Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 141 (2005) 199–220
www.elsevier.com/locate/entcs

Modelling Component Behaviour with Concurrent Automata

Sotiris Moschoyiannis, Michael W. Shields and Paul J. Krause

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

Abstract

The effective (re)use of components requires languages for the precise description of observable behaviour, along with methods for checking the compatibility of component interfaces in a design. This is even more challenging in the presence of concurrency. In previous work we have considered a set-based model of components and their composition, in a concurrent setting. In this paper, we present a class of automata, called Σ-*automata*, in which true-concurrency is treated as an explicit structural property. We show how an automaton can be derived from a component and that every such automaton generates back a component. Apart from determining a usage protocol for the underlying component, this extension to our model provides useful insights on component composition.

Keywords: component behaviour, concurrency, automata

1 Introduction

Complex high-integrity software systems, such as those described in [9], are typically divided into interconnected components. However, experience has shown (e.g. in the consumer electronics industry [28]) that the resulting software may not function as required because of subtle inconsistencies that arise during the complex call interplay between component interfaces. For example, one component may require certain signals to arrive consecutively while the other is generating them concurrently. As a result, systems may exhibit

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

doi:10.1016/j.entcs.2005.04.035

pathological behaviour. That is, behaviour not intended but resulting from such inconsistencies.

We study discrete models of interactive systems and, naturally, our interest lies with the *observable* behaviour of components rather than their internal structure. In this context, it proves crucial to describe behaviours as experienced on interfaces of the component. We present an automata-based language for describing such behaviour of a component. This allows to capture assumptions about the order in which the operations of a component are called, and the order in which the component (in response) calls external operations. In a certain important sense, we are inherently also modelling the environment of a component and when it comes to composition, two components can be put together if there is some environment that satisfies all assumptions of both. This is in line with the *optimistic* view prescribed in [6].

While using an automata-based formalism for modelling behaviour is not new, the proposed automata are interesting in that they make it possible to express concurrency explicitly. It may be argued that there are phenomena in a component setting such as race conditions which may only be understood as an unfortunate interaction between concurrency and nondeterminism. Such phenomena are of interest in areas such as communication and consumer electonics products. It may also be argued that interleaving approaches to concurrency are inadequate for proper treatment of properties such as fairness [14].

Additionally, in previous work [19,18] we have been concerned with the development of a formal model of components which allows for reasoning about generic properties of components and their composition. In this paper, we are concerned with deriving from that model a class of automata which generates the objects of the model and only them. The corresponding automata can be seen as an extension that brings the model a step closer to automation and eventually tool support.

The paper is structured as follows. In Section 2 we give an account of our model of components and outline component composition. In Section 3, we introduce Σ-automata for describing component behaviour. We start with a type of abstract machine and then consider concurrency as a further structural property which leads to the definition of Σ-automata. We also outline preliminary work on a formal notion of composition in terms of Σ-automata. The paper finishes with some concluding remarks and ideas for future work.

2 A model of components

In this section we briefly outline the model of components we are considering. The presentation has been restricted to the key concepts behind the model. Further details and a proper presentation can be found in [19,18].

In the familiar 'design by contract' paradigm [15], a component provides services to other components and, possibly, requires services (pre-condition) from other components in order to deliver those promised (post-condition). 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 light of the contractual use of components, the static semantics of a component is captured in terms of two disjoint sets of interfaces. Those the component requires and those that the component provides. Furthermore, the static semantics specifies for each interface the operations it supports. Let I be the set of names for interfaces and Op be the set of operations associated with interfaces in I.

Definition 2.1 A *component signature* is a tuple $\Sigma = (P_\Sigma, R_\Sigma, \beta_\Sigma)$ where

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

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

The dynamic semantics of a component consists of a set of possible behaviours. Each behaviour associates a sequence of operations with every interface. We define V_Σ to be the set of all functions $\underline{v} : I_\Sigma \to Op^*$ such that for each $i \in I_\Sigma$, we have $\underline{v}(i) \in \beta_\Sigma(i)^*$. We shall refer to elements of V_Σ as *component vectors*.

By $\beta_\Sigma(i)^*$ we denote the set of finite sequences over $\beta_\Sigma(i)$. Mathematically, V_Σ is the cartesian product of the sets $\beta_\Sigma(i)^*$. Component vectors are essentially n-tuples of sequences where each coordinate corresponds to an interface of the component (hence n is the number of component interfaces) and contains a finite sequence of events (e.g. operation calls) that may occur on that interface. The idea is that behaviour of the component as a whole can be described by assigning such a sequence to each of its interfaces.

We may now define a component by restricting to an appropriate subset of V_Σ comprising component vectors that describe intended behaviour only.

Definition 2.2 A *component* c is a pair (Σ, V), where

- Σ is the signature of c

- $V \subseteq V_\Sigma$ (and $V \neq \emptyset$) is the set of behaviours of c.

Thus, the static structure of a component is described by a signature Σ while its behaviour is described by a component language V, which is essentially a 'language' of vectors over Σ.

2.1 Properties of a component language

In this section we introduce the basic operations which will allow us to manipulate a component language and reason about component behaviour.

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

- $\underline{u}.\underline{v}$ to be the unique vector $\underline{w}$ such that $\underline{w}(i) = \underline{u}(i).\underline{v}(i)$, for each $i \in I_\Sigma$ (*concatenation*)
- $\underline{u} \leq \underline{v}$ iff $\underline{u}(i) \leq \underline{v}(i)$, for each $i \in I_\Sigma$ (*prefix ordering*)

It is easy to see that V_Σ is a monoid with binary operation '.' and identity $\underline{\Lambda}_\Sigma$, where $\underline{\Lambda}_\Sigma$ is the unique vector with $\underline{\Lambda}_\Sigma(i) = \Lambda$, for each $i \in I_\Sigma$, and Λ denotes the empty sequence. Furthermore, V_Σ is a partially ordered set (poset) with partial order '$\leq$' and bottom element $\underline{\Lambda}_\Sigma$.

Now based on the order-theoretic properties of V_Σ we may further define, for $\underline{u}, \underline{v} \in V_\Sigma$,

- $\underline{u} \sqcap \underline{v}$ to be the vector $\underline{w}$ which satisfies $\underline{w}(i) = min(\underline{u}(i), \underline{v}(i))$, for each i
- $\underline{u} \sqcup \underline{v}$ (if it exists) to be the vector $\underline{w}$ which satisfies $\underline{w}(i) = max(\underline{u}(i), \underline{v}(i))$

Note that $\underline{u} \sqcup \underline{v}$ is defined only when $max(\underline{u}(i), \underline{v}(i))$ exists, for each i. In terms of partial orders, these operations essentially give the greatest lower bound and the least upper bound, respectively, of $\underline{u}, \underline{v} \in V_\Sigma$, in the usual sense of lattices and domain theory [5,29]. To anticipate, greatest lower and least upper bounds have a significant role to play in defining the normality property (cf Definition 2.5) which allows us to restrict to a class of components, the so-called *well-behaved* components.

