\Sigma_1} \cap P_{\Sigma_2} = \emptyset$
- $R_{\Sigma_1} \cap R_{\Sigma_2} = \emptyset$
- $\forall i \in I_{\Sigma_1} \cap I_{\Sigma_2} : \beta_{\Sigma_1}(i) = \beta_{\Sigma_2}(i)$

Suppose that c_1 and c_2 are components where $c_j = (\Sigma_j, B_j)$, each j. Then, c_1 and c_2 are consistent, and we write $c_1 \downarrow c_2$, if Σ_1, Σ_2 are consistent.

Definition 3.2 *Suppose that Σ_1 and Σ_2 are consistent sorts. Define $\Sigma_1 \oplus \Sigma_2 = \Sigma$ where,*

- $P_\Sigma = (P_{\Sigma_1} \cup P_{\Sigma_2}) \backslash (R_{\Sigma_1} \cup R_{\Sigma_2})$
- $R_\Sigma = (R_{\Sigma_1} \cup R_{\Sigma_2}) \backslash (P_{\Sigma_1} \cup P_{\Sigma_2})$
- $\beta_\Sigma(i) = \beta_{\Sigma_j}(i)$ *wherever* $i \in I_{\Sigma_j}, j = 1, 2$ *(recall that $I_{\Sigma_j} = P_{\Sigma_j} \cup R_{\Sigma_j}$ from Definition 2.1)*

Lemma 3.1 *Suppose that Σ_1, Σ_2 are consistent sorts, then $\Sigma_1 \oplus \Sigma_2$ is a sort.*

Proof (sketch). We first prove that β is a well defined function. Since $I_\Sigma \subseteq I_{\Sigma_1} \cup I_{\Sigma_2}$, it suffices to show that if $i \in I_{\Sigma_1} \cap I_{\Sigma_2}$ then $\beta_{\Sigma_1}(i) = \beta_{\Sigma_2}(i)$ which is precisely point (3) of Definition 3.1. Finally, we note that $P_\Sigma \cap R_\Sigma = ... = \emptyset$ which completes the proof (see [27]).

Informally, the above definitions say that the sort of the resulting system is formed from those of the components by eliminating all interfaces participating in internal communication. This is illustrated in Figure 5 using the notation of [21, 7] for the components of example 2.1. Composition takes place via IDetectSignal interface which is a 'provides' interface of CTuner and a 'requires' interface of CMenu. Notice that it is hidden in the resulting composite component CsMenuTuner which is stereotyped by <<composite

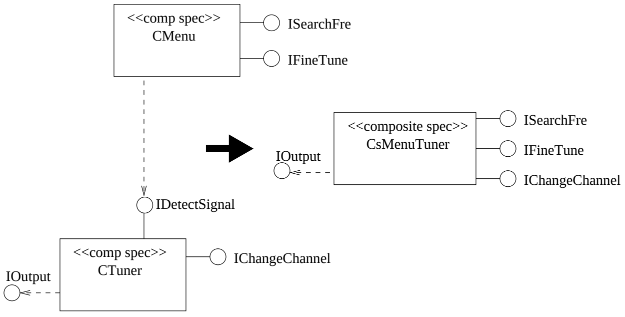

Figure 5. Composition of CMenu and CTuner

spec>>. The other interfaces remain visible and comprise the set of 'requires / provides' interfaces for the composite of CMenu and CTuner.

As far as the dynamics are concerned, we motive the definition as follows. In any behaviour of the composite system, each component c_j will have engaged in a piece of behaviour $\underline{v}_j$. If i is an interface common to both c_j and c_k, then it will be a provided interface of one and a required interface of the other. Without loss of generality, suppose that it is a provided interface of c_j and a required interface of c_k. Then, $\underline{v}_j(i)$ represents the sequence of calls to operations made from c_j to c_k through interface i, which (assuming no delays) is precisely behaviour $\underline{v}_k(i)$.

