A Set-Theoretic Framework for Component Composition

Sotiris Moschyiannis* † and Michael W. Shields*

Department of Computing
University of Surrey
Guildford, Surrey, GU2 7XH, England
s.moschyiannis@eim.surrey.ac.uk

Abstract. Modern software systems become increasingly complex as they are expected to support a large variety of different functions. We need to create more software in a shorter time, and without compromising the quality of the software. In order to build such systems efficiently, a compositional approach is required. This entails some formal technique for analysis and reasoning on local component properties as well as on properties of the composite. In this paper, we present a mathematical framework for the composition of software components, at a semantic modelling level. We describe a mathematical concept of a 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 framework can be used for guiding the composition of components as it advocates formal reasoning about the composite before the actual composition takes place.

Keywords: components, composition, associativity, normality, order theory, vector semantics

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

*We would like to thank the anonymous referees who made several very valuable comments
† Address for correspondence: Department of Computing, University of Surrey, Guildford, Surrey, GU2 7XH, England
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 the widely used UML [22] 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 pathological or undesired behaviour when these components are used in concert.

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 [13] 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 exhibits the desired behaviour, 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’; if the conditions are satisfied the component is guaranteed to behave in predictable ways. 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 vectors of sequences where each coordinate corresponds to an interface of the component and contains a sequence of calls to operations that are associated with that interface. The idea is that by assigning such a sequence to each interface, behaviour of the component as a whole can be described. We restrict the component model by imposing certain conditions; that is, discreteness (cf. Definition 2.5) and local left closure (cf. Definition 2.6). Each component defined in this way, can be associated with an event structure -like object, called a behavioural presentation [26]. In this way, the component model can be related to a general theory of non-interleaving representation of behaviour [27].

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 is not new [24]. However, the vector language used to describe the input / output behaviour of a component in the proposed framework, differs in important respects from that in [24] and rather, is reminiscent of the use of streams in [5] 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 [5] 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 [5] 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
[5] 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-determinism,
concurrency and simultaneity as distinct phenomena.

The work presented here is along the lines of that in [18] which contains an overview of our compo-
sitional approach for software components. The present paper however elaborates on associativity of the
operation of composition and elucidates on preservation of the normality property under composition.

This paper is structured as follows. The next section describes the foundations for formalising a sin-
gle software component and introduces component properties that allow us to characterise a component
as normal. In Section 3, we outline the mathematical framework for the composition of components and
show that the operation of composition is associative. We return to the idea of normality in Section 4
where the effect of composition, in terms of preservation of the normality property, is examined. Finally,
Section 5 includes some concluding remarks and a discussion on future work.

2. Formalisation of a Single 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.

Initially, the view we take of a software component is as unrestricted as possible. In line with [8, 30]
we consider a component as being a black box whose functionality is made available to the rest of the
system only through its interfaces. This applies equally well to hardware components which communi-
cate by sending and receiving signals. This view is also consistent with that taken in the Koala component
model [23]. Koala was developed for adopting a component-oriented approach to the development of
embedded systems for consumer electronics products. Pictorially, a component in Koala is rendered as a
square box with a number of input and output ports. The idea is that components controlling individual
hardware devices have input and output ports that mirror the signal flow in hardware. For the purposes
of this paper, and the examples that follow, each port of a component is considered as being associated
with an interface and communication is established by calling 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 finite set of provides interfaces
- $R_\Sigma \subseteq I$ is a finite set of requires interfaces
- $\beta_\Sigma : P_\Sigma \cup R_\Sigma \rightarrow \wp(Op)$; hence, $\beta_\Sigma(i)$ 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 $\Sigma$ is a sort. We define $V_\Sigma$ to be the set of all functions $\gamma : I_\Sigma \rightarrow Op^*$ such that for each $i \in I_\Sigma$, $\gamma(i) \in \beta_\Sigma(i)^*$. We shall refer to the vectors of $V_\Sigma$ as $\beta_\Sigma$-vectors or simply behaviour vectors.

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

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 $\Sigma$ together with a language of behaviour vectors.

Definition 2.3. A component $c$ is a pair $(\Sigma, B)$, where

- $\Sigma$ is the sort of $c$
- $B \subseteq V_\Sigma$ is the set of behaviours of $c$.

The main concept behind employing behaviour 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 that 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 functionality of a 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 [22, 6]. The stereotype $\langle\langle$comp spec$\rangle\rangle$ 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 [11].

The CMen 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 CMen 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 detectSignal (abbreviated by $c_1$) on its required interface IDetectSignal, which is implemented by the CTuner component.

The 'call interplay' among component interfaces is considered to be synchronous. Issued requests are delivered instantaneously and no significant time elapses between issuing and receiving a request (emitting and absorbing a signal). We assume a reliable communication medium, in the sense that no messages can be lost. Messages between components are understood as operation calls between components and therefore we do not model acknowledgements. These should not prove difficult to incorporate in future, and in any case they do not severely affect the essence of our explanations.

In what follows, we apply the mathematical theory presented earlier to model the CMenu component. By Definition 2.1, $P_\Sigma = \{IFineTune, ISearchFre\}$ and $R_\Sigma = \{IDetectSignal\}$. Hence, we have $I_\Sigma = P_\Sigma \cup R_\Sigma = \{IFineTune, ISearchFre, IDetectSignal\}$ and of course, $P_\Sigma \cap R_\Sigma = \emptyset$. Function $\beta_\Sigma$ as defined in Definition 2.1 provides the set of calls to operations associated with each interface. Hence,

$$\beta_\Sigma(Iurchase) = \{a_1, a_2\}$$
$$\beta_\Sigma(IFnineTune) = \{b_1, b_2, b_3\}$$
$$\beta_\Sigma(IDetectSignal) = \{c_1\}$$

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 $\gamma$ of Definition 2.2 with $\gamma(IFnineTune) = x$, $\gamma(ISearchFre) = y$ and $\gamma(IDetectSignal) = z$, and use $A$ to denote the empty sequence, we can define the set of behaviours for the CMenu component as,

$B = \{(A, A, A), (a_1, A, A), (A, b_1, A), (a_2, a_2, A, A), (A, b_1, b_2, A, A), (a_1, b_1, b_2, A, A), (a_1, b_1, b_2, b_3, A), (a_1, a_2, b_1, 1)\}$

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 $V_\Sigma$. 
The mathematics of behaviour vectors is given in [28] and is very similar to that of [24, 27]. However, while vectors in [24, 27] describe behaviour of systems of sequential processes combined using something like the parallel composition operator $||$ of CSP [9], behaviour vectors describe behaviour of systems using something like the interleaving operator $||$ of CSP. The main technical difference may, perhaps, be seen most clearly in the relationship between vectors and associated order-theoretic structures. In the case of the synchronisation vectors of [24, 27] this relationship is independent of context. In the case of the component behaviour vectors the relationship is very much dependent on what other vectors are in the language. Using the construction described following Definition 2.5, the reader may care to contrast the order theoretic structure associated with $(a, b)$ in the language $\{(A, A), (a, A), (a, b)\}$ and $\{(A, A), (a, A), (A, b), (a, b)\}$.