We next consider a right-cancellation operator '/'. Intuitively, if $\underline{u}$ is an initial part of behaviour $\underline{v}$ so that $\underline{u} \leq \underline{v}$, then $\underline{v}/\underline{u}$ is the 'continuation' of $\underline{u}$ that extends it to $\underline{v}$. Put formally, if $\underline{u} \leq \underline{v}$, then we define $\underline{v}/\underline{u}$ to be the unique element $\underline{z} \in V_\Sigma$ such that $\underline{u}.\underline{z} = \underline{v}$

The right-cancellation operator is particularly useful for defining the transition structure of the corresponding automata since it determines what events occur in going from a behaviour represented by $\underline{u}$ to a behaviour represented by $\underline{v}$. We will have more to say about this when we have also defined well-

behaved components.

2.2 Well-behaved components

In our approach towards modelling component behaviour, we restrict to a particular class of components. These are components whose language is discrete and locally left-closed. These properties are defined as follows (see also [19]).

Definition 2.3 Let $c = (\Sigma, V)$ be a component, then V is *discrete* iff, $\underline{\Lambda} \in V$ and whenever $\underline{u}, \underline{v}, \underline{w} \in V$ such that $\underline{u}, \underline{v} \leq \underline{w}$, then (i) $\underline{u} \sqcup \underline{v} \in V$ and (ii) $\underline{u} \sqcap \underline{v} \in V$

Note that $\underline{u} \sqcup \underline{v} \in V$ is understood as asserting that $\underline{u} \sqcup \underline{u}$ is defined.

Definition 2.4 Let $c = (\Sigma, V)$ be a component, $i \in I_\Sigma$ and $x \in \beta_\Sigma(i)^*$. Then, V is *locally left-closed* iff, whenever $\underline{v} \in V$ such that $\Lambda < x < \underline{v}(i)$, then there exists $\underline{u} \in V$ such that $\underline{u} \leq \underline{v}$ and $\underline{u}(i) = x$.

Discreteness captures the fact that a system's computations always have a starting point and imposes a finiteness constraint in the sense that it excludes infinite ascending or descending chains of events with respect to time ordering. In order to obtain a precise description of discrete behaviour, we further require that every occurrence of an event (e.g. operation call) is 'recorded' in the component language V. This motivates the local left-closure property.

Definition 2.5 Let $c = (\Sigma, V)$ be a component. The set of behaviours V is *normal* iff it is locally left-closed and discrete. Also, c is *well-behaved* if V is normal.

Well-behavedness of the corresponding component reflects the fact that the guarantees that accrue from discreteness and local left-closure are 'embedded' in its behaviour.

A well-behaved component can be associated with an order-theoretic structure called *behavioural presentation* [24]. This non-interleaving behavioural model mildly generalises event structures [20] by considering the time ordering of events to be a pre-order rather than a partial order, thereby allowing the representation of simultaneity as well as concurrency.

Apart from the theoretical motivation, discreteness and local left-closure can have practical benefits for component-based design, as shown in [19]. The idea is that in checking a component language against these properties it is possible to identify missing behaviours - either undesirable or, simply, unthought in design.

In a normal component language, and based on consequences of local left-closure in particular, we may define an ordering among component vectors in

which one is 'immediately beneath' the other, allowing no other vector in V to exist in between them.

Definition 2.6 Let $c = (\Sigma, V)$ be a component and $\underline{u}, \underline{v} \in V$. Then, $\underline{v}$ *covers* $\underline{u}$ in V, and we write $\underline{u} \lhd_V \underline{v}$ iff (i) $\underline{u} \leq \underline{v}$ and $\underline{u} \neq \underline{v}$ and (ii) if $\underline{z} \in V$ such that $\underline{u} \leq \underline{z} \leq \underline{v}$, then $\underline{z} = \underline{u} \vee \underline{z} = \underline{v}$

The normality property also has as a consequence that component vectors in V decompose into products of vectors, each of which has at most one operation call per coordinate. Mathematically, such vectors comprise the set $E_\Sigma = \{\underline{e} \in V_\Sigma \setminus \{\underline{\Lambda}_\Sigma\} : i \in I_\Sigma \Rightarrow |\underline{e}(i)| \leq 1\}$ where $|x|$ denotes the length of sequence x. We also define $E_\Sigma^\perp = E_\Sigma \cup \{\underline{\Lambda}_\Sigma\}$. Thus, the set E_Σ consists of 'column vectors', each of whose coordinates is either the empty sequence or a single action. Intuitively, $\underline{e} \in E_\Sigma$ represents a *simultaneity class* of events; precisely those events $\underline{e}(i)$ with $\underline{e}(i) > \Lambda$. These correspond to simultaneity classes of event occurrences in the corresponding behavioural presentation.

The relation '$\lhd$' determines immediate predecessors / successors in a component language and combined with the corresponding column vectors that extend a predecessor to its immediate successor, we may talk about *immediate causality* in the sense of [13] where labelled prime event structures are used to interpret interactions between instances appearing in a sequence diagram. The following result relates '$\lhd$' with the right-cancellation operator.

Proposition 2.7 *Let $V \subseteq V_\Sigma$ be normal and let $\underline{u}, \underline{v} \in V$. If $\underline{u} \lhd \underline{v}$, then $\underline{v}/\underline{u} \in E_\Sigma$.*

To anticipate further, this allows us to define a transition structure on V when it comes to associating well-behaved components with automata in Section 3.

Example 2.8 Consider the example component of Fig. 1 which is expected to operate as follows (perhaps in the context of a given scenario):

- the component receives calls to operation c on interface p_1 (from other components or the environment)
- once an operation call c is received, the component responds by making a call to operation d on each of the interfaces r_1 and r_2 (implementation of these interfces is provided by other components)
- the component then proceeds to make an operation call t on interface r_3 and is then ready to receive a new c on p_1

The signature Σ_{ex} of the component is given by $\Sigma_{ex} = (P_{\Sigma_{ex}}, R_{\Sigma_{ex}}, \beta_{\Sigma_{ex}})$ where $P_{\Sigma_{ex}} = \{p_1\}$, $R_{\Sigma_{ex}} = \{r_1, r_2, r_3\}$ and $\beta_{\Sigma_{ex}}(p_1) = \{c\}$, $\beta_{\Sigma_{ex}}(r_1) = \{d\} = \beta_{\Sigma_{ex}}(r_2)$ and $\beta_{\Sigma_{ex}}(r_3) = \{t\}$. We may check that $P_{\Sigma_{ex}} \cap R_{\Sigma_{ex}} = \emptyset$

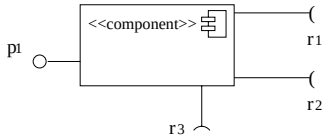

Fig. 1.

and $P_{\Sigma_{ex}} \cup R_{\Sigma_{ex}} = I_{\Sigma_{ex}}$.

We may now obtain a set of behaviours that indicates what would be desirable behaviour of the *ex* component with respect to the informal description of its functionality given above. Hence, we consider vectors over the signature Σ_{ex} and if we write (x, y, z, w) for $\underline{v}(p_1) = x, \underline{v}(r_1) = y, \underline{v}(r_2) = z$ and $\underline{v}(r_3) = w$ such description of behaviour would be interpreted into the following component language
$V_{ex} = \{(\Lambda, \Lambda, \Lambda, \Lambda), (c, \Lambda, \Lambda, \Lambda), (c, d, \Lambda, \Lambda), (c, \Lambda, d, \Lambda), (c, d, d, \Lambda), (c, d, d, t)\}$
Each component vector is to be understood as a *snapshot* indicating which events have already occurred, and on which interface.

