Composition for Component-Based Modeling*

Gregor Gössler[1] and Joseph Sifakis[2]

[1] INRIA Rhône-Alpes, goessler@inrialpes.fr
[2] VERIMAG, sifakis@imag.fr

1 Introduction

Component-based engineering is of paramount importance for rigorous system design methodologies. It is founded on a paradigm which is common to all engineering disciplines: complex systems can be obtained by assembling components (building blocks). Components are usually characterized by abstractions that ignore implementation details and describe properties relevant to their composition e.g. transfer functions, interfaces. Composition is used to build complex components from simpler ones. It can be formalized as an operation that takes in components and their integration constraints. From these, it provides the description of a new, more complex component.

Component-based engineering is widely used in VLSI circuit design methodologies, supported by a large number of tools. Software and system component-based techniques have known significant development, especially due to the use of object technologies supported by languages such as C++, Java, and standards such as UML and CORBA. However, these techniques have not yet achieved the same level of maturity as has been the case for hardware. The main reason seems to be that software systems are immaterial and are not directly subject to the technological constraints of hardware, such as fine granularity and synchrony of execution. For software components, it is not as easy to establish a precise characterization of the service and functionality offered at their interface.

Existing component technologies encompass a restricted number of interaction types and execution models, for instance, interaction by method calls under asynchronous execution. We lack concepts and tools allowing integration of synchronous and asynchronous components, as well as different interaction mechanisms, such as communication via shared variables, signals, rendez-vous. This is essential for modern systems engineering, where applications are initially developed as systems of interacting components, from which implementations are derived as the result of a co-design analysis.

The development of a general theoretical framework for component-based engineering is one of the few grand challenges in information sciences and technologies. The lack of such a framework is the main obstacle to mastering the complexity of heterogeneous systems. It seriously limits the current state of the practice, as attested by the lack of development platforms consistently integrating design activities and the often prohibitive cost of validation.

* to appear in the proceedings of FMCO'02, held November 5–8, 2002, Leiden, the Netherlands.

The application of component-based design techniques raises two strongly related and hard problems.

First, the development of theory for building *complex heterogeneous systems*. Heterogeneity is in the different types of component interaction, such as strict (blocking) or non strict, data driven or event driven, atomic or non atomic and in the different execution models, such as synchronous or asynchronous.

Second, the development of theory for building systems which are *correct by construction*, especially with respect to essential and generic properties such as deadlock-freedom or progress.

In practical terms, this means that the theory supplies rules for reasoning on the structure of a system and for ensuring that such properties hold globally under some assumptions about its constituents e.g. components, connectors. Tractable correctness by construction results can provide significant guidance in the design process. Their lack leaves a posteriori verification of the designed system as the only means to ensure its correctness (with the well-known limitations).

In this paper, we propose a framework for component-based modeling that brings some answers to the above issues. The framework uses an abstract layered model of components. It integrates and simplifies results about modeling timed systems by using timed automata with dynamic priorities [5, 1].

A component is the superposition of three models: a behavioral model, an interaction model, and an execution model.

- Behavioral models describe the dynamic behavior of components.
- Interaction models describe architectural constraints on behavior. They are defined as a set of connectors and their properties. A connector is a maximal set of compatible component actions. The simultaneous occurrence of actions of a connector is an interaction.
- Execution models reduce non determinism resulting from parallel execution in the lower layers. They are used to coordinate the execution of threads so as to ensure properties related to the efficiency of computation, such as synchrony and scheduling.

An associative and commutative composition operator is defined on components, which preserves deadlock-freedom. The operator defines a three-layered component by composing separately the corresponding layers of its arguments. As a particular instance of the proposed framework, we consider components where behaviors are transition systems and both interaction and execution models are described by priority relations on actions.

Our framework differs from existing ones such as process algebras, semantic frameworks for synchronous languages [4, 11, 3, 17] and Statecharts [12], in two aspects.

First, it distinguishes clearly between three different and orthogonal aspects of systems modeling: behavior, interaction (architecture) and execution. This distinction, apart from its methodological interest, allows solving technical problems such as associativity of a unique and powerful composition operator. The proposed framework has concepts in common with Metropolis [2] and Ptolemy [16] where a similar separation of concerns is advocated.

Second, parallel composition preserves deadlock-freedom. That is, if the arguments can perform some action from any state then their product does so. This is due to the fact that we replace restriction or other mechanisms used to ensure strong synchronization between components, by dynamic priorities. Nevertheless, our composition is a partial operation: products must be *interaction safe*, that is, they do not violate strong synchronization assumptions. In that respect, our approach is has some similarity to [7].

The paper is organized as follows.

Section 2 discusses three requirements for composition in component-based modeling. The first is support for two main types of heterogeneity: heterogeneous interaction and heterogeneous execution. The second is that it provide results for ensuring correctness by construction for a few essential and generic system properties, such as deadlock-freedom. The third is the existence of a composition operator that allows abstraction and incremental description.

Section 3 presents a general notion of composition and its properties for components with two layers: behavior and interaction models. Interaction models relate concepts from architecture (connectors) to actions performed by components via the notion of interaction. Interaction models distinguish between complete and incomplete interactions. This distinction induces the concept of interaction safety for models, meaning that only complete or maximal interactions are possible. We show associativity and commutativity of the composition operator. The section ends with a few results on correctness by construction for interaction safety of models and deadlock-freedom.

