CHAPTER 1

A FORMAL APPROACH TO CONSTRUCTING WELL-BEHAVED SYSTEMS USING COMPONENTS

Sotiris Moschyiannis\textsuperscript{1}, Juliana Küster-Filipe\textsuperscript{2} and Michael W. Shields\textsuperscript{1}

\textsuperscript{1}Department of Computing, University of Surrey
Guildford, Surrey, GU2 7XH, England
E-mail: \{s.moschyiannis, m.shields\}@eim.surrey.ac.uk
\textsuperscript{2}School of Informatics, University of Edinburgh
The King's Buildings, Mayfield Road
Edinburgh EH9 3JZ, Scotland
E-mail: j.filipe@inf.ed.ac.uk

Present-day software systems are in increasing need of modification and evolution due to changing requirements. Component-based development constitutes a key methodology for creating large-scale, evolvable systems in a timely fashion as it advocates the (re)use of prefabricated replaceable software components. However, it is often the case that undesirable or unpredictable behaviour emerges when components are combined. This is partly due to lack of behavioural information about the individual components. In this paper, we describe a formal model for component specification which can be used to support the analysis and predictability of component composition and to identify undesirable behaviour. In our approach, component behaviour is modelled by so-called behavioural presentations, a powerful model of true-concurrency. Moreover, the framework is compositional and supports the assembly of the final system from arbitrary components.

1. Introduction

A component-based approach to software engineering emerges as a key paradigm for creating software-intensive systems. The idea is that the final system can be assembled from prefabricated software components, thus increasing the scope for reuse and replacement. However, it is often the case that systems assembled from pre-built components exhibit undesirable or unpredictable behaviour. In some occasions this is due to an unfortunate interaction between concurrency and non-determinism. Part of the problem
has to do with the fact that when a component is designed, its interface often does not contain enough or precise behavioural information. This makes it difficult to infer how the component will behave when placed in a particular context and combined with other components.

Current component technologies such as CORBA, EJB and COM/.NET facilitate the development of the final system from pre-built software components by addressing, mostly, the syntactic restrictions on component interoperability. However, these technologies seem to lack an adequate treatment of components at the specification level. In particular, they offer little support for reasoning about the final system before its parts have been combined, executed and tested as a whole which build the major motivation for our work.

We argue in favour of an a priori reasoning approach to component-based software engineering in which reasoning about composition is based on the properties of the individual components.14 Our efforts are directed at providing support for predicting properties of assemblies of components before the actual composition takes place.

Given a partial description of the behaviour of a component, most likely described by a software engineer at the design level using the Unified Modeling Language (UML),23 the proposed mathematical framework can be used, behind the scenes, for reasoning about generic issues related to components and their composition aiming to reveal potential instances of undesirable behaviour. Ideally, appropriate tool support would allow feedback to be provided to the designer indicating what the component actually does (possibly when combined with others), how it is to be deployed, and so on. The feedback produced by such tools should be given using notation well known to the designer which in our case is assumed to be UML. Considerations about the tools is, however, beyond the scope of the present paper.

In this paper we describe a mathematical framework for formalising components at a semantic modelling level. In particular, we provide a mathematical concept of a software component: the structure of a component is described by a sort while its dynamic characteristics are captured by component vectors. Essentially, these vectors are tuples of sequences of events which are used to model calls to interface operations of the component in question. More importantly, we define conditions that ensure that a component is well-behaved.

The notion of well-behavedness in this context is twofold. From a theoretical perspective, these conditions allow us to associate a compo-
rient with an event structure-like behavioural model, called a behavioural presentation. Using a behavioural presentation to capture component behaviour is advantageous in the sense that temporal relations among occurrences of events can be derived in such a way that non-determinism, concurrency and simultaneity are treated as distinct phenomena. From a more practical perspective, these conditions allow us to model the reactive behaviour of the component and establish a precise description of the time ordering between calls to component interface operations. The key point is that whenever this time ordering is respected, the component is guaranteed to behave in predictable ways.

The foundations of the mathematical framework for formalising components are essentially those described in previous work. The main difference though is that while previous work is concerned with the treatment of composition and the preservation of properties under composition, in this paper we focus on associating the model to behavioural presentations. In this way, the component model can be related to a general theory of non-interleaving representation of behaviour.

This paper is structured as follows. In Section 2 we present the foundations of the abstract mathematical model and introduce two basic conditions imposed on components: discreteness and local left-closure. These conditions inspire the definition of so-called well-behaved components. In Section 3 we associate components with behavioural presentations, which can be used to capture their potential behaviour. Section 4 provides a brief overview of composition of components within our model and shows that the proposed framework is compositional. The paper finishes with a discussion on future work and some concluding remarks in Section 5.

2. Component Model

A component, at a specification level, can be seen as a software entity that 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 this paper, we adopt the basic component concepts for specification as in Meyer and others.

Pictorially a component is a square box with a number of input and output ports. Each port is associated with an interface. We shall assume a countable infinite set \( I \) of interface names and a countable infinite set \( O_p \).
of calls to operations of those interfaces, both sets remaining fixed for the remainder of this paper.

**Definition 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
- $\beta_\Sigma : P_\Sigma \cup R_\Sigma \rightarrow \wp(O_p)$; 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 software component. For simplicity, we refer to events that may occur on an interface as operation calls. However, these could be understood in more general terms as input actions (on provided interfaces), used to model operations/procedures/methods that can be called as well as the return locations of such calls, and as output actions (on required interfaces) which are used to model operation/procedure/method calls and exceptions that may arise during execution. In this sense, the notion of a component sort resembles the alphabet structure of interface automata.

As far as the dynamic characteristics of a component are concerned, we capture these by modelling the sequences of events experienced at the interfaces of a component. For this purpose, we introduce the notion of component vectors in our model.

**Definition 2:**

Suppose that $\Sigma$ is a sort. We define $V_\Sigma$ to be the set of all functions $\mathbf{w} : I_\Sigma \rightarrow O_p^*$ such that for each $i \in I_\Sigma$, $\mathbf{w}(i) \in \beta_\Sigma(i)^*$. We shall refer to elements of $V_\Sigma$ as *component vectors*.

By $\beta_\Sigma(i)^*$ we denote the finite sequences over $\beta_\Sigma(i)$. The idea is that behaviour of the component as a whole can be described by assigning such a sequence to each interface of the component. The function $\mathbf{w}$ returns the finite sequences of events (e.g. calls to operations) on interface $i$, for each interface $i$ of the component. Putting together such sequences, one for each interface, we form (a set of) vectors of sequences. Mathematically, the set of component vectors $V_\Sigma$ is the cartesian product of the sets $\beta_\Sigma(i)^*$. Each coordinate corresponds to an interface of the component and contains a finite sequence of calls to operations on that interface. Thus, the dynamics of a component consist of a set of possible behaviours.
Components are developed under (often differing) assumptions about the context in which they will be placed. For instance, a component may be expecting certain signals to arrive consecutively while the other is generating them concurrently. Or, more generally, a component may assume that calls to interface operations occur in a specific order and it may behave as desired only when this order is respected. It is the purpose of component-based design to document such assumptions and describe the behaviour of the component in contexts which satisfy those assumptions.

Within our component model this amounts to restricting to an appropriate subset of $V_{c}$ comprising component vectors that describe intended or permitted behaviour only.

**Definition 3:** A component $c$ is a pair $(\Sigma, V)$, where

- $\Sigma$ is the sort of $c$,
- $V \subseteq V_{c}$ is the component language of $c$.

Thus, a component consists of the static structure (signature) described by a sort $\Sigma$ together with a 'language' $V$ of component vectors. Intuitively, the idea is that the component language indicates possible constraints on the order in which several operations of the component can or should be called.

It might be noteworthy, that there are a number of ways to restrict the $\Sigma$ to allowed sequences of operation calls. Schmidt and Reussner$^{24}$ attach a finite state machine to each interface, in which case the allowed sequences are essentially given by the language accepted by the machine. Moschovakis$^{21}$ advocates the use of sequence diagrams, LCSs$^{4}$ in particular, for obtaining the component language based on the partial order induced by a sequence diagram, effectively building on Küster-Filipe's$^{13,12}$ work on formalising the interactions that appear on sequence diagrams. Alternative options could be the use of regular expressions or simply a textual description (use cases) of intended behaviour. Without implying any strong preference among the various options, for simplicity we opt for the latter in the example we use to illustrate the basic ideas in this chapter.

**Example 4:**

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 UML. The component architecture of Figure 1 comprises a set of application level components together
with their structural relationships and behavioural dependencies.

![Diagram]

Fig. 1. Component specification architecture

The CMenu component establishes communication with users via its provided interfaces ISearchFre and IFineTune. 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. 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 formalism presented earlier to model the CMenu component. The provided interfaces of CMenu is given by the set $P_\Sigma = \{I\text{SearchFre}, I\text{FineTune}\}$, and the required interfaces is given by the set $R_\Sigma = \{I\text{DetectSignal}\}$. Consequently, the complete set of interfaces is given by $I_\Sigma = \{I\text{SearchFre}, I\text{FineTune}, I\text{DetectSignal}\}$ and of course, $P_\Sigma \cap R_\Sigma = \emptyset$.

