

Interacting DES: Modelling and Analysis*

Sherif Abdelwahed
Institute for Software Integrated Systems
Vanderbilt University
Nashville, TN, USA
sherif.abdelwahed@vanderbilt.edu

W. M. Wonham
Department of Electrical Engineering
University of Toronto
Toronto, Ontario, Canada
wonham@control.toronto.edu

Abstract – *In this paper a modelling and analysis paradigm for multiprocess discrete event systems is presented within the formal language and automata settings. The proposed modelling structure features explicit representation of the system components as well as their interaction constraints. The model is then extended for hierarchical multilevel systems.*

Keywords: Multiprocess discrete event systems, behavioural analysis, hierarchical systems.

1 Introduction

This paper presents a modelling and analysis framework for a general class of multiprocess discrete event systems with well-defined interaction between the system components. The proposed model structure, referred to as interacting discrete event system (IDES), provides a concrete and separate representation for the system components and their interaction specifications. In this setting, standard interaction specifications as well as custom ones can be incorporated directly into the modelling structure. Interaction specification in the IDES structure is directly linked to the system behaviour. The proposed framework can therefore provide a systematic way to explore the relationship between the components interaction and the system behaviour. Such information can then be utilized to develop efficient analysis procedures for multiprocess discrete event systems [1].

In the last two decades, several models for parallel and concurrent logical systems have been proposed. In [7], a general framework to express parallelism in logical multiprocess systems is introduced through which the relationship between several modelling formalisms is investigated. The relation between the system model and its observed behaviour has been addressed recently in formal language theory literature. In [4], the notion of vector controlled concurrent systems was introduced as a behavioural model for concurrent systems. In this approach the system is represented by a set of sequential

processes together with a vector synchronization mechanism controlling their mutual synchronization. The idea originated from the theory of path expressions [2] and its vector firing sequence semantics. The notion of vector firing sequences was also introduced in [6] as a semantic of COSY systems [3]. In another approach [5], the operation of “shuffle on trajectories” is introduced as a generalization of the parallel composition of asynchronous systems.

This paper aims to address the relationship between the system model and its behaviour from a formal language perspective. The proposed modeling can represent many standard synchronous/asynchronous constructs such as refinement, serial composition and interleaving. The model can also represent the tree-like structure of hierarchical interactions. Proofs of the theorems in this paper can be found in [1].

2 Preliminaries and notation

Let Σ be an alphabet representing the events in the process under consideration. A *string* or word is a sequence of events. We will write Σ^+ for the set of all non-empty finite strings with events in Σ , and $\Sigma^* = \Sigma^+ \cup \{\epsilon\}$, where $\epsilon \notin \Sigma$ denotes the empty string. A *language* over the alphabet Σ is any subset of Σ^* . The set of languages over Σ will be denoted $\mathcal{L}(\Sigma)$. A string $s' \in \Sigma^*$ is a *prefix* of $s \in \Sigma^*$, if $\exists u \in \Sigma^*$ such that $s'u = s$. The *prefix closure* of a language $H \subseteq \Sigma^*$, denoted \overline{H} , is the set of all strings in Σ^* that are prefixes of strings in H . The *complement* of a language $L \subseteq \Sigma^*$ is defined as $\Sigma^* - L$ and is denoted L^c .

We will extend the above notation to handle multiprocess systems. Let I be the index set of a collection of processes. An alphabet vector over I is a set $\{\Sigma_i | i \in I\}$ of alphabets. In the following we will use bold letters to distinguish vector quantities. Let $\Sigma = \{\Sigma_i | i \in I\}$ be an alphabet vector. The union of all alphabets in Σ will be denoted $\alpha(\Sigma)$ or simply Σ if no confusion arises. We will write Σ_s for the set of shared (synchronous) events in Σ , namely $\Sigma_s = \bigcup_{i \neq j} (\Sigma_i \cap \Sigma_j)$. A multi-process environment will be referred to as a *process space*. A process space is uniquely defined by its alphabet vector,

hence both terms designate the same thing.

A language vector over Σ is a set $\mathbf{L} = \{L_i \subseteq \Sigma_i^* \mid i \in I\}$. The set of all language vectors over Σ is denoted $\mathcal{L}(\Sigma)$. The language L_i is called the i th component of \mathbf{L} . Similarly a string vector is a set $\mathbf{s} = \{s_i \in \Sigma_i^* \mid i \in I\}$. For two language vectors \mathbf{L}' and \mathbf{L} over Σ , \mathbf{L}' is said to be a *language subvector* or simply a *subvector* of \mathbf{L} if $L'_i \subseteq L_i$ for all $i \in I$. In this case we write $\mathbf{L}' \sqsubseteq \mathbf{L}$. The *componentwise union* and *componentwise intersection* of \mathbf{L}' and \mathbf{L} are denoted $\mathbf{L}' \sqcup \mathbf{L}$ and $\mathbf{L}' \sqcap \mathbf{L}$, respectively.

The decomposition (vector projection) map $\mathbf{P}_\Sigma : \mathcal{L}(\Sigma) \rightarrow \mathcal{L}(\Sigma)$ associates each language $L \in \mathcal{L}(\Sigma)$ with the language vector $\{P_i L \mid i \in I\}$ where $P_i : \Sigma^* \rightarrow \Sigma_i^*$ is the natural projection map that erases all events other than those of the i th component of Σ . On the other hand, the composition (synchronous product) map $B_\Sigma : \mathcal{L}(\Sigma) \rightarrow \mathcal{L}(\Sigma)$ associates each language vector with its synchronous product $\|\mathbf{L}$. To simplify notation we will write $\mathbf{P}B_\Sigma$ to denote the composition $\mathbf{P}_\Sigma \circ B_\Sigma : \mathcal{L}(\Sigma) \rightarrow \mathcal{L}(\Sigma)$, and $B\mathbf{P}_\Sigma$ for the composition $B_\Sigma \circ \mathbf{P}_\Sigma : \mathcal{L}(\Sigma) \rightarrow \mathcal{L}(\Sigma)$.