Section 4 presents two examples illustrating the use of execution models. We assume that execution models can be described by priority orders. The first example shoes how synchronous execution can be enforced by a priority order on the interactions between reactive components. The order respects the causality flow relation between component actions. The second example shows how scheduling policies can be implemented by an execution model.

Section 5 presents concluding remarks about the presented framework.

2 Requirements for Composition

2.1 General

We consider a very simple and abstract concept of components that is sufficient for the purpose of the study. A component can perform actions from a vocabulary of actions. The behavior of a component describes the effect of its actions.

A system of interacting components is a set of components integrated through various mechanisms for coordinating their execution. We assume that the overall effect of integration on the components of a system is the restriction of their behavior and it can be abstractly described by integration constraints. The latter describe the environment of a component. A component's actions may be blocked until the environment offers actions satisfying these constraints.

We distinguish two types of integration constraints: interaction and execution constraints.

Interaction constraints characterize mechanisms used in architectures such as connectors, channels, synchronization primitives. Interactions are the result of composition between actions. In principle, all the actions of a component are "visible" from its environment. We do not consider any specific notion of interface.

Execution constraints restrict non determinism arising from concurrent execution, and ensure properties related to the efficiency of computation, such as synchronous execution and scheduling.

There exists a variety of formalisms proposing concepts for parallel execution of sequential entities, such as process algebras (CCS [19], CSP [13]), synchronous languages (Esterel, Lustre, Statecharts), hardware description languages (VHDL), system description languages (SystemC [20], Metropolis), and more general modeling languages (SDL [14], UML [10]). In our terminology, we use the term "component" to denote any executable description whose runs can be modeled as sequences of actions. Component actions can be composed to produce interactions. Tasks, processes, threads, functions, blocks of code can be considered as components provided they meet these requirements.

The purpose of this section is to present concept requirements for composition in component-based modeling and to discuss the adequacy of existing formalisms with respect to these requirements.

2.2 Heterogeneity

Heterogeneity of Interaction It is possible to classify existing interaction types according to the following criteria:

Interactions can be *atomic* or *non atomic*. For atomic interactions, the behavior change induced in the participating components cannot be altered through interference with other interactions. Process algebras and synchronous languages assume atomic interactions. In languages with buffered communication (SDL, UML) or in multi-threaded languages (Java), interactions are not atomic, in general. An interaction is initialized by sending a message or by calling a method, and between its initiating action and its termination, components non participating in the interaction can interfere.

Interactions can involve *strict* or *non strict* synchronization. For instance, atomic rendez-vous of CSP is an interaction with strict synchronization in the sense that it can only occur if all the participating actions can occur. Strict synchronization can introduce deadlocks in systems of interacting deadlock-free components, that is, components that can always offer an action. If a component persistently offers an action and its environment is unable to offer a set of actions matching the interaction constraints, then there is a risk of deadlock. In synchronous languages, interactions are atomic and synchronization is non strict in the sense that output actions can occur whether or not they match with some input. Nevertheless, for inputs to be triggered, a matching output is necessary.

Finally, interactions can be *binary* (point to point) or *n-ary* for $n \geqslant 3$. For instance, interactions in CCS and SDL are binary (point to point). The implementation of n-ary interactions by using binary interaction primitives is a non trivial problem.

Clearly, there exists no formalism supporting all these types of interaction.

Heterogeneity of Execution There exist two well-known execution paradigms.

Synchronous execution is typically adopted in hardware, in the synchronous languages, and in many time triggered architectures and protocols. It considers that a system run is a sequence of global steps. It assumes synchrony, meaning that the system's environment does not change during a step, or equivalently "that the system is infinitely faster than its environment". In each execution step, all the system components contribute by executing some "quantum" computation, defined through the use of appropriate mechanisms such as timing mechanisms (clocks, timers) or a notion of stable states. For instance, in synchronous languages, an execution step is a reaction to some external stimulus obtained by propagating the reactions of the components according to a causality flow relation. A component reaction is triggered by a change of its environment and eventually terminates at some stable state for this environment. The synchronous execution paradigm has built-in a very strong assumption of fairness: in each step all components execute a quantum computation defined using either quantitative or logical time.

The *asynchronous* execution paradigm does not adopt any notion of a global computation step in a system's execution. It is used in languages for the description of distributed systems such as SDL and UML, and programming languages such as Ada and Java. The lack of a built-in mechanism for sharing resources between components is often compensated by using scheduling. This paradigm is also common to all execution platforms supporting multiple threads, tasks, etc.

Currently, there is no framework encompassing the diversity of interaction and execution models. Figure 1 classifies different system description languages in a three-dimensional space with coordinates corresponding to execution (synchronous/asynchronous) and to interaction: atomic/non atomic and strict/non-strict. It is worth noting that synchronous languages use non strict and atomic interactions. This choice seems appropriate for synchronous execution. On the contrary, for asynchronous execution there is no language using this kind of interaction.