Function $\beta_\Sigma$ as defined in Definition 1 provides the set of calls to operations associated with each interface. In this case we have

$$\beta_\Sigma(I\text{SearchFre}) = \{a_1, a_2\}$$
$$\beta_\Sigma(I\text{FineTune}) = \{b_1, b_2, b_3\}$$
\[ \beta_{\Sigma}(\text{IDetectSignal}) = \{c_1\} \]

Suppose that a component developer considers the expected behaviour of CMenu fulfilling the following:

- The Fine Tune option should be highlighted before the user can change the default fine tune value.
- The Search option should be highlighted before the user can request a frequency search.
- Once the user requests a search (which corresponds to invoking operation \( a_2 \) on \( \text{ISeachFree} \)) the CMenu component calls the CTuner component (calling operation \( c_1 \) on \( \text{IDetectSignal} \)).
- An occurrence of an operation call \( a_2 \) on \( \text{ISeachFree} \) should be followed immediately by an operation call \( c_1 \) on \( \text{IDetectSignal} \), and nothing should be allowed to happen in between.

Given this informal description of behaviour for the CMenu component, and if we write \((x, y, z)\) for the function \( \gamma \) of Definition 2 with \( \gamma(\text{ISeachFree}) = x \), \( \gamma(\text{IFineTune}) = y \) and \( \gamma(\text{IDetectSignal}) = z \) we obtain a set of behaviours that indicates a partial description of what would be desirable behaviour of the CMenu component. In fact such description of behaviour from a component developer would be interpreted into the following set of component vectors. We use \( A \) to denote the empty sequence.

\[
V = \{(A, A, A), (a_1, A, A), (A, b_1, A), (a_1, a_2, A, A), (A, b_1 b_2, A), (A, b_1 b_2, A), (a_1, b_1, A), (a_1, b_1 b_2, A), (a_1, a_2, A, c_1), (a_1, a_2, b_1, c_1), (a_1, b_1 b_2 b_3, A)\}
\]

In further explanation of the notation, there is an ordering between calls to operations on the same interface (within one coordinate) but not between different interfaces of the component (between two coordinates). The ordering amongst calls to operations on different interfaces is obtained from the behavourial presentation for the component, as will be demonstrated in Section 3.

Indeed, \( c = (\Sigma, V) \) is a component (recall Definition 3) where \( \Sigma = (P_{\Sigma}, R_{\Sigma}, \beta_{\Sigma}) \) is a component sort and its component language \( V \) is a subset of all possible component vectors in \( V_{\Sigma} \).

The mathematics of component vectors is given in a report by Shields\(^{28}\) and is very similar to the behaviour vectors in Shields' earlier book.\(^{26}\) The main difference is that while vectors in the book\(^{26}\) describe behaviour of systems of sequential processes combined using something like the \( || \) oper-
ator of CSP component vectors in this text describe behaviour of systems using something like the \( \| \) operator of CSP.

In this paper, we present the fairly basic properties of component vectors. More details can be found in the report.\(^8\) If \( x \) and \( y \) are sequences we write \( x \cdot y \) for the concatenation of \( x \) and \( y \). As is well known, this operation is associative with identity \( \emptyset \), where \( \emptyset \) denotes the empty sequence. We also have a partial order on sequences given by \( x \leq y \) if and only if there exists \( z \) such that \( x \cdot z = y \), and this partial order has a bottom element \( \emptyset \). It is also well known that concatenation is cancellative, thus \( z \) is unique.

Component vectors are essentially tuples of sequences. Thus, the above well-known operations on sequences can be applied to component vectors coordinatewise. Based on the ordering among sequences, the set of component vectors \( V \) becomes a partially ordered set with a partial order \( \preceq \) and bottom element \( \emptyset \). The behaviour \( \emptyset \) assigns the empty sequence to each interface.

The partial order on sequences, together with a relation that allows us to determine which vectors lie immediately underneath another vector (cf Definition 9) gives the ordering of behaviour vectors in our model. Now based on the order theoretic properties of the set \( V \), two basic operations on the set of behaviours of a component can be introduced.

**Definition 5:**

Let \( \mathbf{u} \) and \( \mathbf{v} \) be component vectors in \( V \subseteq V \). Then,

1. \( \mathbf{u} \cup \mathbf{v} \) is defined to be the vector \( \mathbf{w} \) which satisfies \( w(i) = \min(u(i), v(i)) \)
2. if it exists \( \mathbf{u} \cap \mathbf{v} \) is defined to be the vector \( \mathbf{w} \) which satisfies \( w(i) = \max(u(i), v(i)) \)

where, for an arbitrary \( i \), \( u(i), v(i), w(i) \) denote the sequence of the \( i \)-th coordinate of vector \( \mathbf{u}, \mathbf{v}, \mathbf{w} \) respectively.

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

\[
u(i) = \min(u(i), v(i)) \iff \exists z : u(i) \cdot z = u(i), v(i) \cdot z = v(i), i \in I\]

\( u(i) = \max(u(i), v(i)) \) is defined similarly.

In terms of partial orders the above operations essentially give the greatest lower bound and the least upper bound of \( \mathbf{u}, \mathbf{v} \in V \) in the usual sense of lattices and domain theory.\(^5,\(^3\)\) It is well known that if \( (X, \leq) \) is a partially
ordered set (poset) 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 coordinatewise for the behaviour vectors of our model.

Additionally, and based on the fact that if $x, y, z$ are sequences such that both $x$ and $y$ are prefixes of $z$, i.e. $x, y \leq z$, then either $x \leq y$ or $y \leq x$, we may infer for component vectors that if $u, v \leq w$ each $i$, then $u(i), v(i) \leq w(i)$ so $u(i) \leq v(i)$ or $v(i) \leq u(i)$, each $i$. We shall use this fact in the sequel without further comment.

We may now restrict to a class of a components with desirable properties. These are properties of the corresponding component language and allow us to relate our component model to a behavioural model for non-interleaving representation of behaviour. In particular, a component language, under certain conditions, may be translated into an object called behavioural presentation, introduced in Shields’ article\textsuperscript{25}, which generalises the event structures of Nielsen, Plotkin, and Winskel\textsuperscript{22} 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. Effectively, this association builds a bridge between algebraic and order-theoretic representation of component behaviour. How behavioural presentations\textsuperscript{25,26} can be used to model software components is discussed in Section 3. In what follows, we describe the conditions that allow this association between components and behavioural presentations.

A key property of the sets $V_e$ is that they possibly contain discrete subsets.

**Definition 6:** Let $c = (\Sigma, V)$ be a component. We shall say that $V$ is discrete, and consequently also that $c$ is discrete, iff $\underline{1} \in V$ and whenever $v_1, v_2, w \in V$ such that $v_1, v_2 \leq w$ then,

- $v_1 \cup v_2 \in V$
- $v_1 \cap v_2 \in V$

Note that $v_1 \cup v_2 \in V$ is understood as asserting that $v_1 \cup v_2$ is defined.

Also, note that the notion of consistent completeness underlies the definition of discreteness. In short, consistent completeness for a poset dictates that whenever two of its elements are less than a third in the set, then their least upper bound not only exists but is also in the poset. Thus, discreteness simply imposes the additional requirement that the greatest lower bound
also belongs to the poset.

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. This guarantees 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 in which nothing has happened. Exactly how i), ii) and iii) relate to the notion of discreteness shall become more clear when we discuss discrete behavioural presentations in Section 3 (cf Definition 17).

We also wish to ensure that the behavioural presentation for each component contains one occurrence for each call to operation on one of its interfaces. This can be guaranteed by a property called local left-closure, which we now define.

**Definition 7**: Suppose that \( c = (\Sigma, V) \) is a component. We shall say that \( V \) is **locally left-closed**, and consequently also that \( c \) is locally left-closed, iff whenever \( \underline{u} \in V \) and \( i \in \mathcal{I}_\Sigma \) and \( x \in \beta_\Sigma(i) \) such that \( \lambda < x < \underline{u}(i) \), then there exists \( \underline{v} \in V \) such that \( \underline{u} \leq \underline{v} \) and \( \underline{v}(i) = x \).

**Definition 8**: If a component \( c \) is discrete and locally left closed, then we shall say it is **well-behaved**.

Effectively, we require that every occurrence of an event is ‘recorded’ in the set of behaviours of the component. Otherwise, a component might be discrete but its behaviour vectors may represent the occurrence of the last operation call only, on each appropriate interface. Such situations can be eliminated by requiring the local left-closure property of components.

For a well-behaved component, and based on consequences of local left-closure in particular, we may define a further ordering between component vectors in which one is ‘immediately beneath’ the other.