3 Process space analysis

Behavioural analysis problems in the multiprocess space environment usually share two main arguments, namely, a system consisting of a set of concurrent processes, and a specification defining certain tasks or correctness criteria for the system. It is then required to check if the system satisfies the given specification and if not to check whether the system behaviour can be restricted through supervision to satisfy the specification. In both situations the system behaviour has to be compared to the specification. This comparison is based on the containment order on the set of languages.

In multiprocess environments, a system of concurrent processes is represented by a language vector while the specification is usually given as a language. Direct comparison between the two domains is not possible. Therefore, a transformation from one domain to the other is necessary for such comparison. Typically, the composition operation is used to generate the language representing the behaviour of a given language vector while the decomposition operation generates the vector projection of a given language. For any two languages L and L' in $\mathcal{L}(\Sigma)$ and any two language vectors \mathbf{L} and \mathbf{L}' in $\mathcal{L}(\Sigma)$, it is straightforward to verify the following result

Proposition 3.1

$$\begin{aligned} \mathbf{P}_\Sigma(L \cap L') &\sqsubseteq \mathbf{P}_\Sigma(L) \sqcap \mathbf{P}_\Sigma(L') \\ \mathbf{P}_\Sigma(L \cup L') &= \mathbf{P}_\Sigma(L) \sqcup \mathbf{P}_\Sigma(L') \\ B_\Sigma(\mathbf{L}) \cap B_\Sigma(\mathbf{L}') &= B_\Sigma(\mathbf{L} \sqcap \mathbf{L}') \\ B_\Sigma(\mathbf{L}) \cup B_\Sigma(\mathbf{L}') &\subseteq B_\Sigma(\mathbf{L} \sqcup \mathbf{L}') \end{aligned}$$

□

In order to compare the behaviour of a language vector to a given language, one of these two transformations has to be made. In the composition approach, the language vector of a given system is converted to the language generated by the system. In this case, behavioural comparison can be conducted for the system with respect to any given specification. However, it is well-known that the composition operation is intractable with respect to the number of components.

The decomposition (projection) operation is not computationally efficient either. However, in most cases the system is given as a set of components and the specification is given as a language. Therefore, it is only required to decompose the specification. Also, the state size of the specification is usually much less than the size of the composite system. Under these assumptions, it would be more efficient to decompose the specification into a language vector and then compare it componentwise with the language vector of the system. This will not work, however, as the outcome of this comparison does not generally reflect the relation between the behaviour of the system and the specification.

The problem here is that neither the composition nor the decomposition operation preserves behavioural information, or more precisely, the containment order on the set of languages. In the composition operation information is lost due to the synchronization constraints, whereas in the decomposition operation information is lost because of the ambiguity associated with partial observations.

4 Compact language vectors

The composition operation enforces strict synchronization of shared events, that is, shared events must be triggered simultaneously by the corresponding components of the system. Under this rule, it is possible that certain strings in one component do not synchronize with the other components of the system, and therefore these strings do not contribute to the overall behaviour of the system. As a result, componentwise comparison between language vectors cannot give precise information about the relation between their corresponding behaviours. This can be resolved by defining a class of language vectors in which the composition operation is totally order preserving.¹

Let Σ be a process space. Then, for any language vectors \mathbf{L} and \mathbf{L}' in $\mathcal{L}(\Sigma)$ we have

$$\mathbf{L}' \sqsubseteq \mathbf{L} \implies B_\Sigma(\mathbf{L}') \subseteq B_\Sigma(\mathbf{L})$$

The reverse direction does not hold in general. Therefore, the composition operation B_Σ is not fully order preserving. To get a closer look into this situation, consider the kernel of the map B_Σ . This kernel defines an

¹Within the domain of languages and language vectors, the term *order* refers to the (componentwise) containment order.

equivalence relation on the set of language vectors in $\mathcal{L}(\Sigma)$ in which two language vectors are equivalent if they generate the same language. Therefore, each coset of $\ker B_\Sigma$ contains a set of language vectors generating the same behaviour. It is straightforward to see that the set of language vectors within each coset of $\ker B_\Sigma$ is closed under componentwise intersection as indicated by Proposition 3.1. Hence, there is a unique minimal element (with respect to componentwise inclusion) in each coset that can generate the language associated with the coset. This minimal element is formally characterized as follows. A language vector \mathbf{L} over the process space Σ is said to be *compact* if it satisfies

$$(\forall \mathbf{L}' \in \mathcal{L}(\Sigma)) \quad B_\Sigma(\mathbf{L}) = B_\Sigma(\mathbf{L}') \implies \mathbf{L} \sqsubseteq \mathbf{L}'$$

Basically, a compact language vector contains the minimal set of components that is needed to generate its language. Hence, \mathbf{L} is compact if it is the minimal element in its coset in the partition $\ker B_\Sigma$.

It can be verified easily that for a language $L \in \mathcal{L}(\Sigma)$, the language vector $\mathbf{P}_\Sigma(L)$ is always compact. Therefore, if the components of a given system are known to be equal to the vector projection of the system behaviour then this system is compact.