2.3 Correctness by Construction

It is desirable that frameworks for component-based modeling provide results for establishing correctness by construction for at least a few common and generic properties such as deadlock-freedom or stronger progress properties. In practical

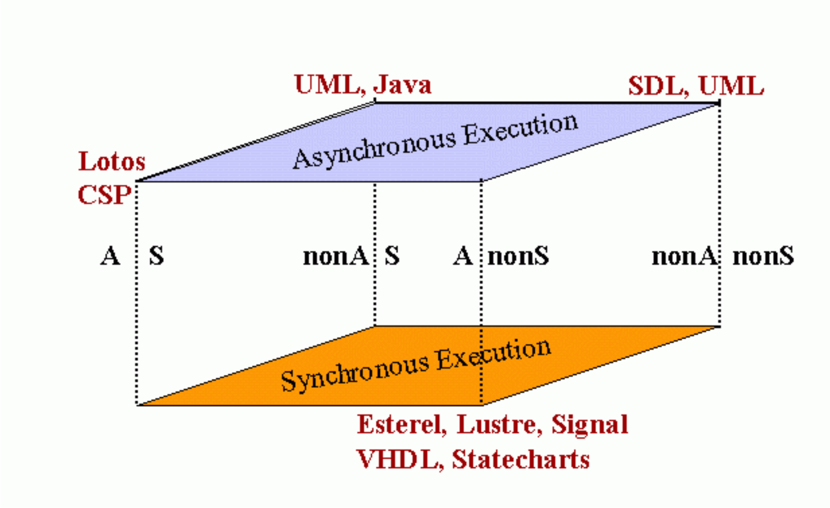

Fig. 1: About composition: heterogeneity. A: atomic, S: strict interaction.

terms, this implies the existence of inference rules for deriving system and component properties from properties of lower-level components. In principle, two types of rules are needed for establishing correctness by construction.

Composability rules allow to infer that, under some conditions, a component will meet a given property after integration. These rules are essential for preserving across integration previously established component properties. For instance, to guarantee that a deadlock-free component (a component that has no internal deadlocks) will remain deadlock-free after integration. Composability is essential for incremental system construction as it allows building large systems without disturbing the behavior of their components. It simply means stability of established component properties when the environment changes by adding or removing components. Property instability phenomena are currently poorly understood e.g. feature interaction in telecommunications, or non composability of scheduling algorithms. Results in composability are badly needed.

Compositionality rules allow to infer a system's properties from its components' properties. There exists a rich body of literature for establishing correctness through compositional reasoning [15, 9, 8]. However, most of the existing results deal with the preservation of safety properties.

2.4 Abstraction and Incrementality

A basic assumption of component-based engineering is that components are characterized by some external specification that abstracts out internal details. However, it is often necessary to modify the components according to the context of their use, at the risk of altering their behavior. Such modifications may be necessary to adapt components to a particular type of composition. For instance, to model non strict synchronization using strict synchronization, a common transformation consists in modifying both the action vocabularies (interfaces) and the behavior of components by adding for each action a of the interface a "complementary" $\bar{a}$ action that will be executed from all the states from which a is not possible. To model strict synchronization using non strict synchronization, similar modifications are necessary (see for instance Milner's SCCS [18]).

We currently lack sufficiently powerful and abstract composition operators encompassing different kinds of interaction.

Another important requirement for composition is incrementality of description. Consider systems consisting of sets of interacting components, the interaction being represented as usual, by connectors or architectural constraints of any kind. Incrementality means that such systems can be constructed by adding or removing components and that the result of the construction is independent of the order of integration. Associative and commutative composition operators allow incrementality.

Existing theoretical frameworks such as CCS, CSP, SCCS, use parallel composition operators that are associative and commutative. Nevertheless, these operators are not powerful enough. They need to be combined with other operators such as hiding, restriction, and renaming in system descriptions. The lack of a single operator destroys incrementality of description. For instance, some notations use hiding or restriction to enforce interaction between the components of a system. If the system changes by adding a new component, then some hiding or restriction operators should be removed before integrating the new component.

Graphical formalisms used in modeling tools such as Statecharts or UML do not allow incremental description as their semantics are not compositional. They are defined by functions associating with a description its meaning, as a global transition system (state machine), i.e., they implicitly use n-ary composition operators (n is equal to the number of the composed components).

It is always easy to define commutative composition, even in the case of asymmetric interactions. On the contrary, the definition of a single associative and commutative composition operator which is expressive and abstract enough to support heterogeneous integration remains a grand challenge.

3 Composition

We present an abstract modeling framework based on a unique binary associative and commutative composition operator.

Composition operators should allow description of systems built from components that interact by respecting constraints of an interaction model. The latter characterizes a system architecture as a set of connectors and their properties.

Given a set of components, composition operations allow to construct new components. We consider that the meaning of composition operations is defined by connectors. Roughly speaking, connectors relate actions of different components and can be abstractly represented as tuples or sets of actions. The related actions can form interactions (composite actions) when some conditions are met. The conditions define the meaning of the connector and say when and how the interaction can take place depending on the occurrence of the related actions. For instance, interactions can be asymmetric or symmetric. Asymmetric interactions have an initiator (cause) which is a particular action whose occurrence can trigger the other related actions. In symmetric interactions all the related actions play the same role.