In this paper, we present the fairly basic properties of behaviour vectors. If $x$ and $y$ are sequences we write $x.y$ for the concatenation of $x$ and $y$. As is well known, this operation is associative with identity $$. We also have a partial order on sequences given by $x \preceq y$ if and only if there exists $z$ such that $x.z = y$, and this partial order has a bottom element $A$. It is also well known that concatenation is cancellative, thus $z$ is unique.

Further, the set of behaviour vectors $V_\Sigma$ is a monoid with binary operation $\cdot$ and identity $A$. The behaviour $A$ assigns the empty sequence to each interface. It is also a partially ordered set (poset) with partial order $\preceq$ and bottom element $A$. The interested reader is referred to [28] where the order theoretic properties of $V_\Sigma$ 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 $V_\Sigma$.

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

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

The minimum (min) and the maximum (max) among sequences appearing in coordinates of behaviour vectors is determined by a prefix ordering defined on the set of sequences formed over $\mathcal{A}(i)$, each $i$. We write $u(i) = \min(u(i), v(i))$ if $u(i)$ is a prefix of $v(i)$. Formally we have,

\[
\min(u(i), v(i)) = \max(u(i), v(i)) = w(i), i \in \mathcal{I}_\Sigma, \quad \exists z(i) : u(i) \preceq z(i) = v(i)
\]

$u(i) = \max(u(i), v(i))$ is defined similarly. Thus, vector $\underline{w}$ in Definition 2.4 is computed by comparing the coordinates of vectors $\underline{u}$ and $\underline{v}$ pairwise and keeping the minimum each time, for point (1) of the definition, and the maximum for point (2) of the definition.

In terms of partial orders the above operations essentially give the greatest lower bound and the least upper bound of $\underline{u}, \underline{v} \in V_\Sigma$, in the usual sense of lattices and domain theory [7, 32]. Recall that if $(X, \preceq)$ is a partially ordered set [7] then the least upper bound of $x_1, x_2 \in X$, if it exists, is the least element $x \in X$ such that $x_1, x_2 \preceq x$. 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 $x \preceq x_1, x_2$. Notice that these are computed coordinate-wise for the behaviour vectors of our model.

A key property of the sets $V_\Sigma$ is that they possibly contain discrete subsets. Before introducing discreteness, we also need to define consistent completeness. We shall say that $B \subseteq V_\Sigma$ is **consistently complete** if and only if i) $A \in B$ and ii) whenever $\underline{u}_1, \underline{u}_2, \underline{w} \in B$ such that $\underline{u}_1, \underline{u}_2 \preceq \underline{w}$, then $\underline{u}_1 \sqcup \underline{u}_2 \in B$. 


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.

A few words are in order to justify why we restrict to consistently complete behaviours. The central
behavioural model of our overall formal approach to software components [28, 29, 18, 19] is that of
behavioural presentations [26]. Based on the order-theoretic properties of the set of behaviours \( B \), a
component can be associated with a behavioural presentation, as shown in [19]. The machinery necessary
for this association is described elsewhere [28, 19] and is beyond the scope of the present paper. It is
worth mentioning though that behavioural presentations, which are left-closed with respect to a relation
\(<\) (cf. Definition 2.7), are prime algebraic and consistently complete. These properties, in a certain
important respect, play a crucial part in relating a behavioural presentation to the vector language of a
component, thereby building a bridge between order-theoretic and algebraic representation of component
behaviour.

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

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

1. \( B \) is consistently complete
2. If \( \bar{u}_1, \bar{u}_2 \in B \), then \( \bar{u}_1 \sqcup \bar{u}_2 \in B \Rightarrow \bar{u}_1 \sqcap \bar{u}_2 \in B \)

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.

As mentioned earlier, a set of behaviours of a software component may be translated into a be-
behavioural presentation [26] which is a behavioural model reminiscent of the event structures model of
[20]. In fact, behavioural presentations generalise event structures in allowing time ordering of events to
be a pre-order (a reflexive and transitive relation) rather than a partial order, thereby allowing the repre-
sentation of simultaneity as well as concurrency. Using the temporal relations derived from behavioural
presentations we can determine the time ordering amongst calls to operations occurring at the interfaces
of the component.

In fact, 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. Thus, the discreteness condition of Defini-
tion 2.5. Least upper bounds and greatest lower bounds in the set of behaviours \( B \) guarantee that 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, and also that there are no ‘gaps’ in the time continuum.
Inclusion of \( A \) in \( B \) guarantees that there is an initial point in which 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 $u, u' \in B$ and $i \in I_\Sigma$ and $x \in \beta_\Sigma(i)^*$ such that $A < x < u(i)$, then there exists $u' \in B$ such that $u \leq u'$ and $u(i) = x$.

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 primal element in $B$ for each simultaneity class of calls to operations received or issued, during the time of this behaviour. The notion of primal vectors refers to vectors which have a unique other vector immediately below them. This is formally put in the following definition.

Definition 2.7. If $(\Sigma, B)$ is a component, then $\underline{z} \in B$ is primal if there exists exactly one $u \in B$ such that $u < \underline{z}$, where $u < \underline{z}$ if

1. $u < \underline{z}$

2. If $u' \in B$, then $u \leq u' < \underline{z} \implies u = u'$

The intuition behind introducing primal vectors is that such vectors are prime in the poset $(B, \leq)$, in the usual sense of lattices and domain theory [7, 32]. Recall that an element $x$ of a poset $(X, \leq)$ is prime if, whenever $U \subseteq X$ and $x \leq \bigcup U \in X$ then $x \leq u$, some $u \in U$. More details, along with the proof of the above claim, can be found in [28, 27].

The local left closure property is intended to resolve ambiguities that may arise from not having enough points to describe the course of the behaviour in question; not the start or the 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.

Local left closure also guarantees that behaviour vectors in $B$ decompose into products of vectors, each of which has at most one operation call per coordinate. These vectors correspond to simultaneity classes in the corresponding behavioural presentation and become particularly important when we attempt to establish a relation between the vector languages of components and automata. This is currently under further consideration. We return to this discussion in the concluding section of the paper.

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 the behaviour vectors $(A, b_1 b_2 A), (A, b_1 b_2 b_3 A)$ and $(a_1 a_2 b_1 c_1)$ 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 $(A, A, A)$ 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 $\underline{y}$ 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.
Figure 2. Order structure of elements in $B$

Figure 3. Discreteness of CMenu component
Figure 4. Local left closure of CMenu component

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. Behaviour vectors $(a_1, a_2, b_1, c_1), (a_1, b_1, b_2, x), (a_1, a_2, b_1, x)$ and $(a_1, b_1, x)$ are such vectors; notice the four lozenge 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 used to model the potential behaviour of the CMenu component.