Proposition 4.1 \mathbf{L} is compact if and only if $\mathbf{L} = \mathbf{P}B_\Sigma(\mathbf{L})$ \square

Therefore, a language vector is compact if and only if its components are exactly the vector projections of the composite behaviour of the system. Based on this result, the compact language vector that can generate the same behaviour as a language vector \mathbf{L} is $\mathbf{P}B_\Sigma(\mathbf{L})$. Hence, we can define a closure operator $\mathbf{C}_\Sigma : \mathcal{L}(\Sigma) \rightarrow \mathcal{L}(\Sigma)$ that associates each language vector in $\mathcal{L}(\Sigma)$ with the compact language vector that generates the same behaviour. That is, $\mathbf{C}_\Sigma(\mathbf{L}) = \mathbf{P}B_\Sigma(\mathbf{L})$. Note that redundant information in vector languages arises from the set of impossible string vectors, which is directly related to the shared behaviour of the system. Therefore, a more efficient procedure can be developed for the computation of $\mathbf{C}_\Sigma(\mathbf{L})$ by tracing only the shared behaviour of the system as follows. For $i \in I$, define the language vector \mathbf{L}^i derived from the language vector \mathbf{L} as follows

$$\mathbf{L}_i^i = \Sigma_i^* \quad \text{and} \quad (\forall j \in (I - \{i\})) \quad \mathbf{L}_j^i = P_s \mathbf{L}_j$$

where $P_s : \Sigma^* \rightarrow \Sigma_s^*$ is the natural projection of the set of shared events in the process space. Hence, \mathbf{L}^i is constructed by replacing the i th component of \mathbf{L} by Σ_i^* , and replacing all other components by the corresponding projection onto the shared events. The following proposition defines another way to compute the map \mathbf{C}_Σ . First the following lemma will be needed.

Lemma 4.1 Let Σ and Σ_o be an alphabet set such that $\Sigma_s \subseteq \Sigma_o \subseteq \Sigma$ and let $P_o : \Sigma^* \rightarrow \Sigma_o^*$ be the associated

natural projection. Then,

$$(\forall \mathbf{L} \in \mathcal{L}(\Sigma)) \quad P_o(B_\Sigma(\mathbf{L})) = B_\Sigma(P_o \mathbf{L})$$

\square

The above lemma is a simple extension of a result in [8]. The proof of this extension is direct based on the associativity of the synchronous product.

Proposition 4.2

$$\mathbf{C}_\Sigma(\mathbf{L}) = \{L_i \cap P_i(B_\Sigma(\mathbf{L}^i)) \mid i \in I\}$$

\square

Based on the above result, the computation of \mathbf{C}_Σ depends only on the shared behaviour of the system components. This result also confirms that asynchronous language vectors (containing no shared events) are always compact.

Proposition 4.3 Let \mathbf{L} and \mathbf{S} be two compact language vectors. Then

$$B_\Sigma(\mathbf{L}) \subseteq B_\Sigma(\mathbf{S}) \iff \mathbf{L} \sqsubseteq \mathbf{S}$$

\square

This says that the composition operation restricted to the set of compact language vectors is order preserving. Because of the importance of this property to basically all forms of behavioural analysis discussed in this paper, we will be dealing mainly with compact language vectors hereafter.

5 Interacting DES

Let Σ be an alphabet vector with index set I . An *interacting discrete event system* over Σ is a pair $\mathcal{L} = (\mathbf{L}, K)$ where \mathbf{L} is a language vector in $\mathcal{L}(\Sigma)$ and K is a language in $\mathcal{L}(\Sigma)$. The language K will be referred to as the *interaction specification language* or simply the *interaction language* of the IDES \mathcal{L} . We will write L_i to denote the i th component of \mathcal{L} . The language generated by \mathcal{L} is given by

$$B_\Sigma(\mathcal{L}) = \|\mathbf{L} \cap K$$

Therefore, the IDES structure consists of a set of components, represented by the language vector \mathbf{L} , running concurrently, and a language K that synchronizes with the composite behaviour of these components.

Based on the setting of the IDES model it is possible to decompose any single process (flat) DES to an IDES structure that generates the same behaviour. This can be done by compensating the information lost in the projection operation, that is, by adding necessary information to the composite behaviour $B\mathbf{P}_\Sigma(\mathbf{L})$ such that the overall behaviour of the structure is equal to \mathbf{L} . It

is easy to see that, for any language L such a compensator depends on L (is a function of L) and must contain L . The set of Σ -compensators for L , denoted $\underline{C}_\Sigma(L)$, is defined as follows

$$\underline{C}_\Sigma(L) = \{K \in \mathcal{L}(\Sigma) \mid L = B\mathbf{P}_\Sigma(L) \cap K\}$$

The set $\underline{C}_\Sigma(L)$ is not empty as it contains L . The set $\underline{C}_\Sigma(L)$ is closed under union and intersection and hence has a supremal and infimal element. The infimal element of $\underline{C}_\Sigma(L)$ is L . We will write $\check{C}_\Sigma(L)$ to denote the infimal element of the set $\underline{C}_\Sigma(L)$ and $\hat{C}_\Sigma(L)$ to denote its supremal element.