Note that $c_{ex} = (\Sigma_{ex}, V_{ex})$ is a component, by construction. It is worth pointing out here that the set V_{ex} would normally be obtained by a (partial) description of behaviour given by a component developer, perhaps even in the form of a sequence diagram describing the interactions of the component with regard to the fragment of behaviour we are considering. For instance, in [17] we describe the use of LSCs [4] (restricted to basic features) as a starting point for a version of our component model. Further considerations on obtaining the component language are however beyond the scope of the present paper.

The component of our example is well-behaved. We do not check against normality here due to space limitations. It is relatively straightforward to check that this is indeed the case by drawing the Hasse diagram for the order structure of V_{ex}.

With regard to Definition 2.6 of '$\lhd$', we have
$(\Lambda, \Lambda, \Lambda, \Lambda) \lhd (c, \Lambda, \Lambda, \Lambda)$ and $(c, \Lambda, \Lambda, \Lambda) \lhd (c, d, \Lambda, \Lambda)$
$(c, \Lambda, \Lambda, \Lambda) \lhd (c, \Lambda, d, \Lambda)$ and $(c, d, \Lambda, \Lambda) \lhd (c, d, d, \Lambda)$
$(c, \Lambda, d, \Lambda) \lhd (c, d, d, \Lambda)$ and $(c, d, d, \Lambda) \lhd (c, d, d, t)$
The set of column vectors (events) associated with the component (wrt the fragment of behaviour considered in our example) is
$\underline{e}_1 = (c, \Lambda, \Lambda, \Lambda)$, $\underline{e}_2 = (\Lambda, d, \Lambda, \Lambda)$, $\underline{e}_3 = (\Lambda, \Lambda, d, \Lambda)$, $\underline{e}_4 = (\Lambda, \Lambda, \Lambda, t)$
and thus, $E_{\Sigma_{ex}} = \{\underline{e}_1, \underline{e}_2, \underline{e}_3, \underline{e}_4\}$. The example is continued in Section 3. □

2.3 Component Composition

Insofar we have been concerned with a single component. In this section, we briefly outline composition within our framework. Full details can be found

in [18]. The key idea behind composition 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 be composed if their restrictions to interface i are the same. From the composition of those behaviours, the sequence of operation calls corresponding to i is removed.

Composition takes place on complementary interfaces; that is, interfaces required by one component and provided by the other, in the spirit of complementary labels in CCS [16] or CSP [11]. We assume disjoint sets of provided and required interfaces for each of the components. As a result, a condition is required on the component signatures. We define Σ_1, Σ_2 to be *consistent*, and write $\Sigma_1 \downarrow \Sigma_2$ iff

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

Note that consistency among component signatures implies that complementary interfaces are those that belong to the set

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

The signature of the composite is formed from the signatures of the individual components by eliminating all complementary interfaces. In effect, the composite signature internalises all shared interfaces. Thus, we define $\Sigma_1 \oplus \Sigma_2 = \Sigma$ where

- $P_\Sigma = (P_{\Sigma_1} \cup P_{\Sigma_2}) \backslash (R_{\Sigma_1} \cup R_{\Sigma_2})$
- $R_\Sigma = (R_{\Sigma_1} \cup R_{\Sigma_2}) \backslash (P_{\Sigma_1} \cup P_{\Sigma_2})$
- $\beta_\Sigma(i) = \beta_{\Sigma_k}(i)$ wherever $i \in I_{\Sigma_k}, k = 1, 2$

Having defined the signature of the composite we now proceed to define its component language, in a fashion similar to the treatment of a single component. First, we describe how component vectors are composed.

Definition 2.9 Let $c_1 = (\Sigma_1, V_{\Sigma_1})$ and $c_2 = (\Sigma_2, V_{\Sigma_2})$ be components. The vectors $\underline{u}_1 \in V_{\Sigma_1}$ and $\underline{u}_2 \in V_{\Sigma_2}$ are *consistent*, and we write $\underline{u}_1 \downarrow \underline{u}_2$, if

$$\underline{u}_1 \lfloor_{I_{\Sigma_1} \cap I_{\Sigma_2}} = \underline{u}_2 \lfloor_{I_{\Sigma_1} \cap I_{\Sigma_2}}$$

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

$$\underline{u}_1 \oplus \underline{u}_2 = (\underline{u}_1 \cup \underline{u}_2) \lfloor_{I_{\Sigma_1} \triangle I_{\Sigma_2}}$$

where $I_{\Sigma_1} \triangle I_{\Sigma_2} = (I_{\Sigma_1} \setminus I_{\Sigma_2}) \cup (I_{\Sigma_2} \setminus I_{\Sigma_1})$ and $\underline{u}_1 \cup \underline{u}_2 : I_{\Sigma_1} \triangle I_{\Sigma_2}$ satisfies

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

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

We may now give a formal definition of composition of components.

Definition 2.10 Let $c_k = (\Sigma_k, V_{\Sigma_k})$, for each k, be components. Define their composition $c_1 \oplus c_2 = (\Sigma, V)$ where,

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

Finally, we will need some notation for the projection of vectors of the composite onto vectors of its constituent components. For $\underline{v} \in V_{\Sigma_1} \oplus V_{\Sigma_2}$, with $\Sigma_1 \downarrow \Sigma_2$, we define $\underline{v}_{[k]} = \underline{v}\lfloor_{I_{\Sigma_k}}$, where $k = 1, 2$. This construction is reminiscent of the projections used to give the trace semantics of parallel composition in COSY [12] and CSP [11].

3 Automata for modelling component behaviour

In this section we define a class of automata, the so-called Σ-*automata*, for modelling the observable behaviour of components. Full details, together with the complete proofs, can be found in [26]. The proposed automata can be seen as an elaboration of asynchronous transition systems [23,2] and a specialisation of hybrid transition systems [25].

In a normal component language V, a vector $\underline{z}$ extends a vector $\underline{u}$ to a vector $\underline{v}$ if $\underline{v} = \underline{u}.\underline{z}$ and there is no other vector in V that lies strictly between $\underline{u}$ and $\underline{v}$. The latter requirement can be expressed by saying that $\underline{v}$ covers $\underline{u}$, in the sense of Definition 2.6. The continuation $\underline{z}$ which extends $\underline{u}$ to $\underline{v}$ is defined using the right-cancellation operator, i.e. $\underline{v}/\underline{u} = \underline{z}$. In a normal component language such continuations turn out to be elements of E_Σ (by Proposition 2.7). This observation gives a transition relation which leads to the definition of a type of transition systems.

Definition 3.1 Let Σ be a component signature. We define a Σ-*machine* to be a pair $M = (Q, \succ)$ where

- Q is a set of states
- $\succ \subseteq Q \times E_\Sigma \times Q$ is the *transition* relation, and we write $q \stackrel{\underline{e}}{\succ} q'$ for $(q, \underline{e}, q') \in \succ$

which satisfies:

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

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

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

We will write $q \succ\!\!\succ^{\underline{e}}$ to denote that there exists $q' \in Q$ such that $q \succ\!\!\succ^{\underline{e}} q'$.

Note that condition (i) includes the case that $\underline{e} = \underline{e}'$ in which case the condition can be rewritten as $q \succ\!\!\succ^{\underline{e}} q_1 \wedge q \succ\!\!\succ^{\underline{e}} q_2 \Rightarrow q_1 = q_2$. This condition guarantees unambiguity and also relates to point (iii) of the subsequent definition of Σ-automata (cf Definition 3.6).