From a component-based design perspective, the benefit of restricting to normal components is that potential defects in the design can be revealed. The idea is that, from an initial set of component desired behaviours provided by the component designer(s), our proposed formal framework can determine whether this set describes desired behaviour only, or on the contrary, there are other behaviours that might still emerge within the course of achieving the desired behaviour. Those missing behaviours might represent either desired or undesired behaviour. In the first case, they should have been included in the initial set of behaviours in the first place. In the second, they might be the root cause of undesired or pathological behaviour. In other words, they might allow sequences of events that represent behaviour the system is not allowed to exhibit. The design should be then refined in such a way that these behaviours do not emerge.
In terms of our example, a component designer would most likely not include \((a_1, a_2, b_1, A)\) in the set of behaviours \(B\) as this vector does not describe desired behaviour. Recall that operation \(a_2\) is to be immediately followed by invocation of \(b_1\). A call to operation \(b_1\) before \(c_1\) actually occurs, is likely to cause the component to exhibit undesirable / pathological behaviour (in that a search for a signal has not been completed while the user requests to fine tune the signal reception). While checking for discreteness, vector \((a_3, a_2, b_1, A)\) would be added to make the component discrete. Therefore, the designer would become aware that in achieving \((a_1, a_2, b_1, C)\) the component might experience pathological behaviour (i.e. \((a_1, a_2, b_1, A)\)) 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 / .NET 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 \(\Sigma_1, \Sigma_2\) are sorts. We say that \(\Sigma_1\) and \(\Sigma_2\) 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 \(\Sigma_1 \downarrow \Sigma_2\) and \(\Sigma_2 \downarrow \Sigma_1\) are consistent.

Definition 3.2. Suppose that \(\Sigma_1\) and \(\Sigma_2\) are consistent sorts. Define \(\Sigma_1 \oplus \Sigma_2 = \Sigma\) where:

- \(P_{\Sigma} = (P_{\Sigma_1} \cup P_{\Sigma_2}) \setminus (R_{\Sigma_1} \cup R_{\Sigma_2})\)
- \(R_{\Sigma} = (R_{\Sigma_1} \cup R_{\Sigma_2}) \setminus (P_{\Sigma_1} \cup P_{\Sigma_2})\)
- \(\forall i \in I_{\Sigma}, j = 1, 2\) (recall that \(I_{\Sigma_j} = P_{\Sigma_j} \cup R_{\Sigma_j}\) by Definition 2.1)

Lemma 3.1. Suppose that \(\Sigma_1, \Sigma_2\) are consistent sorts, then \(\Sigma_1 \oplus \Sigma_2\) is a sort.
Proof:
(Sketch). We first prove that $\beta$ is a well defined function. Since $\xi \subseteq I_{\Sigma_1} \cup I_{\Sigma_2}$, 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 $P_{\Sigma_1} \cap R_{\Sigma_2} = \ldots = \emptyset$ which completes the proof (see also [29]).

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 UML [22], and a pragmatic extension to UML described in [6], 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 $<<\text{composite spec}}>>$. The other interfaces remain visible and comprise the set of ‘requires / provides’ interfaces for the composite of CMenu and CTuner.

The following lemma establishes that the set of interfaces of the resulting composite component comprises all non-connected interfaces of the individual components. $X \triangle Y$ is the symmetric difference of the sets $X$ and $Y$ which is defined to be $(X \setminus Y) \cup (Y \setminus X)$.

**Lemma 3.2.** Suppose that $\Sigma_1, \Sigma_2$ are consistent sorts, then

$$I_{\Sigma_1 \oplus \Sigma_2} = I_{\Sigma_1} \triangle I_{\Sigma_2}$$

**Proof:**
Since $\Sigma_1 \downarrow \Sigma_2$ we have $P_{\Sigma_1} \setminus P_{\Sigma_2} = P_{\Sigma_1}$. Also, by definition, $P_{\Sigma_1} \cap R_{\Sigma_1} = \emptyset$, so

$$P_{\Sigma_1} \setminus (P_{\Sigma_2} \cup R_{\Sigma_2}) = (P_{\Sigma_1} \setminus P_{\Sigma_2}) \cap (P_{\Sigma_1} \setminus R_{\Sigma_2}) = P_{\Sigma_1} \cap (P_{\Sigma_1} \setminus R_{\Sigma_2}) = P_{\Sigma_1} \setminus R_{\Sigma_2} = P_{\Sigma_1} \setminus (R_{\Sigma_1} \cup R_{\Sigma_2})$$

Similarly,

- $P_{\Sigma_2} \setminus (P_{\Sigma_1} \cup R_{\Sigma_1}) = P_{\Sigma_2} \setminus (R_{\Sigma_1} \cup R_{\Sigma_2})$
- $R_{\Sigma_1} \setminus (R_{\Sigma_2} \cup P_{\Sigma_2}) = R_{\Sigma_1} \setminus (P_{\Sigma_1} \cup P_{\Sigma_2})$
- $R_{\Sigma_2} \setminus (R_{\Sigma_1} \cup P_{\Sigma_1}) = R_{\Sigma_2} \setminus (P_{\Sigma_1} \cup P_{\Sigma_2})$

and so

$$I_{\Sigma_1 \oplus \Sigma_2} = P_{\Sigma_1 \oplus \Sigma_2} \cup R_{\Sigma_1 \oplus \Sigma_2} = (P_{\Sigma_1} \cup P_{\Sigma_2}) \setminus (R_{\Sigma_1} \cup R_{\Sigma_2}) \cup (R_{\Sigma_1} \cup P_{\Sigma_2}) \cup (R_{\Sigma_1} \cup P_{\Sigma_2}) \cup (P_{\Sigma_1} \cup R_{\Sigma_2}) \cup (P_{\Sigma_1} \cup P_{\Sigma_2}) \cup (P_{\Sigma_2} \setminus (P_{\Sigma_1} \cup R_{\Sigma_2})$$

$$= (P_{\Sigma_1} \setminus (R_{\Sigma_1} \cup R_{\Sigma_2})) \cup (P_{\Sigma_2} \setminus (R_{\Sigma_1} \cup R_{\Sigma_2})) \cup (R_{\Sigma_1} \setminus (P_{\Sigma_1} \cup P_{\Sigma_2})) \cup (R_{\Sigma_2} \setminus (P_{\Sigma_1} \cup P_{\Sigma_2}))$$

$$= P_{\Sigma_1} \setminus (R_{\Sigma_1} \cup R_{\Sigma_2}) \cup (P_{\Sigma_2} \setminus (P_{\Sigma_1} \cup R_{\Sigma_1})) \cup (R_{\Sigma_1} \setminus (R_{\Sigma_2} \cup P_{\Sigma_2})) \cup (R_{\Sigma_2} \setminus (R_{\Sigma_1} \cup P_{\Sigma_2}))$$
while

\[ I_{\Sigma_1} \triangle I_{\Sigma_2} = (I_{\Sigma_1} \setminus I_{\Sigma_2}) \cup (I_{\Sigma_2} \setminus I_{\Sigma_1}) \]

\[ = ((P_{\Sigma_1} \cup R_{\Sigma_1}) \setminus (P_{\Sigma_2} \cup R_{\Sigma_2})) \cup ((P_{\Sigma_2} \cup R_{\Sigma_2}) \setminus (P_{\Sigma_1} \cup R_{\Sigma_1})) \]

\[ = (P_{\Sigma_1} \setminus (P_{\Sigma_2} \cup R_{\Sigma_2})) \cup (R_{\Sigma_1} \setminus (P_{\Sigma_2} \cup R_{\Sigma_2})) \cup (P_{\Sigma_2} \setminus (P_{\Sigma_1} \cup R_{\Sigma_1})) \cup (R_{\Sigma_1} \setminus (P_{\Sigma_1} \cup R_{\Sigma_1})) \]

Thus, we have shown that \( I_{\Sigma_1 \oplus \Sigma_2} = I_{\Sigma_1} \triangle I_{\Sigma_2} \).