Proposition 5.1

$$\hat{C}_\Sigma(L) = L \cup B\mathbf{P}_\Sigma(L)^c$$

□

It is easy to see that any language K such that $L \subseteq K \subseteq \hat{C}_\Sigma(L)$ is a Σ -compensator for L . In general a compensator K for L may be blocking in the sense that the intersection of K with $B\mathbf{P}_\Sigma(L)$ may produce blocking states. We write $\underline{C}_\Sigma^o(L)$ to denote the set of *nonblocking compensators* for L . Formally, the set $\underline{C}_\Sigma^o(L)$ is defined as follows:

$$\underline{C}_\Sigma^o(L) = \{K \in \underline{C}_\Sigma(L) \mid \overline{B_\Sigma(\mathbf{P}_\Sigma(L), K)} = B_\Sigma(\overline{\mathbf{P}_\Sigma(L)}, \overline{K})\}$$

Also, this set is not empty for any language L as it contains L itself. It is easy to verify that this set is closed under union and hence contains a supremal element. We will denote this supremal element by $\hat{C}_\Sigma^o(L)$ for a given language L . Therefore, it is always possible to generate an optimal non-blocking IDES model that generates the language L . This is based on the following result.

Proposition 5.2

$$\hat{C}_\Sigma^o(L) = L \cup \left[L\Sigma \cap \overline{[B\mathbf{P}_\Sigma(L)]^c} \right] \Sigma^*$$

□

Therefore, the supremal non-blocking compensator $\hat{C}_\Sigma^o(L)$ can be obtained by adding to L any string that extends a string in L without being a prefix of $B\mathbf{P}_\Sigma(L)$.

6 Abstract interactions

In many practical situations, component interactions do not affect the internal behaviour of the components but rather their external arrangement. For instance, in serial composition the system components run sequentially in a way that is independent of the internal structure of these components.

To capture such specifications, we define the *index equivalence relation*, \mathcal{E}_Σ , generated by Σ , as follows

$$(\sigma, \sigma') \in \mathcal{E}_\Sigma \iff (\forall i \in I) \sigma \in \Sigma_i \Leftrightarrow \sigma' \in \Sigma_i$$

That is, two events are equivalent with respect to \mathcal{E}_Σ if they are shared (triggered) by exactly the same set of components. Consequently, each coset of \mathcal{E}_Σ is associated with a subset of I where the corresponding components share exactly the events of the coset. The coset of \mathcal{E}_Σ that contains the event σ will be denoted $[\sigma]_\Sigma$.

The set of subsets of I that are associated with the cosets of the index equivalence relation, \mathcal{E}_Σ , will be denoted \mathcal{J}_Σ . Each element in the set \mathcal{J}_Σ can be viewed as an abstract event that corresponds to a transition made simultaneously and collectively by the corresponding set of components. Define the map $f_\Sigma : \alpha(\Sigma) \rightarrow \mathcal{J}_\Sigma$ as follows (recall that $\mathcal{J}_\Sigma \subseteq \text{Pwr}(I)$),

$$(\forall \sigma \in \Sigma) \quad f_\Sigma(\sigma) = \{i \in I \mid \sigma \in \Sigma_i\}$$

The map f_Σ associates every event in Σ with the set of components in the process space Σ that have to coordinate in triggering this event. The function f_Σ is extended over languages in the usual way.

Languages over the set \mathcal{J}_Σ can be viewed as a form of abstract behaviour that does not distinguish between the events in the system components while recognizing the boundaries of the system components and their synchronization constraints. Languages over \mathcal{J}_Σ will be referred to as *abstract layouts*. The corresponding languages over the alphabet set Σ will be referred to as *layouts*. Formally, a layout in the process space Σ is a language $K \in \mathcal{L}(\Sigma)$ that satisfies

$$(\forall (\sigma, \sigma') \in \ker f_\Sigma) (\forall u, v \in \Sigma^*) u\sigma v \in K \iff u\sigma'v \in K$$

It is easy to see that the above condition is equivalent to having $K = f_\Sigma^{-1}(f_\Sigma(K))$. The composition map $f_\Sigma^{-1} \circ f_\Sigma : \mathcal{L}(\Sigma) \rightarrow \mathcal{L}(\Sigma)$ will be referred to as the *natural abstraction map* and will be denoted \mathbf{F}_Σ .

The set of layouts over Σ will be denoted $\mathcal{Y}(\Sigma)$. Clearly, there is a bijective correspondence between the set of layouts and the set of languages over \mathcal{J}_Σ representing the set of abstract layouts. Therefore, we will not distinguish between these two sets as long as the interpretation is clear from the context.

Abstract interaction specifications (ideally) do not impose any restriction on the internal dynamics of the system components. If this is valid for all language vectors in the process space, then we say that this interaction specification is *universal*. Parallel composition ($K = \Sigma^*$) is an example of a universal interaction specification. Universality is achieved if the projection of the interaction specification is equal to the supremal language vector in the corresponding process space. That is, a specification K is universal if $\mathbf{P}_\Sigma(K) = \mathbf{P}_\Sigma(\Sigma^*)$.

Example 1 Let $\Sigma = \{\{a_i, b_i, c_i, d_i, x\} \mid i \in [1, 2, 3]\}$ be a process space. The index equivalence relation is $\mathcal{E}_\Sigma = \{\{a_1, b_1, c_1, d_1\}, \{a_2, b_2, c_2, d_2\}, \{a_3, b_3, c_3, d_3\}, \{x\}\}$. The corresponding abstract events will be denoted U ,

V , W and X respectively. The natural abstraction map here is defined as follows; $F_{\Sigma}(x) = \{x\}$ and

$$F_{\Sigma}(a_i) = F_{\Sigma}(b_i) = F_{\Sigma}(c_i) = F_{\Sigma}(d_i) = \{a_i, b_i, c_i, d_i\}$$

for $i \in [1, 2, 3]$. Now, consider a system consisting of three machines in this process space. The overall system is shown in Figure 1 for the layout $K = \{U, V, W, X\}^*$ which allows the components to run in parallel.