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

Definition 3.2 Let $M = (Q, \succ)$ be a Σ-machine, $q \in Q$. Define $q \rightarrow^{\underline{u}} q'$ if

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

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

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

This gives the *execution vectors* of a Σ-machine. Effectively, these are component vectors formed by repeatedly concatenating column vectors $\underline{e}$, and can be understood as describing sequences of individual transitions.

Point (i) of Definition 3.2 refers to internal transitions which, perhaps not surprisingly, become significant when it comes to composition. Point (ii) of the definition says that whenever there is a component vector that is a concatenation of another vector, $\underline{v}$, and a column vector, then there is a state which takes you to the target state via the simple transition of the column vector, and that state is reachable from the source state through transition(s) implied by vector $\underline{v}$. It can be seen that this may involve decomposition of a component vector into a series of concatenations with column vectors from E_Σ, as shown in [27]. This is further exploited in showing that the vector language of the corresponding Σ-automaton is locally left-closed, as part of establishing that a Σ-automaton generates a normal component language.

Before introducing Σ-automata we discuss how a Σ-machine can be derived from a well-behaved component. This is done by taking component vectors in V as states and defining a transition relation in a way that reflects the observation that behaviours may be seen to be built up from the empty vector by repeatedly concatenating column vectors to it. In fact, this takes up on the ideas presented prior to defining Σ-machines.

Definition 3.3 Let $c = (\Sigma, V)$ be a component with V normal. Define $M_c =$

$(V, \succ_V)$ where

$$\underline{u} \succ_V^{\underline{e}} \underline{v} \;\Leftrightarrow\; \underline{u} \lhd \underline{v} \wedge \underline{v}/\underline{u} = \underline{e}$$

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

Note that $\underline{v}/\underline{u} \in E_\Sigma$, whenever $\underline{u} \lhd \underline{v}$, by Proposition 2.7 so that the definition makes sense.

This construction gives a Σ-machine M_c. Moreover, it can be shown that the vector language generated by M_c from intial state $\underline{\Lambda}_\Sigma$ determines the same component using the execution vectors of Definition 3.2. In our notation, this is expressed as $V(M_c^*) = V$.

We are now set to consider true-concurrency in a component language and the corresponding Σ-machine, and this while still dealing with normal component languages. In order to express concurrency explicitly in this context, where the same events may sometimes be concurrent and sometimes not, we define an independence relation on component vectors (cf Definition 3.4) and determine its relationship to the transition structure of Σ-machines (cf Definition 3.5). This results in additional constraints on Σ-machines, which lead to the definition of Σ-automata.

We start by considering an independence relation on component vectors.

Definition 3.4 Let $\underline{u}, \underline{v}$ be component vectors in V_Σ. $\underline{u}$ and $\underline{v}$ are *independent*, and we write $\underline{u}\ ind\ \underline{v}$, iff

$$\forall i \in I_\Sigma : \underline{u}(i) > \Lambda \Rightarrow \underline{v}(i) = \Lambda$$

The definition is motivated by the fact that behaviours $\underline{u}$ and $\underline{v}$ may take place independently so long as they engage different interfaces of the component. It might be worth pointing out that independence alone is not enough to guarantee concurrency. This should become clear by examining the following definition.

Definition 3.5 Let $M = (Q, \succ)$ be a Σ-machine. We define a relation $I^M \subseteq Q \times E_\Sigma \times E_\Sigma$, and we write $\underline{e}_1\ I_q^M\ \underline{e}_2$ for $(q, \underline{e}_1, \underline{e}_2) \in I^M$, by

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

We shall, as usual, drop the superscript when it is clear from context.

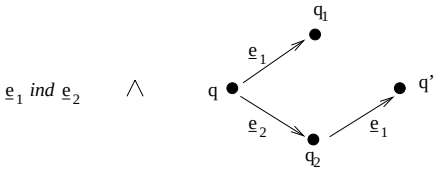

Fig. 2. Concurrent transitions $\underline{e}_1, \underline{e}_2$

Thus, the minimal requirement for concurrency at state q is that both independent outgoing transitions are enabled and both occur between states q and q' in *no* particular order. This is pictured in Figure 2.

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

We are now set to refine Σ-machines to Σ-automata, taking into account both relations.

Definition 3.6 Let Σ be a signature. A Σ-*automaton* M is a Σ-machine $M = (Q, \succ)$ satisfying

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

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

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

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

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

Note that by Definition 3.5 and (i) of Definition 3.6 we have that I_q is symmetric and irreflexive. Symmetricity reflects the fact that concurrency is always mutual while irreflexivity prohibits considering an event as being concurrent with itself.

Condition (i) is characteristic of automata for non-interleaving representation of behaviour and is sometimes called the *lozenge rule* [23,25,27]. Effectively, it says that if two independent events have occurred between two states q and $\hat{q}$, then they have happened in no particular order. In other words, it should be possible for them to have occurred with their order interchanged. This is depicted in Figure-3.

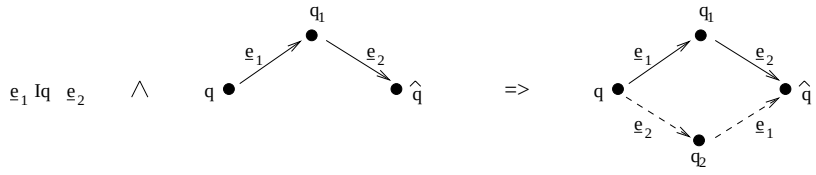

Fig. 3. Condition (i) of Definition 3.6

Condition (ii) relates to discreteness of the generated language. A few words are in order to explain this further. Discreteness requires that elements bounded above in the vector language have their least upper bound and greatest lower bound in it. Therefore, in order to guarantee discreteness we need conditions under which posets are (finite) lattices. Subsequent analysis in [26]

shows that V is a discrete subset of V_Σ precisely when $\downarrow \underline{u} = \{\underline{v} \in V : \underline{v} \leq \underline{u}\}$ is a sublattice of V_Σ. This turns out to be the case only when V satisfies the following

$$\forall \underline{u}, \underline{v}, \underline{x} \in V, \underline{u} \neq \underline{v} \wedge \underline{u} \lhd \underline{x} \wedge \underline{v} \lhd \underline{x} \Longrightarrow \underline{x} = \underline{u} \sqcup \underline{v} \wedge \underline{u} \sqcap \underline{v}$$

This is the so-called *lower lozenge property* (LLP), which essentially says that whenever we have the upper half of a lozenge, then we have the whole lozenge. It manifests itself in the structure of Σ-automata in the form of condition (ii). It is illustrated in Figure 4.

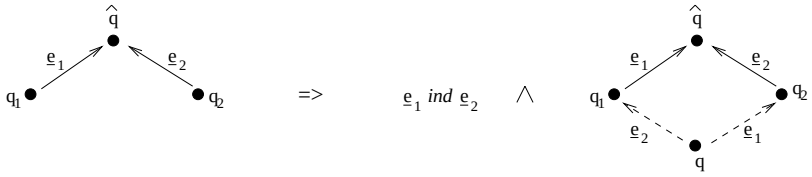

Fig. 4. Condition (ii) of Definition 3.6

Condition (iii) excludes the possibility that an execution vector may be produced in two different ways from sequences of individual transitions. In other words, when the first part of an execution vector takes us from its initial state to an intermediate state, then the remaining part takes us from that state to the (execution vector's) final state. Dually, we may state the same for the second part of the execution vector. The condition is depicted in Figure-5.