**Definition 9**: Let \( \underline{u} \) and \( \underline{v} \) be behaviour vectors in \( V \subseteq V_\Sigma \). Then we define, \( \underline{u} \trianglelefteq \underline{v} \) iff

1. \( \underline{u} < \underline{v} \)
2. if \( \underline{w} \in V \) such that \( \underline{u} \leq \underline{w} < \underline{v} \) then \( \underline{w} = \underline{u} \)

Intuitively, the relation \( \trianglelefteq \) provides an ordering among the elements of \( V \), in which one is ‘immediately beneath’ the other, allowing no vector in \( V \) to exist in between them.
A Formal Approach to Constructing Well-Behaved Systems Using Components

To anticipate, when we describe a construction that maps a component language onto a behavioural presentation in Section 3, we will find this ordering particularly useful in defining primal vectors (cf Definition 18) in a component language. In short, these are vectors which have a unique other vector immediately beneath them. These turn out, as we will see, to be the complete primes in \((V, \leq)\), in the usual sense of lattices and domain theory.

We will have more to say about this in Section 3 when we characterise primes in a component language. Returning to the discussion on local left-closure, this property ensures that there will be a distinct prime in the component language for each simultaneity class of occurrences of events, e.g. calls to operations.

**Example 10:**

We now examine discreteness and local left-closure of the CMenu component of Example 4, and illustrate why these properties are important. The idea is that, from an initial set of component intended behaviours provided by the component designer(s), our proposed formal framework can determine whether this set is complete or on the contrary possible (and potentially faulty) scenarios have been omitted. In our framework, we can determine this by checking whether the set is well-behaved which corresponds to checking the properties of discreteness and local left-closure. Again, the advantages in doing so are that we may identify missing behaviours (either undesirable or simply unthought).

Based on the desirable behaviour for the CMenu component derived from the component designer’s description (see Example 4), we obtain the Hasse diagram of Figure 2 which depicts the order structure of elements in the set \(V\) of Example 4.

It can be seen in Figure 2 that \((A, b_1 b_2, A), (a_1, b_1 b_2, A)\) and \((a_1 a_2, b_1, c_1)\) are the maximal component vectors, in the sense that they do not describe earlier behaviour than any other vector in \(V\). Likewise, the empty vector \((\epsilon, \epsilon, \epsilon)\) is the minimal component 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 \(\overline{g}\) in \(V\) with at least two distinct incomparable immediate predecessors. They, together with their predecessors should constitute (finite) lattices, according to Definition 6 of discreteness. In other words, whenever two component vectors are less or equal than a third, also in \(V\), their least upper bound must exist.
and belong to the set and their greatest lower bound must also be in the set.

By careful examination it can be seen that the discreteness property is violated in Figure 2. In particular, vectors \((a_1, b_1, A)\) and \((a_2, a_2, A, A)\) are less or equal than \((a_1, a_2, b_1, c_1)\). This implies that their greatest lower bound must be in the set, and it is (see vector \((a_1, A, A)\)), but it also entails that their least upper bound should exist and belong to the set \(V\). Their least upper bound is \((a_1, b_1, A) \cup (a_1, a_2, A, A) = (a_1, A, A)\). Thus, according to our mathematical framework this vector should be added in order to make the component language \(V\), and consequently the CMenu component, discrete. By adding in vector \((a_1, a_2, b_1, A)\) we get the following set of behaviours in \(V\). Its order structure is depicted in the diagram of Figure 3.

\[
V = \{(A, A, A), (a_1, A, A), (A, b_1, A), (a_1, a_2, A, A), (A, b_1 b_2, A), (A, b_1 b_3, A),
(a_1, b_1, A), (a_1 b_2, A), (a_1 a_2, b_1, A), (a_1 a_2, A, c_1), (a_1 a_2, b_1, c_1),
(a_1 b_1 b_2 b_3, A)\}
\]

This newly added vector describes behaviour of the component in which a call to operation \(a_2\) is followed by a call to operation \(b_1\). In terms of our example, the user requests a frequency search and before the CMenu
component deals with this request (by making a call to operation \( c_1 \) on \IDetectedSignal, according to the component designer) the user highlights the \FineTune option (via a call to operation \( b_1 \) on \IFineTune). Such a sequence of events might leave the \CMenu component in an inconsistent state or even cause a system failure. Hence, the component vector \((a_1, a_2, b_1, A)\) can be regarded as describing a potential instance of undesirable behaviour.

By comparing the two diagrams for the order structure of the component language \( V \) in Figure 3 and Figure 2, it can be seen that the model is, in a sense, warning the component designer. The diagram of Figure 3 says that in the course of achieving the desirable behaviour, described by vector \((a_1, a_2, b_1, A)\), the component might exhibit the potentially undesirable behaviour described by vector \((a_1, a_2, b_1, A)\). In case this vector is indeed undesirable, some refinement of the component design is required in order to ensure that \((a_1, a_2, b_1, A)\) could be reached only through vector \((a_1, a_2, A, c_1)\) excluding any path that would involve vector \((a_1, a_2, b_1, A)\). If on the contrary vector \((a_1, a_2, b_1, A)\) represents reasonable behaviour and such a sequence of calls to operations should be allowed, then our model is detecting it and serves as a designer’s aid to find the complete set of allowed behaviours of the component.

Now based on Figure 3, we examine whether the discreteness property holds. That this is so, is best illustrated diagrammatically. By inspection,
we have the case depicted as a Hasse diagram in Figure 4, in which each subgraph below a given node exhibits the characteristic structure of a lattice.

![Hasse Diagram](image)

**Fig. 4. Discreteness of CMenu component**

It can be seen in the illustration of Figure 4 that we only include those vectors of $V$ with at least two distinct immediate predecessors. To see that $(a_1, b, c_1), (a_1, b_1, c), (a_1, a_2, b_1, A)$ and $(a_1, b_1, A)$ are such vectors, focus on the familiar lozenge shapes formed in Figure 3. The Hasse diagram of Figure 4 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 $V$, in all four cases. This implies that the CMenu component is discrete (in conformance with Definition 6).

For local left-closure, we concentrate on those vectors in $V$ with at least one component containing a coordinate with length greater than one and examine their predecessors. Figure 5 demonstrates that for each vector in $V$ with at least two events in one of its coordinates there is some other vector in $V$ 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
a  , b  ,   

(a  , b  ,   )11 2

Λ

Λ

11 2 3

Λ

1

Λ

13

12

(   , b  ,     )

Λ

(   , b  ,     )11 2

Λ

(a a  , b  , c  ) 12 1 1(a a  , b  , c  )

Fig. 5. Local left closure of CMenu component

CMenu component, we have shown that it is well-behaved. Consequently, its set of behaviours can be associated with a behavioural presentation (see the discussion in the following section for this association), modelling the potential behaviour of the CMenu component.

3. Behavioural Presentations for Components

In this section we relate the component model to behavioural presentations which comprise the central behavioural model of our approach. We shall use behavioural presentations to model component behaviour. In this way, our component model is equipped with a semantics expressive enough to capture non-determinism, concurrency as well as simultaneity as distinct phenomena.

First, we present this behavioural model and motivate our pre-occupation with discrete behavioural presentations. Then, we describe how a component can be associated with such a behavioural presentation.

We start by outlining the basic concepts behind behavioural presentations. Associated with any system is a set of events. For instance, call to operation startSearch on interface ISearchFre of the CMenu component in our examples is considered an event. When an event actually happens we talk about an occurrence of that event. Therefore, for any system there is a corresponding set E of events and a set O of occurrences of those events.

A description of the possible behaviour of a system may consist of a set of assertions concerning what events have occurred during its execution. An assertion will be valid relative to some point in the space-time of the system.
Therefore, each system is associated with a set of points. A point can be thought of as a "possible world" in which certain events have occurred. Each point is identified with the set of occurrences of events which have taken place prior to that point. The intuition is that each point represents that point in time reached after all occurrences which constitute it have taken place.

Events may have multiple occurrences. Two occurrences of the same event are the same if they have been preceded by the same sequences of events. Consider the sequences of events $aabb$ and $aaabab$. The second $b$ in the sequence $aabb$ is not the same occurrence as the second $b$ in $aaabab$. They take place in different "possible worlds". We may thus refer to events by giving the sequence of which they are the last occurrence.

These concepts comprise the basic behavioural model of our approach, namely that of behavioural presentations. A behavioural presentation, introduced in an article by Shields, is defined as follows.

**Definition 11**: A behavioural presentation is a quadruple $B = (O, \Pi, E, \lambda)$, where

1. $O$ is a set of occurrences
2. $\Pi \subseteq \wp(O)$ is a non-empty set of points
3. $E$ is a set of events
4. $\lambda : O \rightarrow E$ is the occurrence function

which satisfies $\bigcup_{\pi \in \Pi} \pi = O$.

The requirement that $\bigcup_{\pi \in \Pi} \pi = O$ says that every occurrence belongs to some point and essentially reflects the fact that we should not be concerned with things that could never happen. The function $\lambda$ associates occurrences with events. Therefore, $\lambda(o) = e$ is to be read as 'o is an occurrence of e'.