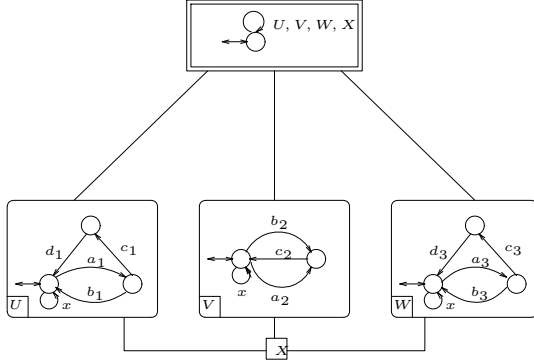


Figure 1: The IDES structure of three machines

Figure 2 shows another interaction setting where the first two machines run in parallel and are then followed by the third machine in a continuous cycle.

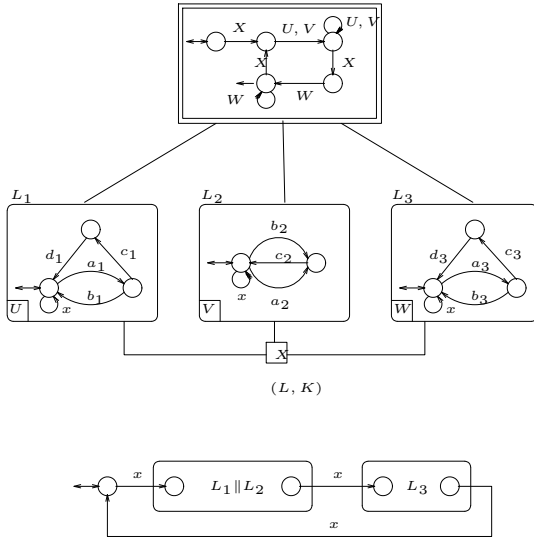


Figure 2: The three machines in another configuration

The overall system behaviour is outlined in the above figure. The detailed state machines for $L_1 || L_2$ and L_3 are omitted for simplicity. \square

Many standard language operations can be simulated using layouts as the corresponding interaction specifications. In the following, some standard binary operations and their corresponding interaction specifications are

presented for the process space $\Sigma = \{\Sigma_1, \Sigma_2\}$ consisting of two components where the corresponding abstract events are denoted $U (= \Sigma_1 - \Sigma_2)$, $X (= \Sigma_1 \cap \Sigma_2)$, $V (= \Sigma_2 - \Sigma_2)$.

Serial composition

In asynchronous environments serial composition can be simulated by the layout $K = U^*V^*$. Note that this layout is a universal interaction specification. For synchronous environments, the layout $K = U^*X^*V^*$ can simulate a form of synchronous composition where one process starts then synchronizes with a second process and then the second process continues and exits.

Parallel composition

Parallel composition corresponds to the least restrictive interaction (no restriction at all) between the system components. This interaction is represented simply by the synchronous product operation which can be simulated by the layout $K = \{U, X, V\}^*$.

Refinement

Language refinements can be expressed using the generalized language composition under certain assumptions. For instance, when shared events are used to initialize the extension (substitution) procedure while all other events are considered internal (not shared). This means that each event σ in Σ is mapped to a language in Δ^* , where $\Sigma \subseteq \Delta$, in the form σH , where $H \subseteq (\Delta - \Sigma)^*$. This form of refinement can be simulated by the layout $K = \{U, XV^*\}^*$.

In handshaking, on the other hand, shared events are used to initiate the extension (refinement) and they are also required to signal its termination. Generally, there are four disjoint sets of events; initiating events, terminating events, internal events of the calling subsystem and internal events of the called subsystem. Considering the same set of initiating and terminating events, handshaking refinements can be simulated by the layout $K = \{U, XV^*X\}^*$.

Interleaving

In interleaving interaction two or more systems are executed interchangeably based on certain time limits and priority schemes. In the simple setting of two systems with equal priorities, no timing constraints, and no shared events, interleaving can be represented by the (universal) layout $K = (UV)^*(U + \epsilon)$. Figure 3 shows this layout as well as the layout of some other standard interactions.

7 Multilevel interaction

The IDES model can be extended to provide direct representation of hierarchical multiprocess DES. This requires expanding the interaction specification from a language to a structure that matches the organization of

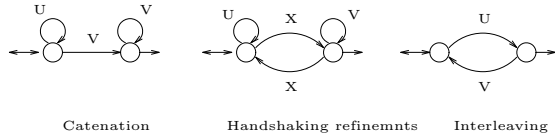


Figure 3: The Layouts of Some Standard Interactions

the systems. This interaction structure can be viewed as a more detailed representation of the interaction specification of the system, consisting of an ordered set of local specifications, each one of which targets a specified level of the system description.

Let Π and Σ be two alphabet vectors with index sets I and J respectively such that $\alpha(\Sigma) = \alpha(\Pi)$. We say that Π is a *cover* for Σ , written $\Sigma \preceq \Pi$, if

1. $(\forall i \in I)(\forall j \in J) \quad \Sigma_i \cap \Pi_j \neq \emptyset \implies \Sigma_i \subseteq \Pi_j$,
2. $(\forall i \in I)(\forall j, k \in J) \quad \Sigma_i \subseteq (\Pi_j \cap \Pi_k) \implies j = k$.