The proposed composition operator differs from existing ones in automata theory and process algebras in the following.

- First, it preserves deadlock-freedom. This is not the case in general, for existing composition operators except in very specific cases. For instance, when from any state of the behavioral model any action offered by the environment can be accepted.
- Second, deadlock-freedom preservation is due to systematic interleaving of all the actions of the composed components, combined with the use of priority rules. The latter give preference to synchronization over interleaving. In existing formalisms allowing action interleaving in the product such as CCS and SCCS, restriction operators are used instead of priorities to prevent occurrence of interleaving actions. For instance, if a and $\bar{a}$ are two synchronizing actions in CCS, their synchronization gives an invisible action $\tau = a \mid \bar{a}$. The interleaving actions a and $\bar{a}$ are removed from the product system by using restriction. This may introduce deadlocks at product states from which no matching actions are offered. Priority rules implement a kind of dynamic restriction and lead to a concept of "flexible" composition.

3.1 Interaction Models and Their Properties

Consider a set of components with disjoint vocabularies of actions A_i for $i \in K$, K a set of indices. We put $A = \bigcup_{i \in K} A_i$.

A *connector* c is a non empty subset of A such that $\forall i \in K \, . \, |A_i \cap c| \leqslant 1$. A connector defines a maximally compatible set of interacting actions. For the sake of generality, our definition accepts singleton connectors. The use of the connector $\{a\}$ in a description is interpreted as the fact that action a cannot be involved in interactions with other actions.

Given a connector c, an *interaction* α of c is any term of the form $\alpha = a_1 \mid \ldots \mid a_n$ such that $\{a_1, \ldots, a_n\} \subseteq c$. We assume that $\mid$ is a binary associative and commutative operator. It is used to denote some abstract and partial action composition operation. The interaction $a_1 \mid \ldots \mid a_n$ is the result of the simultaneous occurrence of the actions $a_1, \ldots, a_n$. When α and α' are interactions we write $\alpha \mid \alpha'$ to denote the interaction resulting from their composition (if its is defined).

Notice that if $\alpha = a_1 \mid \ldots \mid a_n$ is an interaction then any term corresponding to a sub-set of $\{a_1, \ldots, a_n\}$ is an interaction. By analogy, we say that α' is a sub-interaction of α if $\alpha = \alpha' \shortmid \alpha''$ for some interaction α''. Clearly, actions are minimal interactions.

The set of the interactions of a connector $c = \{a_1, \ldots, a_n\}$, denoted by $I(c)$, consists of all the interactions corresponding to sub-sets of c (all the sub-interactions of c). We extend the notation to sets of connectors. If C is a set of connectors then $I(C)$ is the set of its interactions. Clearly for C_1, C_2 sets of connectors, $I(C_1 \cup C_2) = I(C_1) \cup I(C_2)$.

Definition 1 (Interaction model). *The* interaction model *of a system composed of a set of components K with disjoint vocabularies of actions A_i for $i \in K$, is defined by*

- *the vocabulary of actions $A = \bigcup_{i \in K} A_i$;*
- *the set of its connectors C such that $\bigcup_{c \in C} c = A$, and if $c \in C$ then there exists no $c' \in C$ and $c \subset c'$. That is, C contains only maximal connectors;*
- *the set of the complete interactions $I(C)^+ \subseteq I(C)$, such that $\forall b, b' \in I(C)$, $b \in I(C)^+$ and $b \subseteq b'$ implies $b' \in I(C)^+$. We denote by $I(C)^-$ the set of the incomplete (non complete) interactions.*

Notice that all actions appear in some connector. The requirement that C contains only maximal sets ensures bijective correspondence between the set of connectors C and the corresponding set of interactions $I(C)$. Given $I(C)$, the corresponding set of connectors is uniquely defined and is C. To simplify notation, we write IC instead of $I(C)$.

The distinction complete/incomplete is essential for building correct models. As models are built incrementally, interactions are obtained by composing actions. It is often necessary to express the constraint that some interactions of a sub-system are not interactions of the system. This is typically the case for binary strict synchronization (rendez-vous). For example, *send* and *receive* should be considered as incomplete actions but *send*$\shortmid$*receive* as complete. The occurrence of *send* or *receive* alone in a system model is an error because it violates the assumption about strict synchronization made by the designer.

Complete interactions can occur in a system when all the involved components are able to perform the corresponding actions. The distinction between complete/incomplete encompasses many other distinctions such as input/output, internal/external, observable/controllable used in different formalisms. It is in our opinion, the most relevant concerning the ability of components to interact. Clearly, internal component actions should be considered as complete because they can be performed by components independently of the state of their environment. In some formalisms, output actions are complete (synchronous languages, asynchronous buffered communication). In some others, with strict synchronization rules, all actions participating in interactions are incomplete. In that case, it is necessary to specify which interactions are complete. For instance, if $a_1 \shortmid a_2 \shortmid a_3$ is complete and no sub-interaction is complete, this means that a strong synchronization between a_1, a_2, a_3 is required.

A requirement about complete interactions is closedness for containment that is, if α is a complete interaction then any interaction containing it, is complete.