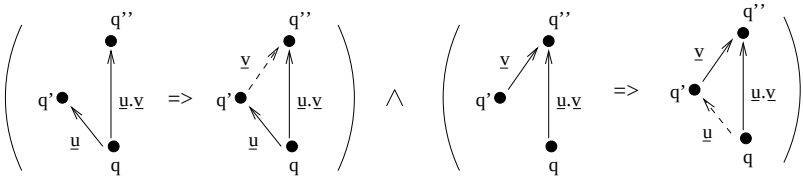

Fig. 5. Condition (iii) of Definition 3.6

Condition (iv) says that if two distinct transitions can start off the same behaviour from q, i.e. be part of the same execution vector from q, then they must do so concurrently. The motivation for this condition is not hard to see. Given a component vector $\underline{u}$ which describes behaviour of the component at state q, the two distinct transitions $\underline{e}_1$, $\underline{e}_2$ essentially provide two different ways, say $\underline{v}_1, \underline{v}_2$, of extending to a behaviour described by $\underline{x}$. In other words, $\underline{u}.\underline{v}_1 = \underline{x}$ and $\underline{u}.\underline{v}_2 = \underline{x}$. But this implies that $\underline{v}_1$ and $\underline{v}_2$ describe the same behaviour. Since, $\underline{e}_1$ is a prefix of $\underline{v}_1$, $\underline{e}_2$ of $\underline{v}_2$ and $\underline{e}_1, \underline{e}_2$ are distinct, we can not have $\underline{v}_1 = \underline{v}_2$ in general. This will only be the case if $\underline{e}_1, \underline{e}_2$ are concurrent. This is the point of condition (iv).

UML [21] is widely used for modelling and documenting software systems. Though it was not developed for component-based design as such, some of

its notation can be useful in bringing our component model closer to more conventinoal approaches to software design. With regard to UML 2.0 state machines and the notion of a compound transition defined therein (pp 500-1 in [21]), condition (iv) can be seen as a formalisation of the case where the head of a compound transition has multiple transitions to a set of orthogonal states (fork). Also, by applying condition (iv) to the conclusion of condition (ii) we may say the same for the tail of a compound transition in UML 2.0 state machines.

The only problem with regard to establishing a relation between compound transitions in UML and condition (iv) (and (ii)), is that the semantics of compound transitions as given in the UML 2.0 spec document does not allow triggers on transitions entering a join or emanating from a fork (pp 470-4 in [21]). According to our 'semantics', transitions can be labelled by an event in both cases and the trigger of the fork or join is the conjunction of the triggers (events) of the individual transitions. Our interpretation is that they are concurrent (see condition (iv)). In fact, this is consistent with the STATEMATE semantics of joins and forks (pp 302-3 in [8]) in statecharts [7].

As a final note on Definition 3.6, it can be seen that condition (iii) is a global rather than a local property. This makes checking against it difficult. [26] establishes the following for this purpose. Let $M = (Q, \succ)$ be a Σ-machine and let $V \subseteq V_\Sigma$. If (a) there exists an onto function $\phi : V \to Q$ such that $\phi(\underline{u}) \succ^{\underline{e}} \phi(\underline{v})$ iff $\underline{v} = \underline{u}.\underline{e}$ and, (b) if $\underline{u}, \underline{v} \in V$ and $\underline{u} \leq \underline{v}$, $(\underline{u} \neq \underline{v})$, then there exists $\underline{e} \in E_\Sigma$ such that $\underline{u}.\underline{e} \in V$ and $\underline{u}.\underline{e} \leq \underline{v}$, then M satisfies (iii) of Definition 3.6.

Example 3.7 Considering the component of Example 2.8, we may now define a tuple $M_{ex} = (Q_{ex}, \succ_{ex})$ where $Q_{ex} = V_{ex}$ and $\succ_{ex}$ is given by

$$q \succ^{\underline{e}}_{ex} \; iff \; \exists \underline{e} \in E_{\Sigma_{ex}} \; and \; \underline{u}, \underline{v} \in V_{ex} \; s.t. \; \underline{u} \lhd \underline{v} \wedge \underline{v}/\underline{u} = \underline{e}$$

For the component vectors in V_{ex} we have,

$\underline{\Lambda} \succ^{\underline{e}_1} (c, \Lambda, \Lambda, \Lambda)$ $\quad$ $(c, \Lambda, \Lambda, \Lambda) \succ^{\underline{e}_2} (c, d, \Lambda, \Lambda)$

$(c, \Lambda, \Lambda, \Lambda) \succ^{\underline{e}_3} (c, \Lambda, d, \Lambda)$ $\quad$ $(c, d, \Lambda, \Lambda) \succ^{\underline{e}_3} (c, d, d, \Lambda)$

$(c, \Lambda, d, \Lambda) \succ^{\underline{e}_2} (c, d, d, \Lambda)$ $\quad$ $(c, d, d, \Lambda) \succ^{\underline{e}_4} (c, d, d, t)$

We check the conditions of Definition 3.1.

- $q \succ^{\underline{e}}$ and $q \succ^{\underline{e}'}$ with $\underline{e} \leq \underline{e}'$ for no $q \in Q_{ex}$, $\underline{e}, \underline{e}' \in E_{\Sigma_{ex}}$. Thus, (i) of Definition 3.1 holds.
- $q \succ^{\underline{e}} q'$ and $q \succ^{\underline{e}'} q'$ with $\underline{e} \neq \underline{e}'$ for no $q, \underline{e}, \underline{e}'$. Thus, (ii) of Definition 3.1 also holds.

Hence, M_{ex} is a Σ-machine.

In order to show further that M_{ex} is a Σ-automaton we need to consider local concurrency. We start by identifying independent column vectors. We have $\underline{e}_1 \; ind \; \underline{e}_{5-k}$, for $k = 1..3$, and $\underline{e}_2 \; ind \; \underline{e}_{5-k}$, for $k = 1, 2$ and also $\underline{e}_3 \; ind \; \underline{e}_4$. Hence, all $\underline{e} \in E_{\Sigma_{ex}}$ are independent in this case. According to Definition 3.5, if $\underline{e} \, I_q \, \underline{e}'$, then $\underline{e} \; ind \; \underline{e}'$ and $q \succ^{\underline{e}} q_1$ and $q \succ^{\underline{e}'} q_2 \succ^{\underline{e}} q'$. We observe that all events associated with the component are independent and therefore we need only check the latter requirement. $q \succ^{\underline{e}}$ and $q \succ^{\underline{e}'}$ hold only for $\underline{e}_2$ and $\underline{e}_3$, for which it is also true that $\exists q_1, q_2, q'$ such that $(c, \Lambda, \Lambda, \Lambda) = q \succ^{\underline{e}_2} q_1 = (c, d, \Lambda, \Lambda)$ and $(c, \Lambda, \Lambda, \Lambda) = q \succ^{\underline{e}_3} q_2 = (c, \Lambda, d, \Lambda) \succ^{\underline{e}_2} \; q' = (c, d, d, \Lambda)$.
Thus, $\underline{e}_2 \; I_{(c,\Lambda,\Lambda,\Lambda)} \; \underline{e}_3$.