Definition 3.3 *Suppose I_1, I_2 are sets of interfaces and $\beta_{\Sigma_j} : I_j \to \wp(Op)$, $j = 1, 2$. We shall say that vectors $\underline{u}_j \in V_{\beta_{\Sigma_j}}$ are consistent, and we write $\underline{u}_1 \downarrow \underline{u}_2$ if*

$$\underline{u}_1 \lfloor_{I_1 \cap I_2} = \underline{u}_2 \lfloor_{I_1 \cap I_2}$$

where $f \lfloor X$ denotes the restriction of function f to X. Define,

$$\underline{u}_1 \oplus \underline{u}_2 = (\underline{u}_1 \cup \underline{u}_2) \lfloor_{I_1 \triangle I_2}$$

where $I_1 \triangle I_2$ is the symmetric difference of I_1 and I_2, defined to be $(I_1 \backslash I_2) \cup (I_2 \backslash I_1)$.

Now, we can give a formal definition of composition of components.

Definition 3.4 *Suppose that c_1, c_2 are consistent components, where $c_j = (\Sigma_j, B_j)$, each j. Then, we define $c_1 \oplus c_2 = (\Sigma, B)$ where,*

- $\Sigma = \Sigma_1 \oplus \Sigma_2$
- $B = B_1 \oplus B_2$ *where* $B_1 \oplus B_2 = \{\underline{v} \in V_\Sigma |$ $\exists \underline{u}_1 \in B_1, \exists \underline{u}_2 \in B_2 : \underline{u}_1 \downarrow \underline{u}_2 \wedge \underline{v} = \underline{u}_1 \oplus \underline{u}_2\}$

It is straightforward to show that $c_1 \oplus c_2 = (\Sigma, B)$ is a component whenever c_1, c_2 are consistent components. Indeed, Σ is a sort by Lemma 3.1 and $B \subseteq V_\Sigma$ by definition.

[27] contains the complete proofs of the above results and establishes the algebraic properties of composition. For instance, '$\oplus$' is commutative since $c_1 \oplus c_2 = c_2 \oplus c_1$ whenever c_1, c_2 are consistent components. Note that using a technical lemma [27] it is possible to combine a third sort with the composite sort of two components, thus allowing further composition of the composite with another component (or another composite). In terms of mathematical theory, it can be shown that, for $\Sigma_1, \Sigma_2, \Sigma_3$ with $\Sigma_j \downarrow \Sigma_k, j \neq k$ we have $\Sigma = (\Sigma_1 \oplus \Sigma_2) \oplus \Sigma_3 = \Sigma_1 \oplus (\Sigma_2 \oplus \Sigma_3) = (P_\Sigma, R_\Sigma, \beta_\Sigma)$, where P_Σ, R_Σ and β_Σ are defined in a way similar to that of Definition 3.2.

Example 3.1 In this example, we apply the formalism introduced above to describe the composition of CMenu and CTuner components of the previous examples. We assume that CTuner has also been formally specified in the way CMenu was in example 2.1. The 'requires' and 'provides' interfaces of the two components are in the center of attention now and thus are written using their full names (as these appear in Figure 1) rather than their abbreviations used in previous examples.

Referring to Definition 3.1, the two components have no 'provides' and no 'requires' interfaces in common. Thus, $P_{\Sigma_M} \cap P_{\Sigma_T} = \emptyset$ and $R_{\Sigma_M} \cup R_{\Sigma_T} = \emptyset$. However, they do have an interface in common; IDetectSignal is a requires interface of CMenu and a provides interface of CTuner, as can be seen in Figure 5. Thus, $I_{\Sigma_M} \cap I_{\Sigma_T} = \{IDetectSignal\}$, for which $\beta_{\Sigma_M}(IDetectSignal) = \{c_1\} = \beta_{\Sigma_T}(IDetectSignal)$ where c_1 denotes a call to operation *detecting()* as in example 2.2.

The composition $c_M \oplus c_T$, where $c_M = (\Sigma_M, B_M)$ denotes the CMenu component and $c_T = (\Sigma_T, B_T)$ denotes the CTuner component, is defined by $c = c_M \oplus c_T = (\Sigma, B)$ where

- $\Sigma = \Sigma_T \oplus \Sigma_M$
- $B = B_T \oplus B_M$

The sort Σ of c is the composite sort of the sorts Σ_M and Σ_T and is obtained as follows.