This requirement follows from the assumption that the occurrence of complete interactions cannot be prevented by the environment.

Very often it is sufficient to consider that the interactions of IC^+ are defined from a given set of complete actions $A^+ \subseteq A$. That is, IC^+ consists of all the interactions of IC where at least one complete action (element of A^+) is involved. In the example of figure 2, we give sets of connectors and complete actions to define interaction models. By convention, bullets represent incomplete actions and triangles complete actions. In the partially ordered set of the interactions, full nodes denote complete interactions. The interaction between *put* and *get* represented by the interaction $put|get$ is a rendez-vous meaning that synchronization is blocking for both actions. The interaction between *out* and *in* is asymmetric as *out* can occur alone even if *in* is not possible. Nevertheless, the occurrence of *in* requires the occurrence of *out*. The interactions between *out*, in_1 and in_2 are asymmetric. The output *out* can occur alone or in synchronization with any of the inputs in_1, in_2.

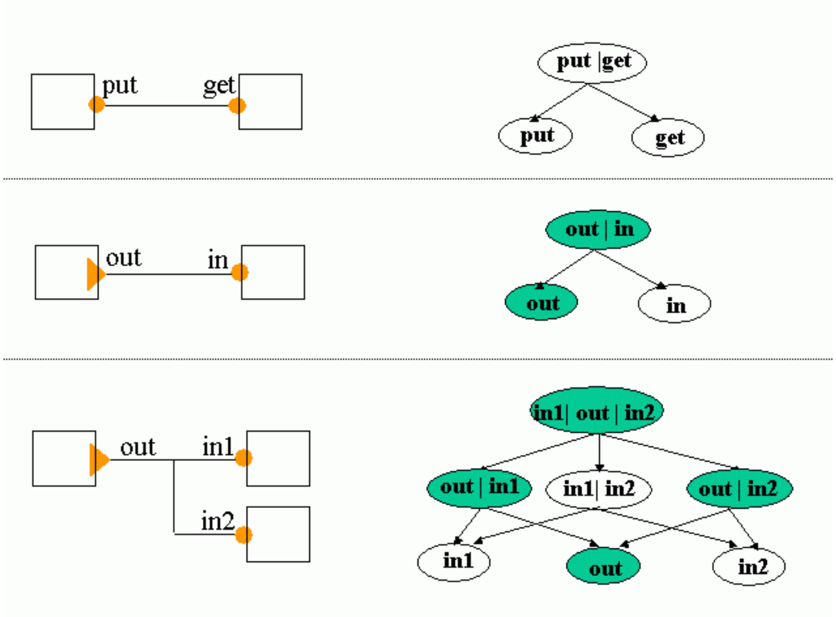

Fig. 2: Flexible composition: interaction structure.

In general, completeness of interactions need not be the consequence of the completeness of some action. For instance, consider a connector $\{a_1, a_2, a_3, a_4\}$ and suppose that the set of the minimal complete interactions of $I\{a_1, a_2, a_3, a_4\}$ is $a_1|a_2$ and $a_3|a_4$. That is, the actions a_1, a_2, a_3, a_4 are incomplete and only interactions containing $a_1|a_2$ or $a_3|a_4$ are complete. This specification requires strict synchronization of at least one of the two pairs (a_1, a_2), (a_3, a_4).

3.2 Incremental Description of Interaction Models

Consider the interaction model $IM = (IC, IC^+)$ of a set of interacting components K with disjoint vocabularies of actions A_i for $i \in K$. IC and IC^+ denote the sets of interactions and complete interactions, respectively on the vocabulary of actions $A = \bigcup_{i \in K} A_i$.

For given $K' \subseteq K$ the interaction model $IM[K']$ of the set of interacting components K' is defined as follows:

- $A[K'] = \bigcup_{i \in K'} A_i$ is the vocabulary of actions of $IM[K']$;
- $C[K'] = \{c' \mid \exists c \in C \,.\, c' = c \cap A[K'] \wedge \nexists c'' \in C \,.\, c' \subsetneq c'' \cap A[K']\}$ is the set of the connectors of $IM[K']$;
- $IM[K'] = (IC[K'], IC[K']^+)$ is the interaction model of $IM[K']$ where $IC[K']$ is the set of the interactions of $C[K']$ and $IC[K']^+ = IC[K'] \cap IC^+$.

Definition 2. *Given a family of disjoint sets of components $K_1, \ldots, K_n$ subsets of K, denote by $C[K_1, \ldots, K_n]$ the set of the connectors having at least one action in each set, that is, $C[K_1, \ldots, K_n] = \{c = c_1 \cup \cdots \cup c_n \mid \forall i \in [1, n] \,.\, c_i \in C[K_i]\}$.*

Clearly, $C[K_1, \ldots, K_n]$ is the set of the connectors of $IM[K_1 \cup \cdots \cup K_n]$ which are not connectors of any $IM[K']$ for any subset K' of at most $n - 1$ elements from $\{K_1, \ldots, K_n\}$.