As far as the dynamics are concerned, we motivate the definition as follows. In any behaviour of the composite system, each component \( c_j \) will have engaged in a piece of behaviour \( \overline{g}_j \). If \( i \) is an interface common to both \( c_j \) and \( c_k \), then it will be a provides interface of one and a requires 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, \( \overline{g}_j(i) \) represents the sequence of calls to operations made from \( c_j \) to \( c_k \) through interface \( i \), which (assuming no delays) is precisely behaviour \( \overline{g}_k(i) \).

**Definition 3.3.** Let \( c_1 = (\Sigma_1, B_1) \) and \( c_2 = (\Sigma_2, B_2) \) be components and suppose that \( I_{\Sigma_1}, I_{\Sigma_2} \) are their sets of interfaces and \( \beta_{\Sigma_j} : I_{\Sigma_j} \to \varphi(Op), j = 1,2 \). We shall say that vectors \( \underline{u}_1 \in B_1 \) and \( \underline{u}_2 \in B_2 \) are consistent, and we write \( \underline{u}_1 \downarrow \underline{u}_2 \) if

\[ \underline{u}_1 \upharpoonright I_{\Sigma_1 \cap I_{\Sigma_2}} = \underline{u}_2 \upharpoonright I_{\Sigma_1 \cap I_{\Sigma_2}} \]

where if \( f \) is a function, \( f \upharpoonright X \) denotes the restriction of function \( f \) to \( X \), in which case we define,

\[ \underline{u}_1 \oplus \underline{u}_2 = (\underline{u}_1 \cup \underline{u}_2) \upharpoonright I_{\Sigma_1 \triangle I_{\Sigma_2}} \]
where $u_1 \cup u_2 : I_{\Sigma_1} \triangle I_{\Sigma_2}$ satisfies

$$(u_1 \cup u_2)(i) = \begin{cases} u_1(i), & i \in I_{\Sigma_1} \\ u_2(i), & i \in I_{\Sigma_2} \end{cases}$$

which is well defined if $u_1 \downarrow u_2$.

As a consequence of the above definition we have the following remark.

**Remark 3.1.** Suppose that $I_{\Sigma}, I_{\Sigma_2}$ are the sets of interfaces of components $c_1, c_2$ and $u_j, u_j \in B_j$, each $j$, such that

1. $u_1 \downarrow u_2$ and $u_1 \downarrow u_2$
2. $u_1 \leq u_1$ and $u_2 \leq u_2$

then, $u_1 \oplus u_2 \leq u_1 \oplus u_2$.

**Proof:**

Let $i \in I_{\Sigma_1} \setminus I_{\Sigma_2}$. Then, by Definition 3.3, $(u_1 \oplus u_2)(i) = u_1(i)$ and $(u_1 \oplus u_2)(i) = u_1(i)$. Since $u_1 \leq u_1$ we can deduce that $(u_1 \oplus u_2)(i) \leq (u_1 \oplus u_2)(i)$.

Similarly, when $i \in I_{\Sigma_2} \setminus I_{\Sigma_1}$ we have $(u_1 \oplus u_2)(i) = u_2(i)$ and $(u_1 \oplus u_2)(i) = u_2(i)$. Since $u_2 \leq u_2$, we conclude that $(u_1 \oplus u_2)(i) \leq (u_1 \oplus u_2)(i)$.

Hence, it follows that $u_1 \oplus u_2 \leq u_1 \oplus u_2$. \hfill $\square$

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 $B_1 \oplus B_2 = \{ v \in V_{\Sigma} | \exists u_1 \in B_1, \exists u_2 \in B_2 : u_1 \downarrow u_2 \land v = u_1 \oplus u_2 \}$

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, $\Sigma$ is a sort by Lemma 3.1 and $B \subseteq V_\Sigma$ holds by definition.

The following lemma shows that the operation of composition is commutative.

**Lemma 3.3.** Suppose that $c_1, c_2$ are components, then $c_1 \downarrow c_2$ if and only if $c_2 \downarrow c_1$, and in either case $c_1 \oplus c_2 = c_2 \oplus c_1$.

**Proof:**

Definitions 3.1, 3.2, 3.3, 3.4 are all symmetric on $\Sigma_1, \Sigma_2$ or $c_1, c_2$ (see also [29]). \hfill $\square$

We now turn our attention to associativity. First we establish conditions under which $(c_1 \oplus c_2) \oplus c_3$ and $c_1 \oplus (c_2 \oplus c_3)$ are defined. Then we show that they are equal.

**Lemma 3.4.** Suppose that $c_1, c_2, c_3$ are components such that $c_j \downarrow c_k$ when $j \neq k$. Then, $c_1 \downarrow (c_2 \oplus c_3)$ and $(c_1 \oplus c_2) \downarrow c_3$. 
Proof:
If the first claim \( c_1 \downarrow (c_2 \oplus c_3) \) were true, then by interchanging the roles of the \( c_j \) we would have \( c_3 \downarrow (c_1 \oplus c_2) \). By Lemma 3.3 then, we have \( (c_1 \oplus c_2) \downarrow c_3 \) which is our second claim. Thus, it suffices to prove our first claim. Let \( c = (\Sigma, B) = c_2 \oplus c_3 \). Checking against Definition 3.1 for consistency of \( c_1 \downarrow c \) we have,

\[
P_{\Sigma_1} \cap P_{\Sigma} = P_{\Sigma_1} \cap ((P_{\Sigma_2} \cup P_{\Sigma_3}) \setminus (R_{\Sigma_2} \cup R_{\Sigma_3}))
\subseteq P_{\Sigma_1} \cap (P_{\Sigma_2} \cup P_{\Sigma_3})
= (P_{\Sigma_1} \cap P_{\Sigma_2}) \cup (P_{\Sigma_1} \cap P_{\Sigma_3})
= \emptyset
\]

which is precisely point (1) of Definition 3.1. In similar fashion,

\[
R_{\Sigma_1} \cap R_{\Sigma} = R_{\Sigma_1} \cap ((R_{\Sigma_2} \cup R_{\Sigma_3}) \setminus (P_{\Sigma_2} \cup P_{\Sigma_3}))
\subseteq R_{\Sigma_1} \cap (R_{\Sigma_2} \cup R_{\Sigma_3})
= (R_{\Sigma_1} \cap R_{\Sigma_2}) \cup (R_{\Sigma_1} \cap R_{\Sigma_3})
= \emptyset
\]