$P_\Sigma = (P_{\Sigma_M} \cup P_{\Sigma_T}) \backslash (R_{\Sigma_M} \cup R_{\Sigma_T}) =$
$= \{ISearchFre, IFineTune, IChangeChannel\}$

Note that IDetectSignal does not appear in P_Σ, though it is in P_{Σ_T}, because it also belongs to R_{Σ_M}.

$R_\Sigma = (R_{\Sigma_M} \cup R_{\Sigma_T}) \backslash (P_{\Sigma_M} \cup P_{\Sigma_T}) = \{IOutput\}$

Note that IDetectSignal does not appear in R_Σ because it belongs to P_{Σ_T}.

The fuction β_Σ satisfies $\beta_\Sigma(i) = \beta_{\Sigma_k}(i)$ wherever $i \in I_{\Sigma_k}, k = M, T$ refers to all 'free' interfaces (i.e. non-connected interfaces) of the composite component c. For instance, in the case of interface IFineTune,

$\beta_{\Sigma}(IFineTune) = \beta_{\Sigma_M}(IFineTune) = \{b_1, b_2, b_3\}$ since $IFineTune \in I_{\Sigma_M}$. Recall that function β_{Σ} associates an interface with the set of all possible calls to operations on that interface.

The set of behaviours B of the composite component contains all vectors $\underline{v}$ for which there exist some vector $\underline{u}_M$ in B_M and some vector $\underline{u}_T$ in B_T such that:

- $\underline{u}_M \oplus \underline{u}_T$ refers to behaviour at the non-connected interfaces and is either $\underline{u}_M$ or $\underline{u}_T$ depending on which component the interface in question belongs to.
- $\underline{u}_M \downarrow \underline{u}_T$ indicates behaviour at the connected interface IDetectSignal of the two components. For this interface, $\underline{u}_M \lfloor_{IDetectSignal} = c_1 = \underline{u}_T \lfloor_{IDetectSignal}$.

In the above expression, c_1 refers to the one element sequence (notice that there are no curly brackets) of calls to operations made to IDetectSignal. In contrast, c_1 in the expression

$$\beta_{\Sigma_T}(IDetectSignal) = \{c_1\} = \beta_{\Sigma_M}(IDetectSignal)$$

we examined earlier in this example refers to the set of calls to operations associated with the IDetectSignal interface.

In further explanation, a frequency search is requested by CMenu via a call to operation c_1 that enables CTuner to perform the frequency search, e.g. by detecting a signal in the available bandwidth. Therefore, the behaviour described by $\underline{u}_M$, restricted to interface IDetectSignal, consists of a call to operation c_1, in our simplified example, and is precisely the behaviour also described by $\underline{u}_T$ at interface IDetectSignal.

4. Normality of the Composite System

Based on Definition 3.2 and Definition 3.4 we have formally defined a notion of composition of components. Essentially, the sort of the resulting system is defined to be the composite of the components' sorts. The dynamics of the system reflect the fact that a behaviour involves behaviours from of each component and that these must agree on shared / connected interfaces.

In Section 2 we considered constraints on the set of behaviours of a single component that ensure it is well behaved; that it is *normal.* In this section, we concentrate on the effects of composition on normal components and in particular, preservation of the normality property.

First, we define a notion of compatibility among components.