The behavioural presentation model is closely related to the event structures model. In fact, behavioural presentations mildly generalise event structures in allowing time ordering of events, given in Definition 12, to be a pre-order (a reflexive and transitive relation) rather than a partial order, thereby allowing the representation of simultaneity as well as concurrency. The relationship between the two is further examined in Shields’ book where it is shown that event structures correspond to behavioural presentations which are closed, in the sense of being prime algebraic and coherent. We omit further details.
In order to obtain a precise description of the behaviour of a component at its interfaces, we need to model: i) the order in which the component makes calls to operations to other components through its required interfaces, and ii) the order in which the component receives calls to operations from other components on its provided interfaces. A behavioural presentation can be rather useful to this end, since it determines various temporal relations on the set of occurrences of events.

**Definition 12:**

Let $B$ be a behavioural presentation and suppose that $o_1, o_2 \in O$. Define,

- $o_1 \not\preceq o_2 \iff \forall \pi \in \Pi : o_2 \in \pi \Rightarrow o_1 \not\in \pi$ and we say $o_1, o_2$ are mutually exclusive
- $o_1 \rightarrow o_2 \iff \forall \pi \in \Pi : o_2 \in \pi \Rightarrow o_1 \in \pi$ and we say $o_1$ has happened no later than $o_2$
- $o_1 \equiv o_2 \iff (o_1 \rightarrow o_2) \land (o_2 \rightarrow o_1)$ and we say $o_1, o_2$ occurred simultaneously
- $o_1 \circ o_2 \iff \neg(o_1 \not\preceq o_2) \land (o_1 \not\rightarrow o_2) \land (o_2 \not\rightarrow o_1)$ and we say $o_1, o_2$ occurred concurrently
- $o_1 < o_2 \iff (o_1 \rightarrow o_2) \land (o_2 \not\rightarrow o_1)$ and we say $o_1$ happened strictly before $o_2$

Using the above temporal relations we can determine the causal and temporal ordering amongst operation calls occurring at component interfaces, and in this way describe the observable behaviour of a component.

It can be seen that the temporal relations derived from behavioural presentations are based on two fundamental relations: $\not\preceq$ and $\rightarrow$. These relations introduce concepts of mutual exclusion and time ordering among events, in a fashion similar to the well-known conflict and causal temporal relations in an article by Küster-Filipe et al.\textsuperscript{10} and elsewhere. The relation $\rightarrow$ is a pre-order - that is, a transitive, reflexive relation. If $o_1 \rightarrow o_2$, then if $o_2$ has happened, then so must $o_1$. The relation $<\not\preceq$ is a strict pre-order. As for $\not\preceq$, if $o_1 \not\preceq o_2$, then an occurrence of $o_2$ excludes future occurrence of $o_1$, and vice versa. It is this relation that allows us to introduce notions of non-determinism into the model. In fact, $\not\preceq$ is an independence relation - that is, an irreflexive, symmetric relation. Put formally, $R$ is an independence relation on $X$ if and only if

$$\forall x, y \in X : (\neg xRy) \land (xRx \Longrightarrow yRx)$$

The following remark gives the basic properties of all temporal relations derived from behavioural presentations.
**Remark 13:** Suppose that $B$ is Behavioural Presentation, then

1. $\rightarrow$ is a pre-order
2. $\leq$ is an independence relation, and whenever $o_1 \rightarrow o_2$ and $d_1 \rightarrow d_2$, then $o_1 \leq d_1 \Longrightarrow o_2 \leq d_2$
3. $\equiv$ is an equivalence relation
4. $\circ$ is an irreflexive and symmetric relation
5. $<$ is a strict pre-order

A pre-ordered set is a pair $(X, \rightarrow)$ where $X$ is a set and $\rightarrow$ is a relation on the set which satisfies:

- $\forall x \in X : x \rightarrow x$ (reflexivity)
- $\forall x, y \in X : x \rightarrow y \land y \rightarrow z \Longrightarrow x \rightarrow z$ (transitivity)

Notice that we do not require $\rightarrow$ to be antisymmetric. Thus, if $x \rightarrow y$ and $y \rightarrow x$, instead of $x \equiv y$ we get $x \equiv y$ which means that $x$ and $y$ are distinct but stand in an equivalence relation. This allows the formal treatment of simultaneity on top of concurrency.

In further explanation of the notation, $\equiv$ is the equivalence relation generated by the pre-order $\rightarrow$. Hence, if $o, o_1, o_2 \in O$ and $o_1 \not\leq o_2$, then

- $o_1 \rightarrow o \iff o_2 \rightarrow o$
- $o \rightarrow o_1 \iff o \rightarrow o_2$
- $o \not\leq o_1 \iff o \not\leq o_2$

In other words, two occurrences related by $\equiv$ stand in exactly the same relationship to other occurrences. Further, suppose that $o_1 \equiv o_2$, and assume that we have some means of deriving from the system the exact time $t$ at which $o_1$ occurred. Then, if $o_1$ is in a relationship with the clock occurrence $t$, then $o_2$ must also be in that relation (and vice versa). Thus, $o_2$ must also have occurred at $t$. The interpretation is that $o_1, o_2$ are simultaneous. Notice that this is not the same as saying that $o_1, o_2$ are concurrent. $o_1 \circ o_2$ says that certainly neither precedes the other and they are not mutually exclusive.

Using the above temporal relations we may capture all occurrences in any Behavioural Presentation, as two occurrences are either:

1. mutually exclusive
2. ordered in time
3. simultaneous
4. concurrent
(5) strictly ordered in time.

and only one of these relations holds for a pair of occurrences.\footnote{Component-based systems are largely conceived of as proceeding in discrete steps. This implies that occurrences of events in the system do not blur into one another. In the spirit of event structures, as discussed by Winskel\cite{1}, this means that any two events in the system can be separated by an open neighbourhood. We would like to be able to describe such systems using our behavioural model. For this purpose, we shall consider a subclass of behavioural presentations which are well suited for representing discrete behaviour. Before defining discrete behavioural presentations we discuss related properties that motivate the definition.}

First, we want to ensure that discrete systems do proceed in an orderly way - in discrete steps. A step in behavioural presentation is understood in the following terms. Assume that the system is in a state where its occurrences of events so far are described by $\pi$. An occurrence $o$ of some event takes place and this additional occurrence is now described by $\pi'$. Thus, we obtain $\pi'$ by adding in $o$, to whatever occurrences were already in $\pi$. (In fact, we need to add in the entire simultaneity class of $o$ since, if $o'$ is some other occurrence such that $o \equiv o'$, then $o'$ must also be in $\pi'$ (by Definition 12 of $\equiv$ and $\rightarrow$).)

Second, we want to ensure that a behavioural presentation contains enough points to separate events which are strictly ordered or non-simultaneous. This is the repletion property and is defined as follows.

\textbf{Definition 14:} If $B = (O, \Pi, E, \lambda)$ is a behavioural presentation, then $B$ will be said to be replete iff whenever $\pi_1, \pi_2 \in \Pi$ such that $\pi_1 \subseteq \pi_2$ and $o_1, o_2 \in \pi_2 \setminus \pi_1$, then

$$o_2 \not\in o_1 \implies \exists \pi_3 \in \Pi : (\pi_1 \subseteq \pi_3 \subseteq \pi_2) \wedge (o_1 \in \pi_3) \wedge (o_2 \not\in \pi_3)$$

In further explanation, suppose that we have points $\pi_1$ and $\pi_2$ such that $\pi_1$ is before $\pi_2$, and $o_1, o_2$ occurred between $\pi_1$ and $\pi_2$. Now if $o_2$ occurred later than or concurrently with $o_1$ (i.e. $o_2 \not\in o_1$), then the repletion property says that there is a point (another possible world) $\pi_3$, after $\pi_1$ and before $\pi_2$, at which it is legitimate to assert that $o_1$ has happened but $o_2$ has not.

The idea of the repletion property perhaps can be best illustrated by the simple example given in Shields' book\footnote{In short, a coin is tossed \textit{occurrence} $c$ and then it either lands with heads on top (occurrence $h$) or tails on top (occurrence $t$). A behavioural presentation in this case would give $\pi_0 = \emptyset$, $\pi_1 = \{c, h\}$, $\pi_2 = \{c, t\}$ and that $c < h, c < t$ and $h \nleq t$. How-}. In short, a coin is tossed (occurrence $c$) and then it either lands with heads on top (occurrence $h$) or tails on top (occurrence $t$). A behavioural presentation in this case would give $\pi_0 = \emptyset$, $\pi_1 = \{c, h\}$, $\pi_2 = \{c, t\}$ and that $c < h, c < t$ and $h \nleq t$. How-
ever, there is a possible world missing. That is, π₃ in which the coin has been tossed but not landed yet. That would be π₃ = {c}. Therefore, π₃ is in between π₀ and π₁ (similarly, for π₀ and π₂) and c has happened but b has not (similarly, c has happened and f has not). It is situations like this that the repletion property is intended to capture. Referring back to Definition 6 of discrete components, repletion ensures that there are no 'gaps' in the time continuum.

Finally, to define discrete behavioural presentations we also need an ordering on subsets of behavioural presentations.