That is, every component in Σ is a subset of some unique component of Π and every component of Π is the union of a unique set of components in Σ . Hence, each $j \in J$ corresponds to a unique subset of I and, under this correspondence, the set J can be identified with a partition of I . Note that, in general, both Σ and Π may contain shared events. We will write $\Sigma \prec \Pi$ for the case when $\Sigma \preceq \Pi$ and $\Sigma \neq \Pi$. For example, consider the case where $\Sigma = \{(a, b, x), (c, x, y), (y, d)\}$ and $\Pi = \{(a, b, x), (c, d, x, y)\}$. Then, $\Sigma \prec \Pi$.

A formal model for multilevel interaction in hierarchical systems can be defined based on the above description. An N -level *interaction structure* for the process space Σ is defined as the tuple $(\underline{\Pi}, \underline{K})$, where the first element $\underline{\Pi}$ is a set of alphabet vectors $\{\Sigma_n \mid n \in [1 \dots N], \alpha(\Sigma_n) = \alpha(\Sigma)\}$, referred to as a *process space structure*, satisfying

$$(\forall n \in [1 \dots N-1]) \quad \Sigma \prec \Sigma_n \prec \Sigma_{n+1} \quad \text{and} \quad \Sigma_N = \{\Sigma\}$$

That is each alphabet vector in $\underline{\Pi}$ is a cover for any lower rank alphabet vector. The highest rank alphabet vector has one component, namely, the alphabet set Σ . A set of partitions for the index set I will be used as indices for the set $\underline{\Pi}$. This set of indices is denoted \underline{I}_Ψ and defined as follows,

$$\underline{I}_\Psi = \{I_n \in \mathcal{E}_I, n \in [1, N] \mid (\forall i \in [1, N-1]) I_i \prec I_{i+1}\}$$

where \mathcal{E}_I is the set of all partitions of the set I . Each element $I_n \in \underline{I}_\Psi$ serves as an index for the alphabet vector $\Sigma_n \in \underline{\Pi}$. Recall that each element in I_n is a subset of I and the set I_n is a partition of I . Also, the partition I_n is finer than the partition I_{n+1} for all $n \in [1 \dots N]$, and at the top level $I_N = \{I\}$.

The following convention will be used to identify the components in the set of alphabet vectors $\underline{\Pi}$. Each com-

ponent will have two subscripts. The first subscript indicates the abstraction level and the second one indicates the index of the component. For example, $\Sigma_{n,j}$ denotes the component j of the alphabet vector Σ_n at the n th level. Note that in this hierarchy, each alphabet component in a given level covers (that is, contains the alphabet of) a unique set of components at the next lower level. The subvector of Σ_{n-1} that is covered by the component $\Sigma_{n,j}$ at the next upper level will be denoted $\Sigma_{n,j}$.

The second element \underline{K} is a set of interaction language vectors $\{K_n \mid n \in [1 \dots N]\}$ satisfying

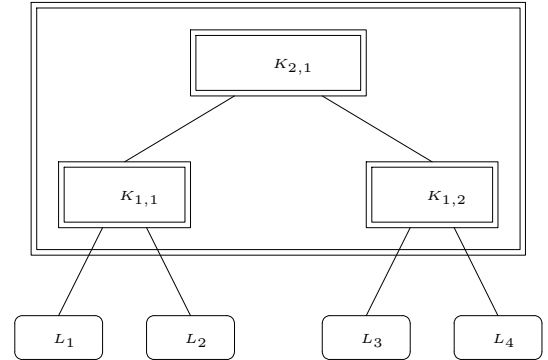
$$(\forall n \in [1 \dots N]) \quad K_n = \{K_{n,j} \subseteq \Sigma_{n,j}^* \mid j \in I_n\}$$

Here also two subscripts will be used to identify each interaction language in the set \underline{K} similar to the convention used above. So, $K_{n,j}$ is the component j of the language vector K_n at the n th level.

Example 2 Let $\Sigma = \{(a, x), (b, x, y), (c, y), (d, y)\}$ be a process space over the index set $I = \{1, 2, 3, 4\}$. Let $\Psi = (\underline{\Pi}, \underline{K})$ be a 2-level interaction structure over Σ defined as follows

1. $\underline{\Pi} = \{\Sigma_1, \Sigma_2\}$, with
 $\Sigma_1 = \{\{a, b, x, y\}, \{c, d, y\}\} = \{\Sigma_{1,A}, \Sigma_{1,B}\}$,
 $\Sigma_2 = \{\{a, b, c, d, x, y\}\} = \{\Sigma_{2,I}\}$
2. $\underline{K} = \{K_1, K_2\}$, with $K_1 = \{K_{1,A}, K_{1,B}\}$, and
 $K_2 = \{K_{2,I}\}$, where
 $K_{1,A} \subseteq \Sigma_{1,A}^*$, $K_{1,B} \subseteq \Sigma_{1,B}^*$, and $K_{2,I} \subseteq \Sigma_{2,I}^*$

The interaction structure Ψ applied to a language vector L is shown in the following figure.



Here we have $\underline{I}_\Psi = \{I_1, I_2\}$ where, $I_1 = \{\{1, 2\}, \{3, 4\}\}$ and $I_2 = \{\{1, 2, 3, 4\}\}$. In the above we write A for $\{1, 2\}$ and B for $\{2, 3\}$. \square