Definition 4.1 *Suppose that $c_1 = (\Sigma_1, B_1)$ and $c_2 = (\Sigma_2, B_2)$ are components. Then, they are compatible if and only if*

1. *c_1 and c_2 are consistent*
2. *If $\underline{v}_1 \in B_1$ and $\underline{v}_2 \in B_2$ such that $\underline{v}_1 \downarrow \underline{v}_2$ then*
   - *If $\underline{u}_1 \in B_1$ such that $\underline{u}_1 \leq \underline{v}_1$ then $\exists \underline{u}_2 \in B_2$ such that $\underline{u}_2 \leq \underline{v}_2$ and $\underline{u}_1 \downarrow \underline{u}_2$*
   - *If $\underline{u}_2 \in B_2$ such that $\underline{u}_2 \leq \underline{v}_2$ then $\exists \underline{u}_1 \in B_1$ such that $\underline{u}_1 \leq \underline{v}_1$ and $\underline{u}_1 \downarrow \underline{u}_2$*
   - *If $\underline{u} \in B_1 \oplus B_2$ and $\underline{u} \leq \underline{u}_1 \oplus \underline{u}_2$ then $\exists \underline{u}_1 \in B_1$ and $\underline{u}_2 \in B_2$ such that $\underline{u}_j \leq \underline{v}_j$, each j, and $\underline{u} = \underline{u}_1 \oplus \underline{u}_2$*
3. *If $\underline{w}_j, \underline{w'}_j \in B_j$, each j, such that $\underline{w}_1 \downarrow \underline{w}_2, \underline{w'}_1 \downarrow \underline{w'}_2$ and $\underline{w}_1 \oplus \underline{w}_2 = \underline{w'}_1 \oplus \underline{w'}_2$ then for each j, $\underline{w}_j \sqcap \underline{w}_j \in B_j$.*

Based on the above definition, it can be shown that the composite $c_1 \oplus c_2$ is locally left closed whenever c_1, c_2 are locally left closed and compatible components.

Lemma 4.1 *If c_1 and c_2 are compatible components which are locally left closed, then so is $c_1 \oplus c_2$.*

Proof. Let $\underline{v} \in B_1 \oplus B_2$ and let $i \in I_{\Sigma_1} \triangle I_{\Sigma_2}$ and let $\Lambda < x < \underline{v}(i)$, then $\underline{v} = \underline{v}_1 \oplus \underline{v}_2$ for $\underline{v}_1 \in B_1$ and $\underline{v}_2 \in B_2$. Without loss of generality let $i \in I_{\Sigma_1} \backslash I_{\Sigma_2}$ so that $\underline{v}(i) = \underline{v}_1(i)$. By local left closure of c_1, there exists $\underline{u}_1 \in B_1$ such that $\underline{u}_1 \leq \underline{v}_1$ and $\underline{v}(i) = x$. By Definition 4.1, there exists $\underline{u}_2 \in B_2$ such that $\underline{u}_2 \leq \underline{v}_2$ and $\underline{u}_1 \downarrow \underline{u}_2$. So $\underline{u}_1 \oplus \underline{u}_2 \in B_1 \oplus B_2$ and $\underline{u}_1 \oplus \underline{u}_2 \leq \underline{v}$ and $(\underline{u}_1 \oplus \underline{u}_2)(i) = x$ which means precisely that $c_1 \oplus c_2$ is locally left closed.

In order to prove that the normality property holds for the composite, we must further show that $c_1 \oplus c_2$ is discrete. To do that, we need the following lemma.

Lemma 4.2 *Suppose that c_1 and c_2 are compatible normal components and $\underline{u}, \underline{v}, \underline{w} \in B_{\Sigma_1} \oplus B_{\Sigma_2}$ such that $\underline{u}, \underline{v} \leq \underline{w}$, then*

- $\underline{u} \sqcup \underline{v} \in B_{\Sigma_1} \oplus B_{\Sigma_2}$
- $\underline{u} \sqcap \underline{v} \in B_{\Sigma_1} \oplus B_{\Sigma_2}$

Proof. See Lemma 4.3 and Lemma 4.4 in [27].

Finally, the main result of this section.

Theorem 4.1 *If c_1 and c_2 are compatible normal components, then $c_1 \oplus c_2$ is normal.*

Proof. By Lemma 4.1 (local left closure) and Lemma 4.2 (discreteness).

Therefore, we have argued that under certain conditions, mainly captured by the notion of compatible components in Definition 4.1, two normal components can be put together and the resulting system shall also be normal.