Proposition 1. *Given K_1, K_2 two disjoint subsets of K.*

$$\begin{aligned}
IC[K_1 \cup K_2] &= IC[K_1] \cup IC[K_2] \cup IC[K_1, K_2] \\
IC[K_1 \cup K_2]^+ &= IC[K_1]^+ \cup IC[K_2]^+ \cup IC[K_1, K_2]^+ \\
IM[K_1 \cup K_2] &= (IC[K_1 \cup K_2], IC[K_1 \cup K_2]^+) \\
&= IM[K_1] \cup IM[K_2] \cup IM[K_1, K_2]
\end{aligned}$$

where $IC[K_1, K_2]^+ = IC[K_1, K_2] \cap IC^+$.

Proof. The first equality comes from the fact that $C[K_1] \cup C[K_2] \cup C[K_1, K_2]$ contains all the connectors of $C[K_1 \cup K_2]$ and other connectors that are not maximal. By definition, IC contains all the sub-sets of C. Thus, $IC[K_1 \cup K_2] = I(C[K_1] \cup C[K_2] \cup C[K_1, K_2]) = IC[K_1] \cup IC[K_2] \cup IC[K_1, K_2]$. ∎

Remark 1. The second equality says that the same interaction cannot be complete in an interaction model $IM[K_1]$ and incomplete in $IM[K_2]$, for $K_1, K_2 \subseteq K$.

This proposition provides a basis for computing the interaction model $IM[K_1 \cup K_2]$ from the interaction models $IM[K_1]$ and $IM[K_2]$ and from the interaction model of the connectors relating components of K_1 and components of K_2.

Property 1. For K_1, K_2, K_3 three disjoint subsets of K,

$$IC[K_1 \cup K_2, K_3] = IC[K_1, K_3] \cup IC[K_2, K_3] \cup IC[K_1, K_2, K_3]$$
$$IM[K_1 \cup K_2, K_3] = IM[K_1, K_3] \cup IM[K_2, K_3] \cup IM[K_1, K_2, K_3]$$

Proof. The first equality comes from the fact that $C[K_1, K_3] \cup C[K_2, K_3] \cup C[K_1, K_2, K_3]$ contains all the connectors of $C[K_1 \cup K_2, K_3]$ and in addition, other connectors that are not maximal. By definition, IC contains all the subsets of C. Thus, $IC[K_1 \cup K_2, K_3] = I(C[K_1, K_3] \cup C[K_2, K_3] \cup C[K_1, K_2, K_3])$ from which we get the result by distributivity of I over union. ■

This property allows computing the connectors and thus the interactions between $IM[K_1 \cup K_2]$ and $IM[K_3]$ in terms of the interactions between $IM[K_1]$, $IM[K_2]$, and $IM[K_3]$.

By using this property, we get the following expansion formula:

Proposition 2 (Expansion formula).

$$\begin{aligned} IM[K_1 \cup K_2 \cup K_3] = & IM[K_1] \cup IM[K_2] \cup IM[K_3] \cup IM[K_1, K_2] \\ & \cup IM[K_1, K_3] \cup IM[K_2, K_3] \cup IM[K_1, K_2, K_3] \,. \end{aligned}$$

3.3 Composition Semantics and Properties

We consider that a system S is a pair $S = (B, IM)$ where B is the behavior of S and IM is its interaction model. As in the previous section, IM is the interaction model of a set of interacting components K with disjoint action vocabularies A_i, $i \in K$.

For given $K' \subseteq K$, we denote by $S[K']$ the sub-system of S consisting of components of K', $S[K'] = (B[K'], IM[K'])$, where $IM[K']$ is defined as before.

We define a composition operator $\|$ allowing to obtain for disjoint sub-sets K_1, K_2 of K, the system $S[K_1 \cup K_2]$ as the composition of the sub-systems $S[K_1]$, $S[K_2]$ for given interaction model $IM[K_1, K_2]$ connecting the two sub-systems. The operator composes separately the behavior and interaction models of the sub-systems.

Definition 3. *The composition of two systems $S[K_1]$ and $S[K_2]$ is the system $S[K_1 \cup K_2] = (B[K_1], IM[K_1]) \| (B[K_2], IM[K_2]) = (B[K_1] \times B[K_2], IM[K_1] \cup IM[K_2] \cup IM[K_1, K_2])$ where $\times$ is a binary associative behavior composition operator such that $B[K_1] \times B[K_2] = B[K_1 \cup K_2]$.*

Due to proposition 1 we have $(B[K_1], IM[K_1]) \| (B[K_2], IM[K_2]) = (B[K_1 \cup K_2], IM[K_1 \cup K_2])$, which means that composition of sub-systems gives the system corresponding to the union of their components.