Definition 15: If \( B = (\Omega, \Pi, E, \lambda) \) is a behavioural presentation and \( X, Y \subseteq \Omega \), then we shall say that \( X \) is left-closed in \( Y \) whenever

\[
X \sqsubseteq Y \iff X \subseteq Y \land (\forall o \in X, \forall d' \in Y : d' \rightarrow o \implies d' \in X)
\]

This ordering relation among subsets of behavioural presentations is reminiscent of the left-closed subsets in the prime event structures model of Winskel.\(^{31}\)

Remark 16: The relation \( \sqsubseteq \) is a partial order on \( \wp(\Omega) \).

Proof: Reflexivity and antisymmetry of \( \sqsubseteq \) follow from reflexivity and antisymmetry of \( \subseteq \). Now suppose that \( X, Y, Z \subseteq \Omega \) with \( X \sqsubseteq Y \) and \( Y \sqsubseteq Z \). Then, we may deduce that \( X \subseteq Z \). Let \( o \in X \) and \( d' \in Z \) such that \( d' \rightarrow o \). It suffices to show that \( d' \in X \). We have that \( o \in Y \) as \( X \subseteq Y \). Since \( Y \subseteq Z \), we may deduce that \( d' \in Y \). So, we have \( o \in X \) and \( d' \in Y \) with \( d' \rightarrow o \). Since \( X \subseteq Y \), we can conclude that \( d' \in X \). Hence, we have shown that \( \sqsubseteq \) is also transitive, which completes the proof.

Now we can define discrete behavioural presentations.

Definition 17: A behavioural presentation \( B = (\Omega, \Pi, E, \lambda) \) will be said to be discrete if and only if, for every \( \pi \in \Pi \) we have,

1. The set of \( \equiv \)-classes of the elements of \( \pi \) is finite
2. If \( X \sqsubseteq \pi \), then \( X \in \Pi \)

Point (1) of the above definition asserts that only a finite number of occurrences may take place within finite time. Hence, it is a finitary condition and with regard to Definition 6 of discrete component languages, it excludes infinite ascending and descending chains of occurrences of events. By examining Definition 14 it can be seen that a non-replete behavioural
presentation may have some points missing. Definition 15 says that such points must lie under existing points. Point (2) of the above definition then, guarantees inclusion of those points and thus ensures that there will be no points missing. With regard to Definition 6 of discrete components, this ensures that there are no 'gaps' in the time continuum.

Note that point (2) essentially reflects Definition 15 so that a discrete behavioural presentation is one that is left-closed and additionally satisfies the finitary condition (point (1) of Definition 17). Finally, note that since $\emptyset \subseteq \pi$, for all $\pi \in \Pi$ it follows, again from point (2) of Definition 17, that discrete behavioural presentations have bottom elements. This is the initial point in which nothing has happened yet, in the sense of discrete component languages (Definition 6).

It might be worth adding a small note to clarify the terminology used so far. Discreteness and local left-closure in component languages are defined as two separate properties. Discreteness in behavioural presentations includes the notion of a left-closed behavioural presentation. Hence, a discrete component language is not necessarily locally left-closed whereas a discrete behavioural presentation is always left-closed. The obvious connotations of the naming are intentional. Well-behavedness (i.e. discreteness and local left-closure) of component languages manifests itself in discrete behavioural presentations, as we will see in the sequel.

Now we turn our attention to relating the (language part of the) component model, described in Section 2, to a behavioural presentation. More specifically, we describe a construction that takes a well-behaved component into a discrete behavioural presentation. The construction uses a familiar idea of taking the elements of the orders to be primes in the partial order of component vectors $(V, \leq)$.

Recall that an element $x$ of a partially ordered set $(X, \leq)$ is prime if, whenever $U \subseteq X$ and $x \leq \bigcup U \in X$, then $x \leq u$, some $u \in U$. The set of all primes of $(X, \leq)$ will be denoted by $Pr(X)$.

In mapping a well-behaved (i.e. discrete and locally left-closed) component language onto a discrete behavioural presentation the main challenge lies with left-closure of the corresponding behavioural presentation. The finitary condition (point (1) of Definition 17) that makes a left-closed behavioural presentation discrete, can be guaranteed by discreteness of the component language (see Definition 6).

In the study of left-closed behavioural presentations in Shield's book\cite{26}, which provides useful insights on the subject matter, it is shown that if $B = (O, \Pi, E, \lambda)$ is a left-closed behavioural presentation, then the poset
(II, \leq) is prime algebraic and consistently complete, with the elements of \downarrow o as primes, where \downarrow o = \{d \in D : d \rightarrow o\}. Subsequent analysis in the reports\textsuperscript{27,28} showed that in order to associate components with behavioural presentations we need to characterise primes in a component language and prove prime algebraicity and consistent completeness. But, let us first describe a construction that maps a component language onto a quadruple (Oc, \Pi c, Ec, \lambda c) which mirrors the behavioural presentation model.

We start by exploiting the basic properties of the associated order-theoretic structures. Recall that the relation \alpha (Definition 9) provides an ordering among vectors in which one is immediately beneath another allowing no other vector in V to exist in between them. For \x{\mathbf{u}} \in V, we define
\[ \text{pre}_V(\mathbf{u}) = \{ \mathbf{u} \in V : \mathbf{u} \alpha \mathbf{v} \} \]
This set contains all vectors that are related to \mathbf{u} by \alpha. Our interest lies with those vectors which have a unique other vector, also in V, immediately beneath them.

**Definition 18:** Let c = (\Sigma, V) be a well-behaved component, then we shall say that \x{\mathbf{z}} \in V is **primal** if there exists exactly one \x{\mathbf{u}} \in V such that \x{\mathbf{u}} \alpha \x{\mathbf{z}}.

Primal vectors turn out (cf Proposition 21) to be the set of complete primes in (V, \leq). In this context, the notion of prime refers to vectors which have a unique other vector immediately below them. There is an analogy to be made with prime numbers in number theory. Essentially, primal vectors allow us to identify significant events and how occurrence of an event causally depends on the previous occurrence of others. Such events are the ones appearing in primal vectors in a component language. The fact that such vectors have a unique other vector immediately beneath them, implies that the behaviour they describe can be obtained in a unique way from the behaviour of the vector sitting directly underneath them via the occurrence of a single event (or its simultaneity class, in case of simultaneous events on different interfaces).

We shall write prms(V) for the set of all primal vectors in V. Put formally,
\[ \text{prms}(V) = \{ \mathbf{u} \in V : |\text{pre}_V(\mathbf{u})| = 1 \} \]
where |X| denotes the cardinality of the set X. If \x{\mathbf{u}} \in \text{prms}(V) then we define base\textsubscript{V}(\x{\mathbf{u}}) to be the unique element of pre\textsubscript{V}(\x{\mathbf{u}}) and we further define
\[ \text{ifs}_V(\mathbf{u}) = \{ i \in I_\Sigma : \text{base}_V(\mathbf{u})(i) < \mathbf{u}(i) \} \]
Each primal vector \( \underline{u} \in \text{prms}(V) \) represents behaviour in which an operation call (and any others simultaneous to it) has occurred on the corresponding interface(s), during the course of behaviour since that described by \( \text{base}_V(\underline{u}) \). The key point is that at most one operation call has occurred per interface during the fragment of behaviour between \( \text{base}_V(\underline{u}) \) and \( \underline{u} \).

We accordingly associate primals in \( V \) with simultaneity classes of event occurrences, as we define next.

**Definition 19:** Suppose that \( c = (\Sigma, V) \) is a well-behaved component and let

\[
O_c = \{(\underline{u}, i) \in V \times I_S : \underline{u} \in \text{prms}(V) \land i \in \text{i}fs_V(\underline{u})\}
\]

We define a function \( \lambda_c : O_c \to Op \times I_S \) by

\[
\lambda_c(\underline{u}, i) = (m, i) \iff \exists y \in \beta_S(i)^* \text{ and } m \in \beta_S(i) \text{ such that } \underline{u}(i) = y \cdot m
\]

The set \( O_c \) comprises all possible occurrences of events in the behaviour of the component. As for the occurrence function, if \( \lambda_c(o) = (m, i) \) then \( o \) is the occurrence of an event consisting of a call to operation \( m \) at interface \( i \), during behaviour described by \( \underline{u} \) and since that already described by \( \text{base}_V(\underline{u}) \). Notice that since \( \underline{u}(i) \neq \lambda \) when \( (\underline{u}, i) \in O_c \), there exists \( y \in \beta_S(i)^* \) such that \( \underline{u}(i) = y \cdot m \). In effect, we isolate the last call out of the sequence of calls to operations at interface \( i \). This is indeed the case since \( \underline{u} \) is a primal vector and thus the sequence \( y \) of events (operation calls) on its \( i \)-th coordinate will have been already described by the corresponding \( \text{base}_V(\underline{u})(i) \). In other words, we isolate \( m \) which is what takes \( \text{base}_V(\underline{u}) \) and 'stretches it up' to \( \underline{u} \).