which is precisely point (2) of Definition 3.1. Finally, suppose that \( i \in I_{\Sigma_1} \cap I_{\Sigma_3} \). This implies that, either \( i \in I_{\Sigma_1} \cap I_{\Sigma_3} \) in which case \( \beta_{\Sigma_1}(i) = \beta_{\Sigma_3}(i) = \beta_{\Sigma}(i) \), or \( i \in I_{\Sigma_1} \cap I_{\Sigma_3} \) in which case \( \beta_{\Sigma_1}(i) = \beta_{\Sigma_3}(i) = \beta_{\Sigma}(i) \). We have proved that \( \Sigma_1 \downarrow \Sigma \). Hence, \( \Sigma_1 \downarrow (\Sigma_2 \oplus \Sigma_3) \). Now Definition 3.1 gives \( c_1 \downarrow (c_2 \oplus c_3) \) which completes the proof. \( \square \)

Remark 3.2. Suppose that \( \Sigma_1, \Sigma_2, \Sigma_3 \) are sorts such that \( \Sigma_j \downarrow \Sigma_k \) when \( j \neq k \), then \( I_{\Sigma_1} \cap I_{\Sigma_2} \cap I_{\Sigma_3} = \emptyset \)

Proof:
Suppose that \( i \in I_{\Sigma_1} \cap I_{\Sigma_3} \). Without loss of generality, and in view of Definition 3.1, we may assume that \( i \in P_{\Sigma_1} \cap R_{\Sigma_3} \). Now, \( i \notin R_{\Sigma_3} \) as \( \Sigma_2 \downarrow \Sigma_3 \), and \( i \notin P_{\Sigma_3} \) as \( \Sigma_1 \downarrow \Sigma_3 \). Hence, \( i \notin I_{\Sigma_3} \). \( \square \)

Before the proposition that establishes associativity of \( \oplus \), we also need the following lemma.

Lemma 3.5. Suppose that \( \Sigma_1, \Sigma_2, \Sigma_3 \) are sorts such that \( \Sigma_j \downarrow \Sigma_k \) when \( j \neq k \), then

\[
(\Sigma_1 \oplus \Sigma_2) \oplus \Sigma_3 = (P, R, \beta) = \Sigma_1 \oplus (\Sigma_2 \oplus \Sigma_3)
\]

where

- \( P = (P_{\Sigma_1} \cup P_{\Sigma_2} \cup P_{\Sigma_3}) \setminus (R_{\Sigma_1} \cup R_{\Sigma_2} \cup R_{\Sigma_3}) \)
- \( R = (R_{\Sigma_1} \cup R_{\Sigma_2} \cup R_{\Sigma_3}) \setminus (P_{\Sigma_1} \cup P_{\Sigma_2} \cup P_{\Sigma_3}) \)
- \( \beta : P \cup R \to \wp(Op) \) is given by \( \beta(i) = \beta_{\Sigma_j}(i) \), whenever \( j \in I_{\Sigma_j} \)
Proof:
(Sketch). \((\Sigma_1 \oplus \Sigma_2) \oplus \Sigma_3\) is defined by Lemma 3.4. The sorts are pairwise consistent, so by definition \(\beta_{\Sigma_j}(i) = \beta_{\Sigma_k}(i)\) whenever \(i \in I_{\Sigma_j} \cap I_{\Sigma_k}\). So \(\beta\) is well defined.

Let \(\Sigma = \Sigma_1 \oplus \Sigma_2\) and \(\Sigma' = \Sigma \oplus \Sigma_3\). We must show that \(\Sigma' = (P, R, \beta)\). We consider two cases, \(i \in P_{\Sigma}\) and \(i \in P_{\Sigma_3}\) and show that in both cases \(i \in P\). This implies that \(P_{\Sigma'} \subseteq P\).

Conversely, suppose that \(i \in P\). Again we consider two cases, \(i \in P_{\Sigma_1} \cup P_{\Sigma_2}\) and \(i \in P_{\Sigma_3}\) and we go on to show that in both cases \(i \in P_{\Sigma'}\). This implies that \(P \subseteq P_{\Sigma'}\).

Thus, we have shown that \(P = P_{\Sigma'}\). Exchanging the roles of \(P\) and \(R\) we also have that \(R = R_{\Sigma'}\). Finally, we note that \(\beta(i) = \beta_{\Sigma_j}(i) = \beta_{\Sigma'}(i)\) whenever \(i \in I_{\Sigma_j}\). In this way, we have shown that \((\Sigma_1 \oplus \Sigma_2) \oplus \Sigma_3 = (P, R, \beta) = (\Sigma_2 \oplus \Sigma_3) \oplus \Sigma_1\). Then, from Lemma 3.3 it can be concluded that \((\Sigma_2 \oplus \Sigma_3) \oplus \Sigma_1 = \Sigma_1 \oplus (\Sigma_2 \oplus \Sigma_3)\). Thus, \((\Sigma_1 \oplus \Sigma_2) \oplus \Sigma_3 = (P, R, \beta) = \Sigma_1 \oplus (\Sigma_2 \oplus \Sigma_3)\) (see [29] for the complete proof).

\[\square\]

Proposition 3.1. Suppose that \(c_1, c_2, c_3\) are components such that \(c_j \downarrow c_k\) when \(j \neq k\), then \((c_1 \oplus c_2) \oplus c_3\) and \(c_1 \oplus (c_2 \oplus c_3)\) are both defined and equal.

Proof:
Both are defined by Lemma 3.4. Let \(c = (\Sigma, B) = c_1 \oplus c_2, \hat{c} = c \oplus c_3\) and let \(\hat{c}' = (\Sigma', B') = c_2 \oplus c_3,\hat{c}' = c_1 \oplus \hat{c}'\). We must show that \(\hat{c} = \hat{c}'\). We have \(\hat{\Sigma} = \hat{\Sigma}'\) by Lemma 3.5. Hence, we must show that \(\hat{B} = \hat{B}'\).

Let \(\mathfrak{a} \in \hat{B}\), then there exists \(\mathfrak{u}_j \in B_j, j = 1, 2, 3\) such that

\[
\begin{align*}
\mathfrak{u}_1 & \downarrow \mathfrak{u}_2 \\
(\mathfrak{u}_1 \oplus \mathfrak{u}_2) & \downarrow \mathfrak{u}_3 \\
\mathfrak{u} & = (\mathfrak{u}_1 \oplus \mathfrak{u}_2) \oplus \mathfrak{u}_3
\end{align*}
\]

We shall prove that

\[
\begin{align*}
\mathfrak{u}_2 & \downarrow \mathfrak{u}_3 \\
\mathfrak{u}_1 & \downarrow (\mathfrak{u}_2 \oplus \mathfrak{u}_3) \\
\mathfrak{u} & = \mathfrak{u}_1 \oplus (\mathfrak{u}_2 \oplus \mathfrak{u}_3)
\end{align*}
\]

Suppose that \(i \in I_{\Sigma_2} \cap I_{\Sigma_3}\). Then \(i \notin I_{\Sigma_1}\), by Remark 3.2. Thus, \(i \in I_{\Sigma_2} \setminus I_{\Sigma_1} \subseteq I_{\Sigma_1} \triangle I_{\Sigma_3} = I_{\Sigma'}\) by Lemma 3.2. And now, \(\mathfrak{u}_2(i) = (\mathfrak{u}_1 \oplus \mathfrak{u}_2)(i)\). Similarly, we may conclude that \(\mathfrak{u}_3(i) = (\mathfrak{u}_1 \oplus \mathfrak{u}_3)(i)\). Hence, \(\mathfrak{u}_2(i) = \mathfrak{u}_3(i)\) and we have proved that \(\mathfrak{u}_2 \downarrow \mathfrak{u}_3\).