Note that this example shows that independence alone is not enough to guarantee local concurrency. Furthermore, it also shows that two column vectors may be concurrent at some state ($q = (c, \Lambda, \Lambda, \Lambda)$ here) but not in others; for instance, we do not have $\underline{e}_2 \; I_{(c,d,d,\Lambda)} \; \underline{e}_3$.

Finally, we show that M_{ex} is a Σ-automaton by checking the conditions of Definition 3.6.

The first condition is relevant only for $\underline{e}_2, \underline{e}_3$ (take $q = (c, \Lambda, \Lambda, \Lambda), q_1 = (c, d, \Lambda, \Lambda), \hat{q} = (c, d, d, \Lambda)$) in which case we have $q \succ^{\underline{e}_3} \; q_2 \succ^{\underline{e}_2} \; \hat{q}$ for $q_2 = (c, \Lambda, d, \Lambda) \in Q_{ex}$. Thus, condition (i) of Definition 3.6 holds.

For condition (ii), we have that, $q_1 \succ^{\underline{e}} \hat{q}$, and $q_2 \succ^{\underline{e}'} \hat{q}$, and $\underline{e} \neq \underline{e}'$, is the case only when $\underline{e} = \underline{e}_2$ and $\underline{e}' = \underline{e}_3$. But for these column vectors we have that $\underline{e}_2 \; ind \; \underline{e}_3$ and there exists $q = (c, \Lambda, \Lambda, \Lambda)$ such that $q \succ^{\underline{e}_3} q_1$ and $q \succ^{\underline{e}_2} q_2$. Thus, condition (ii) holds.

For condition (iii), we find it easier to check against the associated lemma. We define a function $\phi : V_{ex} \to Q_{ex}$ by $\phi(\underline{u}) = q_k$, $\forall \underline{u} \in V_{ex}$, $k = 0..5$ so that
$\phi((\Lambda, \Lambda, \Lambda, \Lambda)) = q_0 \qquad \phi((c, \Lambda, \Lambda, \Lambda)) = q_1 \qquad \phi((c, d, \Lambda, \Lambda)) = q_2$
$\phi((c, \Lambda, d, \Lambda)) = q_3 \qquad \phi((c, d, d, \Lambda)) = q_4 \qquad \phi((c, d, d, t)) = q_5$
By definition of ϕ and the relations '$\lhd$' of example 2.8 the first condition (a) holds. Condition (b) follows from the component language V_{ex} together with the covers relation. Thus, we may deduce that condition (iii) of Definition 3.6 holds.

For condition (iv), the only case in which $q \succ^{\underline{e},\underline{e}'}$ is for $q = (c, \Lambda, \Lambda, \Lambda)$ and $\underline{e} = \underline{e}_2, \underline{e}' = \underline{e}_3$, for which we also have $\underline{e}_2, \underline{e}_3 \leq \underline{x} = (c, d, d, \Lambda)$. Thus, the premises of condition (iv) are met for $\underline{e}_2, \underline{e}_3$. But then, for these column vectors we have $\underline{e}_2 \; I_{(c,\Lambda,\Lambda,\Lambda)} \; \underline{e}_3$ and thus, condition (iv) of the definition holds.

Hence, M_{ex} is a Σ-automaton. It can be represented by a state diagram which - the *co* relation aside - conforms to the UML 2.0 state diagrams [21] notation and that of statecharts [7]. The state diagram for M_{ex} is given in Figure 6, where $q_1 \; co \; q_2$ implies that the transitions from q_1 to q_2 and q_3 take place in *no* particular order. This is expressed in terms of the associated

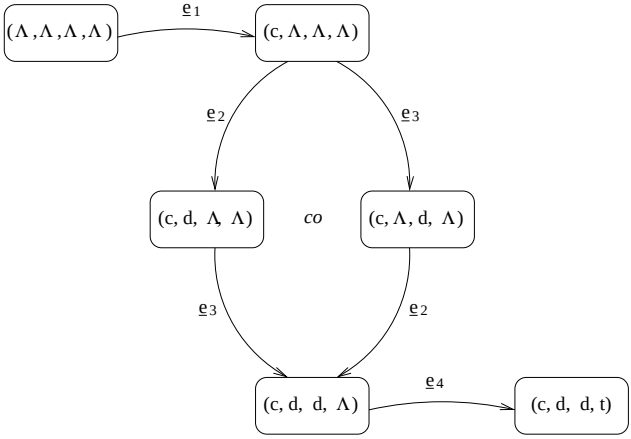

Fig. 6. The Σ-automaton M_{ex}

Σ-automaton by saying that they are related by I_q.

Note that without introducing the *co* relation between states, the diagram would simply denote a nondeterministic choice between taking $\underline{e}_2$ and then $\underline{e}_3$, and taking $\underline{e}_3$ and then $\underline{e}_2$. Since in a Σ-automaton we can express true-concurrency, we want a graphical notation that represents $\underline{e}_2, \underline{e}_3$ happening in no particular order. We opted for the use of *co* mainly driven by early suggestions for indicating independence between events on transitions in asynchronous transitions systems (e.g. shading the lozenge shape in [23,25] or using $\sim$ within a lozenge shape in [10]). A suitable choice of notation is yet an issue for further consideration. □

By exploiting the additional structure that went in to moving from Σ-machines to Σ-automata it can be shown that:

- well-behaved components generate Σ-automata from initial state $\underline{\Lambda}_\Sigma$ and the corresponding Σ-automaton generates back the same component language
- the vector language of a rooted Σ-automaton corresponds to a normal component language which in turn gives rise to a rooted Σ-automaton for which the initial state is $\underline{\Lambda}_\Sigma$.

In the latter construction, the question arises as to the relationship between the rooted Σ-automaton (M, q) we start with and the rooted Σ-automaton $(M_c, \underline{\Lambda}_\Sigma)$ derived from the corresponding component. The answer given in [27] is that they are bisimilar in the sense of strong bisimulation in [16].

The main results of this section are summarised in the following theorem.

Theorem 3.8 *Let C_Σ denote the class of all well-behaved components with signature Σ and let M^*_Σ denote the class of all rooted Σ-automata. Then,*

there exists an onto function $\sigma : M^*_\Sigma \to C_\Sigma$ *given by*

$$\sigma[M^*] = (\Sigma, V(M^*))$$

and furthermore,

$$\sigma[M^*_1] = \sigma[M^*_2] \iff M^*_1 \sim M^*_2$$

where $\sim$ *denotes that there is a bisimulation from* M^*_1 *to* M^*_2

3.1 Composition in terms of Σ*-automata*

In this section, we give a brief account of composition of Σ-automata. The intention is to show that the Σ-automata-theoretic framework described so far is indeed compositional. In a fashion similar to that of composition of components, the key idea is that a component vector $\underline{v}$ represents behaviour of the product $M_1||M_2$ providing it results from behaviours $\underline{v}_k$ of M_k, each k, which agree on complementary interfaces.