Next, we also need to define a set of points.

**Definition 20:** For \( \underline{u} \in V \), we define

\[
\pi_\underline{u} = \{(\underline{u}, i) \in O_c : \underline{u} \leq \underline{u}\}
\]

The set \( \pi_\underline{u} \) is the set of all occurrences of events during the component behaviour represented by \( \underline{u} \). The set of all sets \( \pi_\underline{u} \) for \( \underline{u} \in V \) constitutes the set of points \( \Pi_c \), hence

\[
\Pi_c = \{\pi_\underline{u} : \underline{u} \in V\}
\]
The structure \((O_c, \Pi_c, E_c, \lambda_c)\) is an instance of a behavioural presentation\(^{20}\), where \(O_c\) is the set of occurrences of events in the behaviour of the component \(c = (\Sigma, V)\), \(\Pi_c \subseteq \psi(O_c)\) is a set of points, \(E_c\) is the set of events of the component and \(\lambda_c\) is the occurrence function. Finally, \(\bigcup_{\pi \in \Pi_c} \pi = O_c\).

We may now proceed to characterise primes in a well-behaved component language (in fact, discreteness suffices) as its primal vectors. This is established in the following proposition.

**Proposition 21:** Let \(c = (\Sigma, V)\) be a well-behaved component and \(\vec{z} \in V\) with \(\vec{z} \neq \vec{1}\). Then, the following are equivalent:

1. \(\vec{z}\) is prime
2. \(\vec{z}\) is primal

**Proof:** See Proposition 4.2 in the report by Shields and Moschyiannis\(^{27}\), which also contains a series of useful intermediate results.

Our remaining concerns have to do with proving prime algebraicity and consistent completeness of a well-behaved component language. In fact, consistent completeness follows from discreteness of the component language - it is inherent in the definition of discreteness as discussed before (see Definition 6).

As for prime algebraicity, which is somewhat more tricky, we will need to invoke local left-closure as this will allow the construct primes 'to order'. More specifically, given a non-prime \(\underline{u}\) we may 'pull it down' to an element \(\underline{u}'\) of \(V\) such that \(\underline{u}' < \underline{u}\) by keeping some chosen coordinate fixed until we hit a prime. This entails that \(Pr(\underline{u})\), defined by \(Pr(\underline{u}) = \{\underline{z} \in Pr(V) : \underline{z} \leq \underline{u}\}\), possesses an element \(\underline{u}_i\) with \(u'_i(i) = u(i)\), each \(i\). We may thus argue that \(\bigcup Pr(\underline{u}) i \geq \underline{u}(i)\), all \(i\), which is the hard direction in proving prime algebraicity of a component language.

**Proposition 22:** Let \(c = (\Sigma, V)\) be a well-behaved component, then \(V\) is prime algebraic with the primal vectors as primes.

**Proof:** Let \(\underline{u} \in V\) and define \(Pr(\underline{u})\) as above. We show that \(\bigcup Pr(\underline{u}) = \underline{u}\) by proving that \(\bigcup Pr(\underline{u}) \leq \underline{u}\) and then the reverse inequality. See Proposition 4.1 in the report\(^{27}\) for the complete proof.

The local left-closure property of components (Definition 7) takes up on ideas of left-closed subsets of the set of occurrences in a behavioural
presentation. Consider the component \( c = (\Sigma, V) \), where \( I_c = \{1, 2\} \) with the set of behaviours \( V = \{(A, A), (aa, A), (A, bb), (aa, bb)\} \). It can be shown that the component language \( V \) constitutes a finite lattice, so \( c \) is discrete. However, the corresponding behavioural presentation would have the counterintuitive property that although four operation calls have occurred there are only two elements in \( O_c \) to describe them, namely \((aa, A)\) and \((A, bb)\). This is because the two primal vectors represent the occurrence of the second of the operation calls on each interface. It is for this reason that we require local left-closure of the component language \( V \) of a component. In fact local comes from the fact that it is applied to each interface of the component (i.e. to each coordinate of a component vector in \( V \)).

**Example 23:**

We extend Examples 4 and 10 with a sequence of demonstrations that will allow us to use a behavioural presentation to model the potential behaviour of the CMenu component.

First, for each element \( \xi \) in \( V \) of Example 4 we determine its corresponding set \( \text{pre}_V(\xi) \). Recall that this is the set of component vectors that lie immediately underneath \( \xi \).

\[
\begin{align*}
\text{pre}_V(A, A, A) &= \emptyset \\
\text{pre}_V(a_1, A, A) &= \{(A, A, A)\} \\
\text{pre}_V(A, b_1, A) &= \{(A, A, A)\} \\
\text{pre}_V(a_1, a_2, A, A) &= \{(a_1, A, A)\} \\
\text{pre}_V(a_1, b_1, b_2, A) &= \{(a_1, b_1, A)\} \\
\text{pre}_V(a_1, a_2, b_1, A) &= \{(a_1, a_2, A, A)\} \\
\text{pre}_V(a_1, a_2, b_1, c_1) &= \{(a_1, a_2, A, A)\} \\
\text{pre}_V(a_1, b_1, b_2, A) &= \{(a_1, b_1, b_2, A)\} \\
\text{pre}_V(a_1, b_1, b_2, b_3, A) &= \{(a_1, b_1, b_2, b_3, A)\}
\end{align*}
\]

All \( \xi \) that have a unique vector in \( V \) immediately below them comprise the set \( \text{prms}(V) \).

\[
\text{prms}(V) = \{ (a_1, A, A), (A, b_1, A), (a_1, a_2, A, A), (A, b_1, b_2, A), (A, b_1, b_2, A), (a_1, a_2, A, c_1, (a_1, b_1, b_2, b_3, A) \}
\]

Next, we associate the behaviour vectors in \( \text{prms}(V) \) with the interfaces to which the last call to an operation occurred. In effect, we are applying the definition of \( O_c \) to each primal element in the component language of the CMenu component. The following occurrences make up the set \( O_c \) for the component.
\[ \sigma_1 = ((a_1, A, A), ISearchFree) \]
\[ \sigma_2 = ((A, b_1, A), IFineTune) \]
\[ \sigma_3 = ((a_1, a_2, A, A), ISearchFree) \]
\[ \sigma_4 = ((A, b_1 b_2, A), IFineTune) \]
\[ \sigma_5 = (A, b_1 b_2, A), IFineTune \]
\[ \sigma_6 = ((a_1, a_2, A, c_1), IDetectSignal) \]
\[ \sigma_7 = (A, b_1 b_2, A), IFineTune \]

The occurrence function \( \lambda_c \) can also be used to determine exactly to which call to operation, and on which interface, each occurrence refers to.

\[ \lambda_c(\sigma_1) = (a_1, ISearchFree) \]
\[ \lambda_c(\sigma_2) = (b_1, IFineTune) \]
\[ \lambda_c(\sigma_3) = (a_2, ISearchFree) \]
\[ \lambda_c(\sigma_4) = (b_2, IFineTune) \]
\[ \lambda_c(\sigma_5) = (b_3, IFineTune) \]
\[ \lambda_c(\sigma_6) = (c_1, IDetectSignal) \]
\[ \lambda_c(\sigma_7) = (b_3, IFineTune) \]

The occurrence function is to be read in conjunction with the corresponding occurrence given earlier. For instance, \( \lambda_c(\sigma_7) = (b_3, IFineTune) \) implies that \( \sigma_7 \) is the occurrence of call to operation \( b_3 \) on interface \( IFineTune \). This is also the case for \( \lambda_c(\sigma_5) \) (with \( \lambda_c(\sigma_5) = (b_3, IFineTune) \)).

However, \( \lambda_c(\sigma_5) \) is associated with \( \sigma_5 = ((A, b_1 b_2, A), IFineTune) \) which means that \( \sigma_5 \) refers to occurrence of \( b_3 \) when \( b_1 \) has happened earlier whereas \( \lambda_c(\sigma_5) = (b_3, IFineTune) \) is associated with \( \sigma_5 = ((a_1, b_1 b_2 b_3, A), IFineTune) \) and refers to occurrence of \( b_3 \) on \( IFineTune \) but only after the component has experienced calls to operations \( a_1, b_1 \) and \( b_2 \).