Let \(i \in I_{\Sigma_1} \cap I_{\Sigma_2}\). As \(I_{\Sigma_1} \cap I_{\Sigma_2} = I_{\Sigma_1} \cap (I_{\Sigma_2} \triangle I_{\Sigma_3})\), either \(i \in I_{\Sigma_1} \cap I_{\Sigma_3}\) or \(i \in I_{\Sigma_2} \cap I_{\Sigma_3}\). In the first case, \((\mathfrak{u}_2 \oplus \mathfrak{u}_3)(i) = \mathfrak{u}_2(i) = \mathfrak{u}_3(i)\), and in the second, \((\mathfrak{u}_2 \oplus \mathfrak{u}_3)(i) = \mathfrak{u}_3(i) = (\mathfrak{u}_1 \oplus \mathfrak{u}_2)(i)\). Hence, we have shown that \(\mathfrak{u}_2 \downarrow (\mathfrak{u}_2 \oplus \mathfrak{u}_3)\).

Finally, \((\mathfrak{u}_1 \oplus \mathfrak{u}_3)(i)\) and \((\mathfrak{u}_1 \oplus (\mathfrak{u}_2 \oplus \mathfrak{u}_3))(i)\) are both equal to \(\mathfrak{u}_j(i)\) where \(j\) is the unique number such that \(i \in I_{\Sigma_j}\) (\(j\) is unique by Remark 3.2). Thus, we have shown that \(\hat{B} = \hat{B}'\) which completes the proof.

\[\square\]

In other words we have shown that, under the stated conditions, the operation \(\oplus\) of composition, on both sorts and components, is associative. This means that the resulting composite can be further
composed with other components or other composites. The interested reader is referred to [29] which contains the complete proofs of the above results and establishes the algebraic properties of composition.

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.

Referring back to Definition 3.1, the two components must have no ‘provides’ and no ‘requires’ interfaces in common. Indeed, \( P_{\Sigma_M} \cap P_{\Sigma_T} = \emptyset \) and \( R_{\Sigma_M} \cap R_{\Sigma_T} = \emptyset \). However, they do have an interface in common; IDetectSignal is a requires interface of the CMenu component and a provides interface of the CTuner component, as depicted in Figure 5. Thus, \( I_{\Sigma_M} \cap I_{\Sigma_T} = \{ \text{IDetectSignal} \} \), for which \( \beta_{\Sigma_M}(\text{IDetectSignal}) = \{ c_1 \} = \beta_{\Sigma_T}(\text{IDetectSignal}) \) where \( c_1 \) denotes a call to operation detectSignal as in Example 2.1.

The composition \( c_M \oplus c_T \) of the two components, 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 \( \Sigma \) of \( c \) is the composite sort of the sorts \( \Sigma_M \) and \( \Sigma_T \) and is obtained as follows.

\[
P_{\Sigma} = (P_{\Sigma_M} \cup P_{\Sigma_T}) \setminus (R_{\Sigma_M} \cup R_{\Sigma_T}) = \{ \text{ISelectFrequency}, \text{IFineTune}, \text{IChangeChannel} \}
\]

Note that IDetectSignal does not appear in \( P_{\Sigma} \), though it is in \( P_{\Sigma_T} \), because it also belongs to \( R_{\Sigma_M} \).

\[
R_{\Sigma} = (R_{\Sigma_M} \cup R_{\Sigma_T}) \setminus (P_{\Sigma_M} \cup P_{\Sigma_T}) = \{ \text{IOutput} \}
\]

Note that IDetectSignal does not appear in \( R_{\Sigma} \) because it belongs to \( P_{\Sigma_T} \).

The function \( \beta_{\Sigma} \) satisfies \( \beta_{\Sigma}(i) = \beta_{\Sigma_T}(i) \) whenever \( i \in I_{\Sigma_k}, k = M, T \) refers to all ‘free’ interfaces (i.e. non-connected interfaces) of the composite component \( c \). For instance, in the case of interface IFineTune, we have \( \beta_{\Sigma}(\text{IFineTune}) = \beta_{\Sigma_M}(\text{IFineTune}) = \{ b_1, b_2, b_3 \} \) since \( \text{IFineTune} \in I_{\Sigma_M} \).

Recall that function \( \beta_{\Sigma} \) 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 \( u \) for which there exist some vector \( u_M \) in \( B_M \) and some vector \( u_T \) in \( B_T \) such that:

- \( u_M \oplus u_T \) refers to behaviour on the non-connected interfaces and is either \( u_M \) or \( u_T \) depending on which component the interface in question belongs to

- \( u_M \downarrow u_T \) indicates behaviour on the connected interface IDetectSignal of the two components. For this interface, \( u_M \downarrow \text{IDetectSignal} = c_1 = u_T \downarrow \text{IDetectSignal} \)

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

\[
\beta_{\Sigma_T}(\text{IDetectSignal}) = \{ c_1 \} = \beta_{\Sigma_M}(\text{IDetectSignal})
\]

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 \( \mu_M \), restricted to interface IDetectSignal, consists of a call to operation \( c_1 \), in our simplified example, and is precisely the behaviour also described by \( \mu_T \) 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, which is associative and commutative. 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 each of the components 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. Essentially, discreteness guaranteed that only a finite number of events may occur within finite time, allowed no gaps in the time continuum in which behaviour of the component cannot be captured, and ensured that there is an initial point in time in which nothing has happened. Local left closure guaranteed that every occurrence of an event (e.g. call to an operation) at an interface of the component is recorded in the set of behaviours of the component; there is a behaviour vector in \( B \) to describe it.