Notice that under these assumptions composition is associative:

$$\begin{aligned} & \big((B[K_1], IM[K_1]) \| (B[K_2], IM[K_2])\big) \| (B[K_3], IM[K_3]) = \\ & = (B[K_1 \cup K_2], IM[K_1 \cup K_2]) \| (B[K_3], IM[K_3]) \\ & = (B[K_1] \times B[K_2] \times B[K_3], IM[K_1 \cup K_2] \cup IM[K_3] \cup IM[K_1 \cup K_2, K_3]) \\ & = (B[K_1 \cup K_2 \cup K_3], IM[K_1 \cup K_2 \cup K_3]) \end{aligned}$$

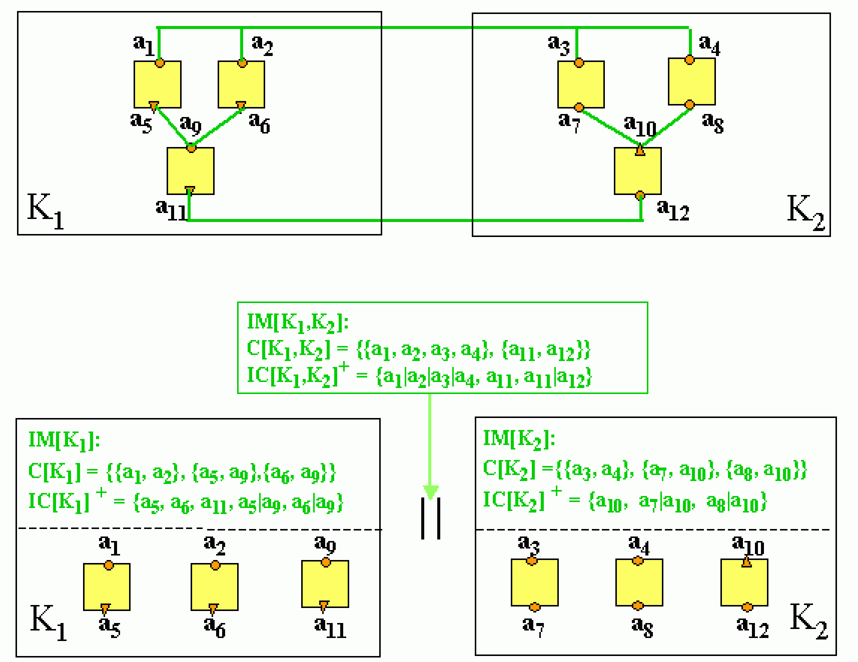

Fig. 3: The composition principle.

by application of proposition 2.

Transition Systems with Priorities As a rule, interaction models constrain the behaviors of integrated components. We consider the particular case where interactions are atomic, component behaviors are transition systems, and the constraints are modeled as priority orders on interactions. Transition systems with dynamic priorities have already been studied and used to model timed systems. The interested reader can refer to [6, 1].

Definition 4 (Transition system). *A* transition system *B is a triple $(Q, I(A), \rightarrow)$ where Q is a set of states, $I(A)$ is a set of interactions on the action vocabulary A, and $\rightarrow \subseteq Q \times I(A) \times Q$ is a transition relation.*

As usual, we write $q_1 \stackrel{\alpha}{\rightarrow} q_2$ instead of $(q_1, \alpha, q_2) \in \rightarrow$.

Definition 5 (Transition system with priorities). *A* transition system with priorities *is a pair $(B, \prec)$ where B is a transition system with set of interactions $I(A)$, and $\prec$ is a* priority order, *that is, a strict partial order on $I(A)$.*

Semantics: A transition system with priorities represents a transition system: if $B = (Q, I(A), \rightarrow)$, then $(B, \prec)$ represents the transition system $B' =$

$(Q, I(A), \rightarrow')$ such that $q_1 \xrightarrow{\alpha} {'}q_2$ if $q_1 \xrightarrow{\alpha} q_2$ and there exists no α' and q_3 such that $\alpha \prec \alpha'$ and $q_1 \xrightarrow{\alpha'} q_3$.

Definition 6 **($\oplus$).** *The sum $\prec^1 \oplus \prec^2$ of two priority orders $\prec^1$, $\prec^2$ is the least priority order (if it exists) such that $\prec^1 \cup \prec^2 \subseteq \prec^1 \oplus \prec^2$.*

Lemma 1. *$\oplus$ is a (partial) associative and commutative operator.*

Definition 7 **($\|$).** *Consider a system $S[K]$ with interaction model $IM[K] = (IC[K], IC[K]^+)$. Let $S[K_1] = (B[K_1], \prec^1)$ and $S[K_2] = (B[K_2], \prec^2)$ with disjoint K_1 and K_2 be two sub-systems of $S[K]$ such that their priority orders do not allow domination of complete interactions by incomplete ones, that is for all $\alpha_1 \in IC[K]^+$ and $\alpha_2 \in IC[K]^-$, $\neg(\alpha_1 \prec \alpha_2)$.*

The composition operator $\|$ is defined as follows. If $B_i = (Q_i, IC[K_i], \rightarrow_i)$ for $i = 1, 2$, then $S[K_1] \| S[K_2] = (B_1 \times B_2, \prec^1 \oplus \prec^2 \oplus \prec^{12})$, where

$B_1 \times B_2 = (Q_1 \times Q_2, IC[K_1 \cup K_2], \rightarrow_{12})$ with

$$q_1 \xrightarrow{\alpha}_1 q_1' \text{ implies } (q_1, q_2) \xrightarrow{\alpha}_{12} (q_1', q_2)$$

$$q_2 \xrightarrow{\alpha}_2 q_2' \text{ implies } (q_1, q_2) \xrightarrow{\alpha}_{12} (q_1, q_2')$$

$$q_1 \xrightarrow{\alpha_1}_1 q_1' \text{ and } q_2 \xrightarrow{\alpha_2}_2 q_2' \text{ implies } (q_1, q_2) \xrightarrow{\alpha_1 \alpha_2}_{12} (q_1', q_2') \text{ if } \alpha_1 | \alpha_2 \in IC[K_1 \cup K_2].$$