Based on Definition 20, we may now obtain the sets of all occurrences of calls to operations during the course of behaviour in question.

\[ \tau((A, A, A)) = \emptyset \]
\[ \tau((a_1, A, A)) = \{ \sigma_1 \} \]
\[ \tau((b_1, A, A)) = \{ \sigma_2 \} \]
\[ \tau((a_1 a_2, A, A)) = \{ \sigma_1, \sigma_3 \} \]
\[ \tau((A, b_1 b_2, A)) = \{ \sigma_4 \} \]
\[ \tau((a_1, b_1, A)) = \{ \sigma_1, \sigma_2 \} \]
\[ \tau((a_1, a_2, A, c_1)) = \{ \sigma_1, \sigma_3, \sigma_6 \} \]
\[ \tau((a_1, b_1, b_2, A)) = \{ \sigma_1, \sigma_2, \sigma_4 \} \]
\[ \tau((a_1, b_1 b_2 b_3, A)) = \{ \sigma_1, \sigma_2, \sigma_4, \sigma_7 \} \]
A Formal Approach to Constructing Well-Behaved Systems Using Components

\[ \pi(a_1, a_2, b, c_1, c_2) = \{a_1, a_2, a_3, a_6\} \]

For instance, \(\pi(a_1, a_2, A, c_1) = \{a_1, a_2, a_6\}\) contains occurrence of event \(a_1\) at interface ISearchFre (i.e. \(a_1\)), occurrence of event \(a_2\) at interface ISearchFre (i.e. \(a_2\)) and occurrence of event \(c_1\) at interface IDetectSignal (i.e. \(a_6\)). Occurrence of all three events comprises the behaviour of the component represented by \((a_1, a_2, A, c_1)\).

Referring back to Definition 11, we have constructed the sets \(O_c\) and \(P_c\) for the CMenu component. Finally, from the set of all occurrences of events given above and based on Definition 12 we extract the temporal relations among occurrences of events for the CMenu component shown in Table 1.

<table>
<thead>
<tr>
<th>for (a_1)</th>
<th>for (a_2)</th>
<th>for (a_3)</th>
<th>for (a_4)</th>
<th>for (a_5)</th>
<th>for (a_6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a_1 &lt; a_2)</td>
<td>(a_2 \leq a_3)</td>
<td>(a_3 \leq a_4)</td>
<td>(a_4 \leq a_5)</td>
<td>(a_5 \leq a_6)</td>
<td>(a_6 \leq a_7)</td>
</tr>
<tr>
<td>(a_1 \leq a_2)</td>
<td>(a_2 &lt; a_3)</td>
<td>(a_3 \leq a_6)</td>
<td>(a_4 \leq a_6)</td>
<td>(a_5 \leq a_6)</td>
<td>(a_6 \leq a_7)</td>
</tr>
<tr>
<td>(a_1 &lt; a_6)</td>
<td>(a_2 &lt; a_5)</td>
<td>(a_3 &lt; a_6)</td>
<td>(a_4 &lt; a_7)</td>
<td>(a_5 \leq a_7)</td>
<td>(a_6 \leq a_7)</td>
</tr>
<tr>
<td>(a_1 \leq a_7)</td>
<td>(a_2 &lt; a_7)</td>
<td>(a_3 \leq a_7)</td>
<td>(a_4 \leq a_7)</td>
<td>(a_5 \leq a_7)</td>
<td>(a_6 \leq a_7)</td>
</tr>
</tbody>
</table>

As an illustration of how these temporal relations are obtained we examine the relation between \(a_2\) and \(a_6\). Occurrence \(a_6\) refers to a call to operation \(c_1\) at interface IDetectSignal, but only when calls to operations \(a_1\) and then \(a_2\) at ISearchFre have preceded it. Occurrence of a call to operation \(a_1\) followed by \(a_2\) at the same interface is precisely occurrence \(a_2\). Thus, \(a_3\) strictly precedes \(a_6\) since \(a_1\) and \(a_2\) must have occurred at interface ISearchFre (this is \(a_2\)) before \(c_1\) can occur at interface IDetectSignal (this is \(a_6\)).

![Fig. 6. Behavioural presentation model for the CMenu component](image-url)
A behavioural model of the CMenu component can be seen in Figure 6 which depicts the temporal relations among occurrences of events, as these are experienced by the component on its interfaces.

4. Composition of Components

Up to this point we have been concerned with a single software component. We described a component model and identified conditions on component behaviours which enabled the link to a behavioural model, namely that of behavioural presentations. As a result, component behaviour can be modelled using behavioural presentations.

The major challenge in a component setting is that of understanding the consequences of interconnecting components. In this section, we give an overview of the notion of composition of components within our framework. We shall not discuss composition in great detail though as it is not essential for the understanding of the current paper. A more rigorous treatment of composition can be found in the article by Moschoyiannis and Shields while a previous paper outlines the key technical results.

The main purpose in this section is to show that the mathematical framework we have been concerned with so far is indeed compositional. In particular, we consider an operation on components which takes a set of components and combines them by allowing communication between their provided and required interfaces.

The basic concept behind composition, as this is considered in our framework, is the following. If component \( c_1 \) provides interface \( i \) and component \( c_2 \) requires interface \( i \), then a behaviour of \( c_1 \) and a behaviour of \( c_2 \) can only be composed if their restrictions to interface \( i \) are the same. From the composition of those behaviours, the sequence of events that correspond to interface \( i \) is removed.

In this form of composition, components are combined by connecting their complementary interfaces; that is, interfaces provided by one component and required by another. This reminds of the notion of complementary labels in Milner’s CCS. It has as an implication that the two components must have disjoint sets of both provides and requires interfaces. They may have complementary interfaces in common. The function \( \beta_z \); then, on those interfaces, must return the same set of operations on both components. In mathematical terms, the components must be consistent. This is formally put as follows.

**Definition 24:** Suppose that \( c_1 = (\Sigma_1, V_1) \) and \( c_2 = (\Sigma_2, V_2) \) are compo-
ments. We say that their sorts $\Sigma_1$ and $\Sigma_2$ are consistent and we write $\Sigma_1 \downarrow \Sigma_2$ if and only if

- $P_{\Sigma_1} \cap P_{\Sigma_2} = \emptyset$
- $R_{\Sigma_1} \cap R_{\Sigma_2} = \emptyset$
- for each $i \in I_{\Sigma_1} \cap I_{\Sigma_2}$ we have $\beta_{\Sigma_1}(i) = \beta_{\Sigma_2}(i)$

We also define $c_1, c_2$ to be consistent, and we write $c_1 \downarrow c_2$, if $\Sigma_1$ and $\Sigma_2$ are consistent.

In a fashion similar to the formalisation of a single software component, we are again interested in the static structure and the dynamic characteristics, this time for the composite component. Before we compose components themselves, we compose their sorts (static structure) and their component languages (dynamic characteristics).

The sort of the composite is formed from those of the individual components by eliminating all interfaces participating in internal communication. This is formally put in the following definition.

**Definition 25:**
Suppose that $\Sigma_1$ and $\Sigma_2$ are consistent sorts. Define $\Sigma_1 \oplus \Sigma_2 = \Sigma$ where,

- $P_\Sigma = (P_{\Sigma_1} \cup P_{\Sigma_2}) \setminus (R_{\Sigma_1} \cup R_{\Sigma_2})$
- $R_\Sigma = (R_{\Sigma_1} \cup R_{\Sigma_2}) \setminus (P_{\Sigma_1} \cup P_{\Sigma_2})$
- $\beta_\Sigma(i) = \beta_{\Sigma_j}(i)$ wherever $i \in I_{\Sigma_j}, j = 1, 2$

The following lemma establishes that $\Sigma_1 \oplus \Sigma_2$ is indeed a sort whenever $\Sigma_1$ and $\Sigma_2$ are consistent sorts.

**Lemma 26:** Suppose that $\Sigma_1$, $\Sigma_2$ are consistent sorts, then $\Sigma_1 \oplus \Sigma_2$ is a sort.

**Proof:** Let $\Sigma_1 \oplus \Sigma_2 = \Sigma$. We first prove that $\beta_\Sigma$ 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 $\beta_{\Sigma_1}(i) = \beta_{\Sigma_2}(i)$. But this is precisely point (3) of Definition 24. Finally, we note that,

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

which completes the proof. $\Box$
As for the behaviour of the resulting composite, it is an aggregate of behaviours from each of the components at the non-connected - or better, the non-complementary - interfaces which remain essentially unaffected, and of its connected interfaces on which the sequences of events must agree.

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 $\mathcal{u}_j$, if $i$ is a complementary interface of components $c_j$ and $c_k$, then it will be a provided interface of one and a required interface of the other. Without loss of generality, suppose it is an interface provided by $c_j$ and required by $c_k$. Then, $\mathcal{u}_j(i)$ represents the sequence of operations sent from $c_j$ to $c_k$, which (assuming no delays) is precisely $\mathcal{u}_k(i)$.

In the following definition, $I_{\Sigma_1} \Delta I_{\Sigma_2}$ is the symmetric difference of $I_{\Sigma_1}$ and $I_{\Sigma_2}$, defined to be $(I_{\Sigma_1} \setminus I_{\Sigma_2}) \cup (I_{\Sigma_2} \setminus I_{\Sigma_1})$.