5. Conclusions and Future Work

In this paper, we presented the foundations of an abstract model for the composition of components, at a semantic modelling level. We described an arguably liberal model for the behaviour of a single component and established conditions on the model (i.e. normality) which ensure that the associated behavioural presentation is discrete and the potential behaviour of the component in hand can be captured. Based on the formal definition of composition, we examined the effect of combining components and derived conditions which may be used to guide composition as they guarantee that the composite of two normal components is also normal. Therefore, we argue that the proposed abstract component model allows for formal reasoning about properties of the composite based on properties of the individual components.

The mathematical landscape of this work consists of a wide variety of concurrency theories, from Mazurkiewicz traces [15] to event structures [20] to process algebras [5, 10, 17, 18] and is thus located within established theoretical computer science. However, up to this point, our work has been mostly theoretical. If this theory is to be of any practical use then it must be presented in a way accessible to the non-theoretician. For this reason, we are looking into pragmatic extensions of the UML [7, 16] as a possible medium for transferring our theoretical results to practice by embedding them in this industrially well-known standard.

Component behaviour could be described by employing (a subset of) the UML [21] diagrams, namely the collaboration, sequence and statechart diagrams. These are primarily used for capturing dynamic aspects of objects but can also be used for component behaviour, as was (partially) demonstrated in our examples. However, there is no standard associated formalism and the UML diagrams provide a semi-formal description of behaviour. The UML includes OCL [21] which introduces logical expressions for describing constraints that complement the UML diagrams in terms of pre- and postconditions of interface operations. Yet, OCL seems to lack the appropriate expressiveness to describe provides / requires dependencies, also called component contracts [28], as it is applied to the class level rather than a higher software unit level such as components or frameworks. This is tackled in [12] and [13] by using a Catalysis [9] like notation to describe component interactions and frameworks, respectively. Work is in progress in this area and especially in increasing the expressive power of OCL (see [4]), in order to aid designers in writing specifications for certain aspects of a system under dynamic interaction conditions. Possible correspondence between results of this work and the temporal relations derived from behavioural presentations in our model needs to be further investigated.

Another interesting approach to formalising software components is that of [11] which describes a distributed logical framework for formalising components and their composition. The initial set out is quite different to our model since [11] introduces a module distributed temporal logic, M_{DTL}, for inter- and intra-module communication which can be also adopted for components in a straightforward manner [12]. However, similarly to our approach, [11] also considers an event structure -like object, called labelled prime event structures, used to provide an operational semantics to M_{DTL}. In this way, [11] can use event structures as the basis for deriving causality and conflict relations to address non-determinacy and concurrency among occurrences of events. As for composition, it is addressed through an elegant categorical construction.

We illustrated our approach by means of a simple example, but it is also important that we expose our theory to real-life case studies. As well as revealing possible deficiencies of the theory, a reasonable case study would inevitably place focus on more practical aspects of component-based design. For instance, we are interested in relaxing Definition 4.1 of compatible components which seems to be quite strong as it stands. Currently, we are also investigating approaches to describing component interactions such as the use of session types [29] or interaction patterns [3] which tend to sacrifice expressiveness in order to achieve computational tractability. We envisage embedding our mathematical theory in similar, more practical approaches.

It would also be interesting to determine ways in which the tuples of sequences, used to describe component behaviour at the interfaces, could be enhanced to provide additional behavioural information. The proposed mathematical framework allows for precise modelling of the order of calls to operations at interfaces of components. One issue under further investigation is to consider not only the order of calls to operations but also the order in which their corresponding responses are received.

One possible extension of our work is to consider composition of components in terms of automata. In particular, the local left closure property has as a consequence that when two behaviour vectors $\underline{u}, \underline{v}$ are such that $\underline{u} < \underline{v}$ and $\underline{u} < \underline{w} < \underline{v}$ for no behaviour vector $\underline{w}$ then $\underline{v} = \underline{u}.\underline{e}$, where $\underline{e}$ is a vector each of whose coordinates is either a single action or the empty sequence. We may accordingly associate each component with an automaton having vectors such as $\underline{e}$ as labels on transitions. This also paves the way for employing UML statecharts to describe the behaviour of software components. The automata we have in mind can be seen as elaborations of asynchronous transition systems [1, 23] and specialisations of hybrid transition systems [25]. Further, it has been shown that every component generates such automata and every automaton generates a component. As a result, the component model may admit a complete automata theory. An automata-theoretic view of composition is currently under investigation.