Essentially, the transition relation is given by $(q_1, q_2) \succ\!\!\!-^{\underline{e}} (q'_1, q'_2)$ if and only if for each k either $\underline{e}_{[k]} \neq \underline{\Lambda}_{\Sigma_k}$ and $q_k \succ\!\!\!-^{\underline{e}_{[k]}} q'_k$ or $\underline{e}_{[k]} = \underline{\Lambda}_{\Sigma_k}$ and $q_k = q'_k$. This is expressed more succinctly as $q_k \succ\!\!\!-\!\!\!\succ^{\underline{e}_{[k]}} q'_k$, each k. Mathematically,

$$q_1 \succ\!\!\!-\!\!\!\succ^{\underline{e}} q_2 \iff (q_1 \succ\!\!\!-^{\underline{e}} q_2) \vee (\underline{e} = \underline{\Lambda}_\Sigma \wedge q_1 = q_2)$$

In terms of notation, using $\underline{q}$ to denote that $(q_1, q_2) \in Q_1 \times Q_2$, and consequently $\underline{q}_k$ for q_k, each k, we may write $\underline{q} \succ\!\!\!-^{\underline{e}} \underline{q}'$ for the transition relation of the composite, which is translated in terms of the constituent automata as $\underline{q}_k \succ\!\!\!-\!\!\!\succ^{\underline{e}_{[k]}} \underline{q}'_k$, each k.

In defining the transition relation of the composite we need to take account of two possibilities:
i) $\underline{e}(i) \neq \Lambda$, $i \in I_{\Sigma_1} \cap I_{\Sigma_2}$, in which case $\underline{e}_{[k]} \neq \underline{\Lambda}_{\Sigma_k}$, for each k, and execution of the transitions from each automaton involves communication. This means that both transitions must be executed simultaneously so that the composite has a transition $(q_1, q_2) \succ\!\!\!-^{\underline{e}} (q'_1, q'_2)$.
ii) $\underline{e}(i) = \Lambda$, all $i \in I_{\Sigma_1} \cap I_{\Sigma_2}$, in which case there is no communication and execution of the transition of one of the constituent automata (the one for which $\underline{e}_{[k]} \neq \underline{\Lambda}_{\Sigma_k}$, $k = 1$ or $k = 2$) may occur independently of any transition in the other. Hence, the composite automaton has a transition $(q_1, q_2) \succ\!\!\!-^{\underline{e}} (q'_1, q_2)$ if $k = 1$, and $(q_1, q_2) \succ\!\!\!-^{\underline{e}} (q_1, q'_2)$ if $k = 2$.

Note that while $\underline{e}(i) = \Lambda$, all $i \in I_{\Sigma_1} \cap I_{\Sigma_2}$, it is still possible for $\underline{e}_{[k]} \neq \underline{\Lambda}_{\Sigma_k}$, each k. Composition would then force the rest of the events (those appearing on the coordinates which correspond to the non-connected interfaces of each i.e. $\underline{e}_{[k]}(i)$, each $k : i \notin I_{\Sigma_1} \cap I_{\Sigma_2}$) to be simultaneous while this clearly need not be the case in general. Thus, an additional requirement on the transition structure is that if the $\underline{e}_{[k]}$ are both non-null, then they must have some non-

empty coordinate in common.

Bringing the above concepts together we may now give a formal definition of composition.

Definition 3.9 Let $M_1 = (Q_1, \succ)$ and $M_2 = (Q_2, \succ)$ be Σ_k-machines, $k = 1, 2$ for which $\Sigma_1 \downarrow \Sigma_2$ and let $\Sigma = \Sigma_1 \oplus \Sigma_2$. Define $M_1||M_2 = (Q_1 \times Q_2, \succ)$, where $\succ \subseteq (Q_1 \times Q_2) \times E_\Sigma \times (Q_1 \times Q_2)$ is given by $\underline{q} \succ^{\underline{e}} \underline{q}' \iff$

- $\underline{q} \succ \succ^{\underline{e}_{[k]}} \underline{q}'$, for each k
- If $\underline{e}(i) = \Lambda$, all $i \in I_{\Sigma_1} \cap I_{\Sigma_2}$, then either $\underline{e}_{[1]} = \underline{\Lambda}_{\Sigma_1}$ or $\underline{e}_{[2]} = \underline{\Lambda}_{\Sigma_2}$.

Point (1) of the above definition refers to the case i) discussed above. Point (2) relates to case ii) and expresses the additional requirement that composition does not force otherwise independent column vectors to be necessarily simultaneous.

Our aim is to show that the proposed Σ-automata-theoretic framework is compositional, and to this end we need to establish that the composition (following Definition 3.9) of Σ-automata is itself a Σ-automaton.

First, we need to consider compatibility of Σ-automata in terms of their respective transitions. With regard to the discussion prior to Definition 3.9, we are concerned with the case where communication is involved. In this case, the corresponding components have (at least one) complementary interfaces and thus, their component vectors should agree on the corresponding non-empty coordinates. We may express this formally, and write $\underline{u}_1 \Downarrow \underline{u}_2$ iff

$$\forall i \in I_{\Sigma_1} \cap I_{\Sigma_2} : \underline{u}_1(i), \underline{u}_2(i) \neq \Lambda \Rightarrow \underline{u}_1(i) = \underline{u}_2(i)$$

Also, we may define the set of events (transitions) of a machine to be

$$E(M) = \{\underline{e} \in E_\Sigma | \exists q \in Q : q \succ^{\underline{e}}\}$$

Thus, we now have that $\underline{e}_1 \Downarrow \underline{e}_2$ iff on all non-empty common coordinates the $\underline{e}_k$ agree. Hence, '$\Downarrow$' does not cater for cases where, say, $\underline{e}_1(i) = \Lambda \wedge \underline{e}_2(i) \neq \Lambda$, $i \in I_{\Sigma_1} \cap I_{\Sigma_2}$. The following definition rectifies this by imposing that if two transitions have at least one non-empty common coordinate on which they agree, then this must be the case for all their common coordinates.

Definition 3.10 Let M_1 and M_2 be Σ_k-machines, $k = 1, 2$, and $\Sigma_1 \downarrow \Sigma_2$. Then, M_1, M_2 are *compatible*, and we write $M_1 \downarrow M_2$ if

$$\forall \underline{e}_1 \in E(M_1), \forall \underline{e}_2 \in E(M_2). \underline{e}_1 \Downarrow \underline{e}_2 \Rightarrow \underline{e}_1 \downarrow \underline{e}_2$$

This gives the compatibility condition within our automata-theoretic framework. One important consequence of this is that the execution vectors of the composite automaton are precisely those which project on execution vectors of the constituents, i.e. $(q_1, q_2) \rightarrow^{\underline{u}} (q_1', q_2') \iff q_k \rightarrow_k^{\underline{u}_{[k]}} q_k'$, for each k.

Second, we need to establish a relationship between local concurrency in the constituents and that of the composite automaton. We start by addressing independence. It is relatively straightforward to show that behaviours $\underline{u}, \underline{v}$ of the composite automaton can take place independently iff their projections onto the constituent automata are independent. Put formally,

$$\underline{u}\ ind\ \underline{v} \iff \underline{u}_{[k]}\ ind\ \underline{v}_{[k]},\ each\ k$$

This allows for concurrency in the composite automaton to be defined in terms of the translation of the composite's transition relation for the constituents.