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 \( \mu_1 \in B_1 \) and \( \mu_2 \in B_2 \) such that \( \mu_1 \downarrow \mu_2 \) then
   - If \( \mu_1 \in B_1 \) such that \( \mu_1 \leq \mu_1 \) then \( \exists \mu_2 \in B_2 \) such that \( \mu_2 \leq \mu_2 \) and \( \mu_1 \downarrow \mu_2 \)
   - If \( \mu_2 \in B_2 \) such that \( \mu_2 \leq \mu_2 \) then \( \exists \mu_1 \in B_1 \) such that \( \mu_1 \leq \mu_1 \) and \( \mu_1 \downarrow \mu_2 \)
   - If \( \mu \in B_1 \oplus B_2 \) and \( \mu \leq \mu_1 \oplus \mu_2 \) then \( \exists \mu_1 \in B_1 \) and \( \mu_2 \in B_2 \) such that \( \mu_1 \downarrow \mu_2, \mu_1 \leq \mu_1, \mu_2 \leq \mu_2, \text{ and } \mu = \mu_1 \oplus \mu_2 \)
3. If \( \mu_j, \mu_j' \in B_j, \) each \( j \), such that \( \mu_1 \downarrow \mu_2, \mu_1' \downarrow \mu_2' \) and \( \mu_1 \oplus \mu_2 = \mu_1' \oplus \mu_2' \) then for each \( j \), \( \mu_j \uplus \mu_j' \in B_j \).

Based on the above definition, it can be shown that the composite \( c_1 \oplus c_2 \) is locally left closed whenever \( c_1 \) and \( c_2 \) 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 $\nu \in B_1 \oplus B_2$ and let $i \in I_{\Sigma_1} \bigtriangleup I_{\Sigma_2}$ and let $A < x < \nu(i)$, then $\nu = \nu_1 \oplus \nu_2$ for $\nu_1 \in B_1$ and $\nu_2 \in B_2$. Without loss of generality let $i \in I_{\Sigma_1} \setminus I_{\Sigma_2}$ so that $\nu(i) = \nu_1(i)$. By local left closure of $c_1$, there exists $u_1 \in B_1$ such that $\nu_1 \leq u_1$ and $u_1(i) = x$. By Definition 4.1, there exists $w_1 \in B_1$ such that $w_1 \leq u_1$ and $w_1 \sqsubseteq \nu_1 \oplus w_2$. Thus $u_1 \oplus w_1 \leq u_1 \oplus w_2 = \nu$, by Remark 3.1. So we have $u_1 \oplus w_1 \in B_1 \oplus B_2$ and $u_1 \oplus w_1 \leq \nu$ and $(u_1 \oplus w_1)(i) = x$ 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. The following lemma shall prove useful in establishing discreteness for the composite component.

Lemma 4.2. Suppose that $\Sigma_1, \Sigma_2$ are consistent sorts and that $u_1, v_1 \in B_1$ and $u_2, v_2 \in B_2$ such that

- $u_j \cup v_j \in B_j$, each $j$
- $u_1 \sqsubseteq u_2$ and $v_1 \sqsubseteq v_2$

then,

1. $(u_1 \cup v_1) \sqsubseteq (u_2 \cup v_2)$
2. $(u_1 \oplus u_2) \cup (v_1 \oplus v_2) \in B_1 \oplus B_2$
3. $(u_1 \oplus u_2) \cup (v_1 \oplus v_2) = (u_1 \cup v_1) \oplus (u_2 \cup v_2)$

Proof:
Let $i \in I_{\Sigma_1} \cap I_{\Sigma_2}$. Since $u_1 \sqsubseteq u_2$ and $v_1 \sqsubseteq v_2$, we have $u_1(i) = u_2(i)$ and $v_1(i) = v_2(i)$, and so

$$(u_1 \cup v_1)(i) = \max(u_1(i), v_1(i)) = \max(u_2(i), v_2(i)) = (u_2 \cup v_2)(i)$$

Thus, $(u_1 \cup v_1) \sqsubseteq (u_2 \cup v_2)$, establishing (1).

Suppose, next, that $i \in I_{\Sigma_1} \setminus I_{\Sigma_2}$. Then, $(u_1 \oplus u_2)(i) = u_1(i)$ and $(v_1 \oplus v_2)(i) = v_1(i)$. Now, $u_1 \sqsubseteq u_2 \in B_1$ which implies that $\max(u_1(i), v_1(i))$ is defined and hence, by what we have just seen, $\max((u_1 \oplus u_2)(i), (v_1 \oplus v_2)(i))$ is defined. This also holds when $i \in I_{\Sigma_2} \setminus I_{\Sigma_1}$, by symmetry. It follows that $(u_1 \oplus u_2) \cup (v_1 \oplus v_2) \in B_1 \oplus B_2$, giving (2).

Finally, if $i \in I_{\Sigma_1} \setminus I_{\Sigma_2}$, then

$$(u_1 \oplus u_2) \cup (v_1 \oplus v_2)(i) = \max((u_1 \oplus u_2)(i), (v_1 \oplus v_2)(i)) = \max(u_1(i), v_1(i)) = (u_1 \cup v_1)(i) = ((u_1 \cup v_1) \oplus (u_2 \cup v_2))(i)$$
and similarly when \( i \in I_{\Sigma_2} \setminus I_{\Sigma_1} \). It follows that
\[
(u_1 \cup u_2) \sqcup (v_1 \cup v_2) = (u_1 \sqcup v_1) \sqcup (u_2 \sqcup v_2)
\]
giving (3), which completes the proof. \(\square\)

**Lemma 4.3.** Suppose that \( c_1 \) and \( c_2 \) are compatible normal components and \( u, v, w \in B_1 \oplus B_2 \) such that \( u \sqcap v \leq w \), then

1. \( u \sqcup v \in B_1 \oplus B_2 \)
2. \( u \sqcap v \in B_1 \oplus B_2 \)

**Proof:**
We begin by proving point (1) of the lemma. Since \( w \in B_1 \oplus B_2 \), by definition there exist \( u_1, u_2 \in B_1 \) and \( v_1, v_2 \in B_2 \) such that \( w = u_1 \oplus v_1 = u_2 \oplus v_2 \). Since \( u \in B_1 \oplus B_2 \) and \( u \leq w = u_1 \oplus u_2 \) we can conclude from 2(c) of Definition 4.1 that there exist \( u_1 \in B_1 \) and \( u_2 \in B_2 \) such that \( u_1 \leq u_1, u_2 \leq u_2 \) and \( u = u_1 \oplus u_2 \). Similarly, there exist \( v_1 \in B_1 \) and \( v_2 \in B_2 \) such that \( v_1 \leq v_1, v_2 \leq v_2 \), and \( v = v_1 \oplus v_2 \).

Since \( u, v, \) and \( w \) are compatible normal components and \( c \) is normal, we can deduce that \( u_1 \sqcup v_1 \in B_1 \). Similarly, we can deduce that \( u_2 \sqcup v_2 \in B_2 \).

So we have shown that \( u_1 \sqcup v_1 \leq u_1 \sqcup v_2 \) and \( u_1 \sqcup v_1 \sqcap u_2 \sqcup v_2 \). Hence, by (2) of Lemma 4.2 we can deduce that \( (u_{1} \sqcup v_{1}) \sqcup (u_{2} \sqcup v_{2}) \in B_1 \oplus B_2 \). Also, by (3) of Lemma 4.2 we can conclude that \( u \sqcup v = (u_1 \oplus u_2) \sqcup (v_1 \oplus v_2) \in B_1 \oplus B_2 \). Hence, \( u \sqcup v \in B_1 \oplus B_2 \), establishing (1).