Acknowledgements

We would like to thank the anonymous refereees for the useful comments.

References

[1] M. A. Bednarczyk. *Categories of Asynchronous Systems*. PhD thesis, University of Sussex, 1988.

[2] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Object Technolgy Series, Addison Wesley, 1999.

[3] A. Braccialli, A. Brogi, and F. Turini. Coordinating Interaction Patterns. In *Proceedings of SAC'01*. ACM Press, 2001.

[4] J. Bradfield, J. Küster Filipe, and P. Stevens. Enriching OCL Using Observational mu-Calculus. In R. D. Kutsche and H. Weber, editors, *Proceedings of Fundamental Approaches to Software Engineering (FASE 2002)*, volume 2306 of *Lecture Notes in Computer Science*, pages 203–217. Springer Verlag, 2002.

[5] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A Theory of Communicating Sequential Processes. *Journal of the ACM*, 31(3):560–599, 1984.

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

[7] J. Cheesman and J. Daniels. *UML Components*. Component Software Series, Addison Wesley, 2001.

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

[9] D. F. D'Souza and A. C. Wills. *Objects, Components and Frameworks with UML: The Catalysis Approach*. Object Technology Series. Addison wesley, 1999.

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

[11] J. Küster Filipe. Fundamentals of a Module Logic for Distributed Object Systems. *Journal of Functioning and Logic Programming*, 2000(3), March 2000.

[12] J. Küster Filipe. A Logic-Based Formalisation for Component Specification. *Journal of Object Technology, special issue: TOOLS USA 2002 Proceedings*, 1(3):231–248, 2002.

[13] J. Küster Filipe, K. K. Lau, M. Ornaghi, K. Taguchi, H. Yatsu, and A. C. Wills. Formal Specification of Catalysis Frameworks. In *Proceedings of the 7th Asia-Pacific Software Engineering Conference*, pages 180–187. IEEE Computer Society Press, 2000.

[14] K. K. Lau. Component Certification and System Prediction: Is there a Role for Formality? In *Proceedings of ICSE'01, 4th International Workshop on Component-Based Software Engineering*, Toronto, Canada, 2001.

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

[16] S. Mellor and M. Balcer. *Executable UML*. Object Technology Series. Addison Wesley, 2002.

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

[18] A. J. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[19] S. Moschoyiannis and M. W. Shields. A Formal Approach to Software Components. *Submitted paper to Formal Aspects of Computing*, 2002.

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

[21] OMG. *Unified Modelling Language Specification, version 1.4*. OMG document formal/01-09-67, available from http://www.omg.org/technology/documents/formal/uml.htm, February 2001.

[22] M. W. Shields. Adequate Path Expressions. In *Proceedings of Semantics for Concurrent Computation*, volume 70 of *Lecture Notes in Computer Science*, pages 249–265. Springer Verlag, 1979.

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

[24] M. W. Shields. Behavioural Presentations. In de Bakker, de Roever, and Rozenberg, editors, *Linear Time, Branching Time and Partial Orders in Logics and Models for Concurrency*, volume 354, pages 671–689. Springer Verlag, 1988.

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

[26] M. W. Shields and D. Pitt. Component-Based Systems I: Theory of a Single Component. Technical Report SCOMP-TC-01-01, Department of Computing, University of Surrey, 2001.

[27] M. W. Shields and D. Pitt. Component-Based Systems II: Composition of Components. Technical Report SCOMP-TC-03-01, Department of Computing, University of Surrey, 2001.

[28] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley, 1997.

[29] A. Valecillo, T. Vasconcelos, and A. Ravara. Typing the Behaviour of Objects and Components using Session Types. *Fundamenta Informaticae*, XX(2002):1–15, 2002.

[30] 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.

[31] 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.