Definition 3.11 Let M be a Σ-machine and $q \in Q$ and $\underline{e}_1, \underline{e}_2 \in E_\Sigma^\perp$. Define,

$$\underline{e}_1\ \hat{I}_M^q\ \underline{e}_2 \iff \underline{e}_1\ ind\ \underline{e}_2 \wedge (\exists q', q'', \hat{q} \in Q : q \succ\!\!\succ^{\underline{e}_1} q' \wedge q \succ\!\!\succ^{\underline{e}_2} q'' \succ\!\!\succ^{\underline{e}_1} \hat{q})$$

Now we are in a position to establish the relationship betwen concurrency in the constituents and concurrency in the composite automaton by

$$\underline{e}\ I_M^q\ \underline{f} \iff \underline{e}_{[k]}\ I_M^{q_k}\ \underline{f}_{[k]}, each\ k$$

where $\underline{e}, \underline{f} \in E_\Sigma$ and M_1, M_2 are Σ_k-machines, $k = 1, 2$ and $\Sigma_1 \downarrow \Sigma_2$ with $\Sigma = \Sigma_1 \oplus \Sigma_2$ and $M = M_1 || M_2$.

To sum up, we have considered a notion of compatibility among transitions of the constituent automata and related concurrency in constituents to that of the composite. It can now be shown that the composite of Σ-automata, following the construction given in Definition 3.9, is itself a Σ-automaton.

4 Conclusions and Future Work

We have presented an automata-based formalism for modelling the observable behaviour of components, in the presence of concurrency. We have also described how to compose Σ-automata. In fact, this extension to our existing component model [19,18] allows two approaches to composition: either generate the corresponding components and then compose or compose the automata and then generate a component from the composite automaton. The two approaches are shown to commute in [26].

Further work on composition is in progress and in particular, preservation of normality under automata composition. Preliminary results are encouraging. An interesting side effect of pursuing the dual aspects of composition - in terms of automata - has to do with relaxing the compatibility conditions that ensure preservation of normality in component composition [18].

Σ-automata can be seen as a usage protocol state machine for a component. They model both provides and requires assumptions and thus restrict the environment in a fashion similar to the interface automata of [6]. In contrast

to interface automata however, Σ-automata capture true-concurrency and this is carried on to the structure of the composite automaton, as discussed earlier.

The concept of concurrency in our component model is based on that introduced by C.A. Petri in his 1962 thesis and further discussed in [22]. Technically, nets at the condition-event level may be equipped with a semantics in terms of asynchronous transition systems, which may be shown to correspond to that of so-called process nets via a Mazurkiewicz trace language semantics for the asynchronous transition systems ([25], ch. 16-17). Further, an extension of this semantics involves the so-called hybrid transition systems in which transitions are associated with multisets of event names ([25], ch. 22). The Σ-automata described in this paper lie somewhere between the two, in that the underlying asynchronous transition system is equipped with a specialised association of transitions to column vectors (in E_Σ), which may be interpreted as multisets.

It may be instructive to relate our approach to the algebraic model of [3] where behaviours are also described in terms of finite and infinite (we only consider finite) sequences associated with ports. The dynamics of components are given in terms of functions $f : \mathcal{I} \rightarrow \mathcal{O}$ where $\mathcal{I}$, resp. $\mathcal{O}$, is the set of all vectors with input, resp. output, ports as coordinates. Under certain conditions these functions can be described by equations in the manner of process algebra while recursion is handled using fixed point techniques. [3] points out that such equations may be represented by automata. Curiously, the independence of two vectors with distinct coordinates non-empty, and the possible pitfalls which figure prominently in this paper, are not addressed.

In order to increase the scope for reuse in different contexts, the component developer should not have to foresee possible reuse contexts during design. This implies a need for dynamic coupling between provided and required services. Drawing on parametric contracts [1], essentially a finite state machine based approach, we would like to see whether environmental properties can become parameters of the component's provides/requires assumptions. Extending our model to address this in a concurrent setting, is one possible direction for future work.

In [17], we considered a scenario-based description of behaviour, in terms of LSCs [4], as the starting point for our component model. In particular, we showed how component interactions within a given scenario can give rise to an algebraic representation of behaviour, in terms of component vectors. Associating components with automata, as described in this paper, builds a bridge between algebraic and order-theoretic representation of component behaviour. This offers interesting perspectives concerning the move from a scenario-based specification to a state-based specification which we are keen

to explore further.

References

[1] S. Becker, R. H. Reussner, and V. Firus. Specifying Contractual Use, Protocols and Quality Attributes for Software Components. In *Proc. of 1st Int'l Workshop on Component Engineering and Methodology*, pages 13–22, 2003.

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

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

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

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

[6] L. de Alfaro and T. Henzinger. Interface Automata. In *Proc. of Foundations of Software Engineering (FSE'01)*, pages 109–120. ACM Press, 2001.

[7] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[8] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Trans. on Software Engineering and Methodology*, 5(4):293–333, October 1996.

[9] C. Heitmeyer. Managing Complexity in Software Development with Formally Based Tools. In *Proc. ETAPS 2004 workshop on Formal Foundations of Embedded Software and Component-Based Software Architectures (FESCA'04)*, volume 108 of *ENTCS*, pages 11–19. Elsevier, 2004.

[10] T. H. Hildebrandt and V. Sassone. Comparing Transition Systems with Independence and Asynchronous Transitions Systems. Technical Report RS-96-18, BRICS Report Series, University of Aarhus, June, 1996.

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

[12] R. Janicki and P. E. Lauer. *Specification and Analysis of Concurrent Systems: The COSY Approach*, volume 26 of *EATCS Monographs in Theoretical Computer Science*. Springer Verlag, 1992.

[13] J. Küster-Filipe. Modelling Concurrent Interactions. In C. Rattray, S. Maharaj, and C. Shankland, editors, *Proceedings of Algebraic Methodology and Software Technology (AMAST 2004)*, volume 3116 of *Lecture Notes in Computer Science*, pages 304–318. Springer, 2004.

[14] M. Z. Kwiatkowska. *Fairness in Non-Interleaving Concurrency*. PhD thesis, University of Leicester, 1989.

[15] B. Meyer. Applying "design by contract". *IEEE Computer*, 25:40–51, 1992.

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

[17] S. Moschoyiannis. Generating Snapshots of a Component Setting. In *Proc. ETAPS 2004 workshop on Formal Foundations of Embedded Software and Component-Based Software Architectures (FESCA'04)*, volume 108 of *ENTCS*, pages 83–98. Elsevier, 2004.

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

[19] S. Moschoyiannis, M. W. Shields, and J. Küster-Filipe. Formalising Well-Behaved Components. In H. Dang Van and Z. Liu, editors, *Proc. of FME 2003, workshop on Formal Aspects of Component Software (FACS'03)*, pages 121–142. UNU/IST Report No. 284, 2003.

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

[21] OMG. *UML 2.0 Superstructure Draft Adopted Specification*. OMG document ad/03-01-07, available from http://www.omg.org, August 2003.

[22] C. A. Petri. Concurrency. In G. Goos and J. Hartmanis, editors, *Proceedings of Advanced Course on General Net Theory of Processes and Systems*, volume 84 of *Lecture Notes in Computer Science*, pages 251–260. Springer-Verlag, 1979.

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

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

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

[26] M. W. Shields. An Automata Theory for Components. Technical Report SCOMP-TC-04-04, Department of Computing, University of Surrey, 2004.

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

[28] R. van Ommering. Building Product Populations with Software Components. In *Proceedings of International Conference on Software Engineering (ICSE'02)*, pages 255–265. ACM Press, 2002.

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