**Definition 27:** Suppose that $c_1 = (\Sigma_1, V_1)$ and $c_2 = (\Sigma_2, V_2)$ are components and let $I_{\Sigma_1}$, $I_{\Sigma_2}$ be their sets of interfaces, respectively. We shall say that vectors $\mathcal{u}_1 \in V_1$ and $\mathcal{u}_2 \in V_2$ are consistent, and we write $\mathcal{u}_1 \downarrow \mathcal{u}_2$ if

$$\mathcal{u}_1|_{I_{\Sigma_1} \cap I_{\Sigma_2}} = \mathcal{u}_2|_{I_{\Sigma_1} \cap I_{\Sigma_2}}$$

where $f|_X$ denotes the restriction of function $f$ to the set $X$, in which case we define,

$$\mathcal{u}_1 \oplus \mathcal{u}_2 = (\mathcal{u}_1 \cup \mathcal{u}_2)|_{I_{\Sigma_1} \Delta I_{\Sigma_2}}$$

where $\mathcal{u}_1 \cup \mathcal{u}_2 : I_{\Sigma_1} \Delta I_{\Sigma_2}$ satisfies

$$(\mathcal{u}_1 \cup \mathcal{u}_2)(i) = \begin{cases} \mathcal{u}_1(i) & , i \in I_{\Sigma_1} \\ \mathcal{u}_2(i) & , i \in I_{\Sigma_2} \end{cases}$$

which is well defined if $\mathcal{u}_1 \downarrow \mathcal{u}_2$.

This defines the operation $\oplus$ of composition on component languages. Two component vectors can be composed following Definition 27 providing that they are consistent. Consistency basically ensures that the vectors from each language agree on complementary interfaces. There is no point in considering vectors whose sequences of events on shared interfaces do not agree. In the resulting composite vector $\mathcal{u}_1 \oplus \mathcal{u}_2$, the coordinates that correspond to complementary interfaces are removed - notice the restriction to $I_{\Sigma_1} \Delta I_{\Sigma_2}$ in the definition of $\mathcal{u}_1 \oplus \mathcal{u}_2$. The remaining coordinates are essentially unaffected by composition and the components exhibit the same behaviour as that prior to composition, on the corresponding interfaces.
Having composed their sorts (following Definition 25) and their component vectors (following Definition 27), we can now give the formal definition of composition of components.

**Definition 28:** Suppose that \( c_1 = (\Sigma_1, V_1) \) and \( c_2 = (\Sigma_2, V_2) \) are components. Then, we define their composition \( c_1 \oplus c_2 = (\Sigma, V) \) where,

- \( \Sigma = \Sigma_1 \oplus \Sigma_2 \)
- \( V = V_1 \oplus V_2 \) where,

\[
V_1 \oplus V_2 = \{ \mathbf{v} \in V_Y \mid \exists \mathbf{u}_1 \in V_1, \exists \mathbf{u}_2 \in V_2 : \mathbf{u}_1 \downarrow \mathbf{u}_2 \land \mathbf{v} = \mathbf{u}_1 \oplus \mathbf{u}_2 \}
\]

Informally, the definition says that the static structure of the composite is formed from those of the components while the dynamics reflect the fact that behaviours of the composite comprise component vectors from each of the components on non-connected interfaces so long as they are consistent on the shared interfaces.

Based on the above definitions, it can be shown that \( c_1 \oplus c_2 \) is a component whenever \( c_1, c_2 \) are consistent components. This is formally put in the following lemma.

**Lemma 29:** Suppose that \( c_1, c_2 \) are consistent components, then \( c = c_1 \oplus c_2 \) is a component.

**Proof:** Let \( c = c_1 \oplus c_2 = (\Sigma, V) \) hence, \( \Sigma = \Sigma_1 \oplus \Sigma_2 \) and \( V = V_1 \oplus V_2 \). Then, \( \Sigma \) is a sort by Lemma 26 and \( V \subseteq V_Y \) by definition.

The operation \( \oplus \) of composition of components is commutative - notice that Definitions 24, 25, 27, 28 are all symmetric on \( \Sigma_1, \Sigma_2 \) or \( c_1, c_2 \) - and associative.\(^{20}\) This has the advantage of being able to build systems from generic components. We can take two components, put them together and the resulting composite can then be further composed with another component or another composite.

The work presented in the related article\(^{20}\) also deals with the all important issue of preservation of well-behavedness (i.e. discreteness and local left-closure properties) under the operation of composition. It can be shown that under certain conditions, captured by the notion of compatibility between components,\(^{20}\) the composition of well-behaved components results in a well-behaved composite component.

Local left-closure of the resulting composite is relatively straightforward. Discreteness requires further analysis of the interaction between \( \sqcup \) and \( \downarrow \) in the corresponding component languages. This is useful in showing that
vectors bounded above in the composite component language have their least upper bound and greatest lower bound in it.

The fact that well-behavedness is preserved under composition of components allows us to infer prime algebraicity of the composite component language (based on Proposition 22). In this way, the resulting well-behaved composite component can be readily associated with a discrete behavioural presentation. Therefore, its behaviour can be described again using the powerful model of true-concurrency presented in Section 3. The interested reader is referred to the paper\textsuperscript{19} or the article\textsuperscript{20} for further details of the results establishing the algebraic properties of the operation \(\oplus\) of composition as well as the issue of preservation of well-behavedness under composition.

5. Conclusions and Future Work

In this paper we presented a mathematical model for formalising software components based on the use of tuples of sequences of events to model component behaviour. These sets of tuples can be used to model the behaviour of components provided they are well-behaved, that is, they satisfy two conditions, namely discreteness and local left-closure. In that case, our component model corresponds to an order-theoretic based model of behaviours with desirable properties. In this paper, the presentation of the mathematical framework has been mostly concerned with the behaviour of a single component. The fact that the framework is compositional was briefly discussed. Component composition is dealt with in the paper\textsuperscript{19} and further in the article by Moschoyiannis and Shields\textsuperscript{20} which contains an intensive study of the algebraic properties of the operation of composition within the framework.

The overall objective of the work presented in this paper is the application of mathematical methods in enhancing the design of component-based systems. Our approach relates to both building a component (how to specify a component taking into account structural and behavioural aspects) and reuse (how can the component be reused and combined with others, how can it be operated, what are the expected sequences of operations). In creating a component, a designer would be expected to specify the desired set of behaviours. Using our model, and in checking against discreteness and local left-closure, we identify possible missing behaviours and refine the set of behaviours for the component (i.e. additional vectors might be included in the set to make the component well-behaved; such vectors might indicate pathological behaviour, leading to refinement of the initial design). The re-
A Formal Approach to Constructing Well-Behaved Systems Using Components

fixed set of behaviours then describes what the component does as well as the ordering of calls to operations, which needs to be respected whenever the component is deployed in different configurations.

Probably the closest to our model is the algebraic specification model of Broy.\textsuperscript{2, 3} The use of tuples of sequences to model component behaviour is reminiscent of the use of streams in Broy’s approach to represent messages communicated along the channels of a component. Essentially, the set out of our model is quite similar to that of Broy,\textsuperscript{2, 3} where a software component is modelled as an input/output function transferring input streams to the set of possible output streams. Semantically, a component in Broy’s model is represented by a predicate defining a set of behaviours where each behaviour is represented by a stream processing function.\textsuperscript{2} In this respect, we take a different approach since our model is based on the order theoretic structure of the set of behaviours of a component and is then related to behavioural presentations which provide a denotational semantics expressive enough to capture non-determinism, concurrency and simultaneity.

Another approach to formalising software components is that of Küster-Filipe\textsuperscript{9, 11} which describes a distributed logical framework for components and their composition. The initial set out is quite different to our model since Küster-Filipe introduces a distributed temporal logic MDTL for specifying components. The semantics of MDTL is based on event structures which, similarly to behavioural presentations, can capture non-determinism and concurrency, but by contrast cannot express simultaneity.\textsuperscript{26}

At the design level a software engineer could envisage using UML\textsuperscript{23} to specify component-based systems. Even though UML was not developed for component-based design some of its notation can be useful (as shown for example by Cheesman and Daniels\textsuperscript{9}). In particular, UML includes a constraint language called Object Constraint Language (OCL) which can be used for describing component contracts, mostly in terms of pre- and postconditions on interface operations.\textsuperscript{10} OCL 1.x is still essentially a static language and lacks the appropriate expressiveness to capture provides/requires dependencies, which are essential for describing component contracts precisely.\textsuperscript{20} This is tackled in work by Küster-Filipe\textsuperscript{11} and Küster-Filipe et al.\textsuperscript{10} using a Catalysis-like notation\textsuperscript{7} for describing component interactions and component frameworks, respectively. Work in increasing the expressive power of OCL is in progress and possible correspondence to the temporal relations derived from behavioural presentations needs to be further investigated. In any case, OCL 2.0 will have added expressive power and consequently provide more useful notation for describing some component contracts.
Finally, one natural extension of our work is to associate component behaviour with automata. This transition is straightforward because behavioural presentations give rise to a certain class of automata, building on consequences of discreteness and local left-closure. Preliminary work suggests that we may take vectors appearing in a component language as states and define transitions in a way that reflects the observation that behaviours may be seen to be built up from the empty vector by repeatedly concatenating 'event vectors' to it. These are vectors in which each coordinate is either a single event or the empty sequence. An obvious advantage of automata is that it would make our formal approach to components amenable to automated verification and consequently tool support. Such considerations are however subject for further work.

References


30. R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala Component Model for Consumer Electronics. *IEEE Transactions on Com-