The composition operation B_Σ can now be extended to handle multiprocess systems with interaction structure specification. Let L be a language vector over Σ and let $\Psi = (\underline{\Pi}, \underline{K})$ be an interaction structure over Σ . The composition of L under Ψ , denoted $B_\Sigma(L, \Psi)$, is

defined through the following recursion. Let $\mathbf{L}_0 = \mathbf{L}$. For each $i \in [1 \dots N]$ define

$$\mathbf{L}_i = \{K_{i,J} \cap \parallel_{j \in J} L_{i-1,j} \mid J \in I_i\}$$

where $L_{i-1,j}$ is the component j of the language vector \mathbf{L}_{i-1} . This iteration will end up with the language vector \mathbf{L}_N which contains a single element, $L_{N,I}$, which is the result of the compound composition $B_{\Sigma}(\mathbf{L}, \Psi)$. That is,

$$B_{\Sigma}(\mathbf{L}, \Psi) = L_{N,I}$$

Clearly, $B_{\Sigma}(\mathbf{L}, \Psi)$ is the language generated by the language vector \mathbf{L} under the restriction of the interaction structure Ψ . Applying the above recursion on the last example we get,

$$\begin{aligned} \mathbf{L}_0 &= \{L_{0,i} \mid L_{0,i} = L_i, i \in [1, \dots, 4]\} = \mathbf{L}, \\ \mathbf{L}_1 &= \{L_{1,1}, L_{1,2}\}, \text{ where} \\ L_{1,1} &= (L_{0,1} \parallel L_{0,2}) \cap K_{1,A}, \quad L_{1,2} = (L_{0,3} \parallel L_{0,4}) \cap K_{1,B}, \\ \mathbf{L}_2 &= \{L_{2,1}\}, \text{ where} \\ L_{2,1} &= (L_{1,1} \parallel L_{1,2}) \cap K_{2,I}, \quad B_{\Sigma}(\mathbf{L}, \Psi) = L_{2,I} \end{aligned}$$

It is worthwhile to see if the effect of the interaction structure Ψ on a given alphabet vector can be simulated by an interaction language. Define the language $B_{\Sigma}(\mathbf{P}_{\Sigma}(\Sigma^*), \Psi)$ as the interaction language generated by interaction structure Ψ . This language may be referred to as the Ψ -interaction language and will be denoted K_{Ψ} . Based on its definition, the language K_{Ψ} can be obtained by composing the set \mathbf{K} recursively starting from the vector \mathbf{K}_1 in a way similar to the procedure described above for computing B_{Σ} . Applying this procedure to the last example we get

$$K_{\Psi} = (K_{1,A} \parallel K_{1,B}) \cap K_{2,I}$$

We claim that the language K_{Ψ} simulates the effect of the interaction structure Ψ on any given language vector in the process space Σ . That is, the restriction of any language vector \mathbf{L} in Σ to K_{Ψ} is equivalent to the restriction of \mathbf{L} to the structure Ψ . Let $\Sigma_i \subseteq \Sigma_j$ be two alphabet sets. We will write $P_{j/i} : \Sigma_j^* \rightarrow \Sigma_i^*$ for the natural projection from Σ_j to Σ_i . The inverse of this projection is denoted $P_{j/i}^{-1}$.

Lemma 7.1 Let $\Sigma_i \subseteq \Sigma_j \subseteq \Sigma_k$ be alphabet sets. Then,

$$P_{k/j}^{-1} \circ P_{j/i}^{-1} = P_{k/i}^{-1}$$

□

The following result confirms the claim above and shows that the restriction effect of any interaction structure on a given language can be simulated by an interaction language that depends only on the interaction structure.

Theorem 7.1 Let Ψ be an interaction structure in the process space Σ . Then

$$(\forall \mathbf{L} \in \mathcal{L}(\Sigma)) \quad B_{\Sigma}(\mathbf{L}, \Psi) = B_{\Sigma}(\mathbf{L}, K_{\Psi})$$

□

Based on the above result, an IDES model can be represented as a tuple $\mathcal{L} = (\mathbf{L}, \Psi)$ where Ψ is an interaction structure. The same system can be described by the *multilevel* IDES $\mathcal{L} = (\mathbf{L}, K_{\Psi})$, where K_{Ψ} is the interaction language equivalent to Ψ .

Layout languages were introduced earlier to represent abstract interaction specifications. The abstract characteristics of layouts can be extended to the multilevel interaction structure as follows. In a process space Σ with index set I , an N -level interaction structure $\Psi = (\mathbf{I}, \mathbf{K})$ is said to be a *layout structure* if

$$(\forall I_n \in \mathbf{I}_{\Psi})(\forall j \in I_n) \quad K_{n,j} \in \mathcal{Y}(\Sigma_{n,j})$$

That is, every interaction language in the set \mathbf{K} is a layout with respect to the corresponding process space. The following result shows that the interaction language generated by a layout structure is a layout.

Proposition 7.1

$$\Psi \text{ is a layout structure} \implies K_{\Psi} \text{ is a layout}$$

□

A multilevel system can be represented as a tree in which the components appear at the end leaves inside rounded boxes while interaction languages are shown at all other nodes inside double boxes. In this diagram each interaction language shows the interaction between the components that directly descend from its node.

Example 3 This example is a slightly modified version of the three machines system of example 1. In this example an extra shared event, y , is used to allow more flexible arrangements of the system. The system is arranged in order to satisfy the following two requirements;

- The second machine can be initiated only after event c_1 and must terminate before the event d_1
- The third machine works only after the first machine and must terminate before any of the two machines can start again.

This arrangement can be represented as a hierarchical layout structure as shown in the Figure 4. In this figure, the layout K_1 restricts the first two machines in accordance with the first specification. The second specification is handled at the second level using the upper level layout. Note that in the first level the abstract event Z refers to the event shared between the composition of the first two machines (under K_1) and the third machine, namely the event y .

Figure 5 shows the above system under the interaction language generated by the above interaction structure.