Next, we go on to prove point (2) of the lemma. Let \( u_1, u_2 \in B_1 \) and \( v_1, v_2 \in B_2 \) such that \( u_1 \sqcup u_2 = u \) and \( u_1 \sqcap u_2 = v \). As in the proof of point (1), we may deduce that \( u_1 \sqcap u_2 \in B_1 \) and \( u_2 \sqcup v_2 \in B_2 \). Since \( u_1 \sqcup u_2 \) and \( v_1 \sqcap v_2 \) both exist, we must have \( u_1(i) = u_2(i) \) and \( u_1(i) = u_2(i) \), for each \( i \in I_{\Sigma_1} \cap I_{\Sigma_2} \), so \( (u_1 \sqcap u_2)(i) = (u_2 \sqcup v_2)(i) \), for each \( i \in I_{\Sigma_1} \cap I_{\Sigma_2} \). Thus, \( u \sqcup v = u_1 \sqcup v_2 \) is defined and belongs to \( B_1 \oplus B_2 \). Finally, if \( i \in I_{\Sigma_2} \setminus I_{\Sigma_1} \), then as in the proof of Lemma 4.2 we have
\[
x(i) = (u_1 \sqcap v_2)(i) = (u \sqcap v)(i)
\]
Thus, \( u \sqcap v \in B_1 \oplus B_2 \), establishing (2). \(\square\)

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:**
We have \( A \in B_1 \) and \( B \in B_2 \) since \( c_1 \) and \( c_2 \) are normal. Thus, \( A \in B_1 \oplus B_2 \). Now \( c_1 \oplus c_2 \) is discrete by Lemma 4.3 and it is locally left closed by Lemma 4.1. Hence, \( c_1 \oplus c_2 \) is normal. \(\square\)

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 a set-theoretic model for software components and their composition, at a semantic modelling level. The static characteristics of a component were captured in terms of two disjoint sets of interfaces - those that the component provides and those it requires. Each interface was associated with a set of calls to operations that may occur on that interface. We also described an arguably liberal model for the behaviour of a single component. Behaviour of a component as a whole was modelled by a set of behaviour vectors which associate a sequence of operation calls with each interface. Furthermore, we established the normality property which ensures that the potential behaviour of the component can be captured during a period of activation. Normal components can be associated with behavioural presentations [26]. This establishes a link between the component model and a general theory of non-sequential behaviour [27].

Next, we formally defined a notion of component composition and showed that the operation of composition in our framework is commutative and associative. This has the advantage of being able to build systems out of generic components. Based on the formal definition of composition, we examined the effect of combining components and derived conditions which may be used to guide the composition of components as they guarantee that the composite of two normal components is also normal. Therefore, we argue that the proposed component model allows for formal reasoning about normality of the composite based on normality of the individual components.

The idea is that if the sequencing of events (operation calls) on the component’s interfaces is respected, then the component is guaranteed to exhibit the desired behaviour. On the contrary, if any event occurs out of the order prescribed in the set of behaviours $B$, the component might exhibit undesired or pathological behaviour. There might be other legal sequences of events, but also illegal ones. Therefore, an implementation conforms to the component specification if it results in events (operation calls) being sequenced according to the order structure of the set $B$.

In a certain important respect, the environment of the component is being constrained to desired behaviour only. This is consistent with the view taken in [1]. In fact, constrained behavior is imposed inherently when components are specified within our framework. When a component is normal, we have a precise description of the order in which it issues operation calls and the order in which it accepts operation calls from other components. When we put two such components together, preservation of the normality property under composition implies that the resulting composite respects the ordering of events in the individual components. Being associative, this form of composition then allows us to put the normal composite together with another normal component. In this way, we know precisely what behaviour to expect of the resulting system.

The mathematical landscape of this work consists of a wide variety of concurrency theories, from Mazurkiewicz traces [14] to event structures [20] to process algebras [9, 17] 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. We illustrated our approach by means of a simple running example in an attempt to make the theory relevant to more practical aspects of component-based design.

As demonstrated partly in our examples, we are looking into adopting an appropriate subset of UML with some version of our framework as a formal underpinning. Pragmatic extensions to the UML [6, 15] seem to be a promising solution in this respect. Further encouragement can be drawn from the draft adopted specification of UML 2.0 [21] in which components are also treated at the specification
level rather than solely at the implementation level of UML 1.x. The introduction of concepts such as ports, signals, connectors (especially, the assembly connector) for components in UML 2.0 seems to be consistent with the view of components taken in this paper and in other of our texts [28, 29, 18, 19]. Additionally, we are currently investigating approaches to describing component interactions such as the use of session types [31] 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.

Component behaviour could be described by a software engineer, at the design level, using (a subset of) UML [22]. Although UML was developed for modelling object-oriented rather than component-based systems, some of its notation might be useful as shown for example in [6]. The UML includes a constraint language OCL which introduces logical expressions for describing constraints in terms of pre- and postconditions [16] on interface operations. Yet, OCL is a static language and seems to lack the appropriate expressiveness to describe provides / requires dependencies, also called component contracts [30], precisely. This is tackled in [11] and [12] by using a Catalysis [8] 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 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 approach to formalising software components is that of [10, 11] which describes a distributed logical framework for formalising components and their composition. The initial set out is quite different to our model since [10] introduces a module distributed temporal logic, MDTL, for inter- and intra-module communication which can be also adopted for components in a straightforward manner [11]. To give a semantics to MDTL, [10] uses labelled prime event structures which are similar to behavioural presentations as discussed before. In this way, [10, 11] can address non-determinism and concurrency but not simultaneity as is the case with behavioural presentations.

The most closely related model is that of Broy [5]. That work originates from the functional approach to the description of communicating systems [4]. It is then extended in [5] to algebraic specific cation concepts through the introduction of operations on behaviours. The set out of the two models is quite similar, as discussed before. The main difference however is that only finite sequences are considered in our model whereas Broy’s model maps channels onto sets of finite and infinite sequences. This allows the use of fixed point theory in describing feedback loops in systems comprising data flow components. Such infinite behaviour cannot be modelled in the proposed formal framework. Nevertheless, our framework has been used to model software components for reactive embedded software used in the consumer electronics industry. On the other side, modelling infinite behaviour is something that we shall be concerned with in future work.

One possible extension of our work is to consider composition of components in terms of automata. Preliminary work has established a relationship (through behavioural presentations) between components and a certain class of automata which we may be able to exploit to relate the theory presented here with industrial standards such as statecharts. In particular, the local left closure property has as a consequence that when two behaviour vectors \( u, v \) are such that \( u < v \) and \( u < w < v \) for no behaviour vector \( w \) then \( z = w \cdot e \) where \( e \) 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 \( e \) as labels on transitions. The automata we have in mind can be seen as elaborations of asynchronous
transition systems [2, 25] and specialisations of hybrid transition systems [27]. 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 which is one step towards automated verification and eventually tool support. An automata-theoretic view of composition is currently under investigation.

References