□

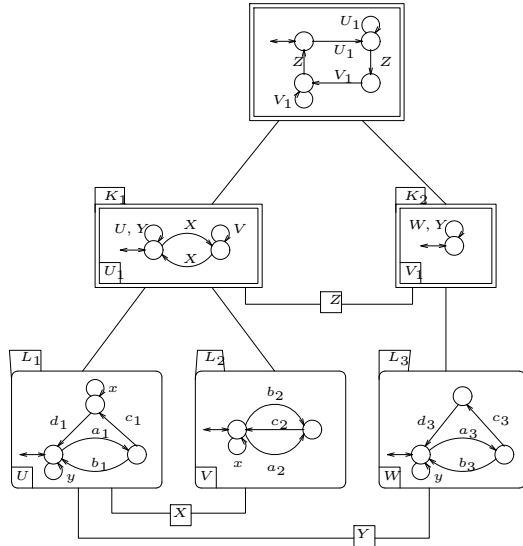


Figure 4: Three machines in a hierarchical structure

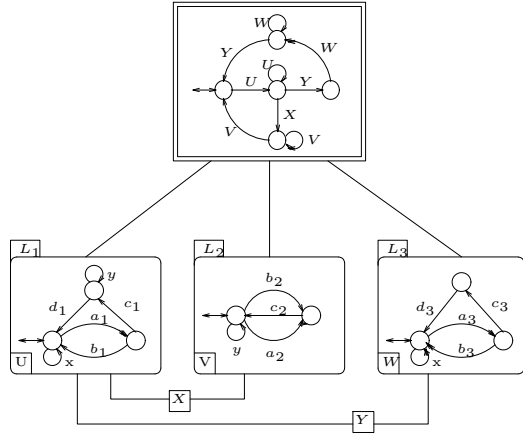


Figure 5: The IDES of the three machines system

Multilevel decomposition

Let $\underline{\Pi} = \{\Sigma_n \mid n \in [1 \dots N]\}$ be an N -level process space structure over the process space Σ and let S be a language in $\mathcal{L}(\Sigma)$. An interaction structure $\Psi = (\underline{\Pi}, \underline{K})$ is said to be a *compensation structure* for S if the composition of $P_{\Sigma}(S)$ under the interaction structure Ψ is equal to S , namely

$$S = B_{\Sigma}(P_{\Sigma}(S), \Psi)$$

Based on Theorem 7.1, Ψ is a compensation structure for S if and only if $K_{\Psi} \in \hat{C}_{\Sigma}(S)$. Let $\Psi_1 = (\underline{\Pi}, \underline{K}_1)$ and $\Psi_2 = (\underline{\Pi}, \underline{K}_2)$ be two interaction structures. The componentwise union of Ψ_1 and Ψ_2 is denoted $\Psi_1 \sqcup \Psi_2$ and is given as a structure $\Psi_1 = (\underline{\Pi}, \underline{K})$ where

$$\underline{K} = \{\underline{K}_{n1} \sqcup \underline{K}_{n2} \mid n \in [1 \dots N]\}$$

That is, the interaction vectors of \underline{K} are the componentwise union of the corresponding interaction vectors in \underline{K}_1 and \underline{K}_2 . Componentwise intersection of interaction structures is defined similarly. Note that in both operations the arguments and the output are structurally matched, namely, defined over the same process space structure $\underline{\Pi}$. It can be shown easily that the set of compensation structures for a given language S is closed under componentwise union and therefore has a supremal element. Clearly, if Ψ is the supremal compensation structure for S then $K_{\Psi} = \hat{C}_{\Sigma}(S)$.

For a process space Σ and N -level process space structure $\underline{\Pi}$ over Σ with $N > 1$, it is easy to see that the supremal compensation structure for a language $S \subseteq \Sigma^*$ is given by $(\underline{\Pi}, \underline{K})$ where,

$$\underline{K}_i = P_{\Sigma_i}(\Sigma^*) \quad i \in [1 \dots N - 1]$$

and $\underline{K}_N = \{\hat{C}_{\Sigma}(S)\}$. That is, the r th component of $\underline{K}_j \in \underline{K}$, with $1 \leq j < N$ is given as $K_{r,j} = \Sigma_{r,j}^*$. The supremality of this compensation structure is direct based on the associativity of the synchronous product operation.

References

- [1] S. Abdelwahed. *Interacting Discrete Event Systems: Modelling, Verification, and Supervisory Control*. PhD thesis, University of Toronto, 2002.
- [2] R. H. Campell and A. N. Habermann. The specification of process synchronization by path expressions. In *Lecture Notes in Computer Science*, volume 16, pages 89–102. Springer, Berlin, 1974.
- [3] R. Janicki and P. E. Lauer. *Specification and Analysis of Concurrent Systems, The COSY Approach*. EATCS Monographs on Theoretical Computer science. Springer, Berlin, 1992.
- [4] N. W. Keesmaat and H. C. M. Kleijn. Restrictions and representations of vector controlled concurrent system behaviours. *Theoretical Computer Science*, 179:61–102, 1997.
- [5] A. Mateescu, G. Rozenberg, and A. Salomaa. Shuffle on trajectories: Syntactic constraints. Technical Report 41, Turku Centre for Computer Science, 1996.
- [6] M. W. Shields. Adequate path expressions. In *Lecture Notes in Computer Science*, volume 70, pages 249–265. Springer, Berlin, 1979.
- [7] M. W. Shields. *Semantics of Parallelism: Non-Interleaving Representation of Behaviour*. Springer-Verlag, London, 1997.
- [8] W.M. Wonham. *Notes on Control of Discrete-Event Systems*. ECE Department, University of Toronto, revised 2002.07.01. www.control.utoronto.ca/people/profs/wonham.