$\prec^{12}$ is the minimal priority order on $IC[K_1 \cup K_2]$ such that

- *$\alpha_1 \prec^{12} \alpha_1 | \alpha_2$ for $\alpha_1 | \alpha_2 \in IC[K_1, K_2]$ (maximal progress priority rule);*
- *$\alpha_1 \prec^{12} \alpha_2$ for $\alpha_1 \in IC[K_1 \cup K_2]^{--}$ and $\alpha_2 \in IC[K_1 \cup K_2]^+$ (completeness priority rule), where $IC[K_1 \cup K_2]^{--}$ denotes the elements of $IC[K_1 \cup K_2]^-$ that are non-maximal in $IC[K_1 \cup K_2]$.*

The first priority rule favors the largest interaction. The second ensures correctness of the model. It prevents the occurrence of incomplete interactions if they are not maximal. The occurrence of such interactions in a model is a modeling error. If a component can perform a complete action, all non maximal interactions of the other components are prevented. By executing complete actions the components may reach states from which a maximal incomplete interaction is possible.

Proposition 3. *$\|$ is a total, commutative and associative operator.*

Proof. Total operator: prove that for $K_1 \cap K_2 = \emptyset$, $\prec^1 \oplus \prec^2 \oplus \prec^{12}$ is a priority order, that is, the transitive closure of the union of $\prec^1$, $\prec^2$, and $\prec^{12}$ does not have any circuits.

The maximal progress priority rule defines a priority order identical to the set inclusion partial order, and is thus circuit-free.

The completeness priority rule relates incomplete and complete interactions and is circuit-free, too. The only source of a priority circuit could be the existence of interactions $\alpha_1, \alpha_2, \alpha_3 \in IC[K_1 \cup K_2]$ such that $\alpha_1 = \alpha_2 | \alpha_3$, $\alpha_1 \in IC[K_1 \cup$

$K_2]^{--}$, and $\alpha_2 \in IC[K_1 \cup K_2]^{+}$. This is impossible due to the monotonicity requirement of definition 1.

Associativity:

$$\begin{aligned}
&\big((B[K_1],\ \prec^1)\|(B[K_2],\ \prec^2)\big)\|(B[K_3],\ \prec^3) = \\
&= (B[K_1 \cup K_2],\ \prec^1 \oplus \prec^2 \oplus \prec^{12})\|(B[K_3],\ \prec^3) \\
&= (B[K_1 \cup K_2 \cup K_3],\ \prec^1 \oplus \prec^2 \oplus \prec^{12} \oplus \prec^3 \oplus \prec^{[12],3})
\end{aligned}$$

where $\prec^{[12],3}$ is the least priority order defined by

- $\alpha_1 \prec^{[12],3} \alpha_1 | \alpha_2$ for $\alpha_1 | \alpha_2 \in IC[K_1 \cup K_2, K_3]$, and
- $\alpha_1 \prec^{[12],3} \alpha_2$ for $\alpha_1 \in IC[K_1 \cup K_2 \cup K_3]^{--}$ and $\alpha_2 \in IC[K_1 \cup K_2 \cup K_3]^{+}$.

It can be shown that the order $\prec = \prec^{12} \oplus \prec^{[12],3}$ is the one defined by

- $\alpha_1 \prec \alpha_1 | \alpha_2$ for $\alpha_1 | \alpha_2 \in IC[K_1, K_2] \cup IC[K_1, K_3] \cup IC[K_2, K_3] \cup IC[K_1, K_2, K_3]$, and
- $\alpha_1 \prec \alpha_2$ for $\alpha_1 \in IC[K_1 \cup K_2 \cup K_3]^{--}$ and $\alpha_2 \in IC[K_1 \cup K_2 \cup K_3]^{+}$.

So the resulting priority order is the same independently of the order of composition. ■

Example 1. Consider the system consisting of a producer and a consumer. The components interact by rendez-vous. The actions *put* and *get* are incomplete. We assume that the actions *prod* and *cons* are internal and thus complete. Figure 4 gives the interaction model corresponding to these assumptions. The product system consists of the product transition system and the priority order defined from the interaction model. The priority order removes all incomplete actions (crossed transitions).

3.4 Correctness by Construction

We present results allowing to check correctness of the models with respect to two properties: interaction safety and deadlock-freedom.

Interaction Safety of the Model As explained in section 3.1, the distinction between complete and incomplete interactions is essential for building correct models. In existing formalisms, undesirable incomplete interactions are pruned out by applying restriction operators to the model obtained as the product of components [19]. In our approach, we replace restriction by priorities. This allows deadlock-freedom preservation: if an interaction is prevented from occurring, then some interaction of higher priority takes over. Nevertheless, it is necessary to check that our "flexible" composition operator does not allow illegal incomplete actions in a system model. For this we induce a notion of correctness called *interaction safety.*

Interaction safety is a property that must be satisfied by system models at any stage of integration. Notice however, that legality of incomplete interactions

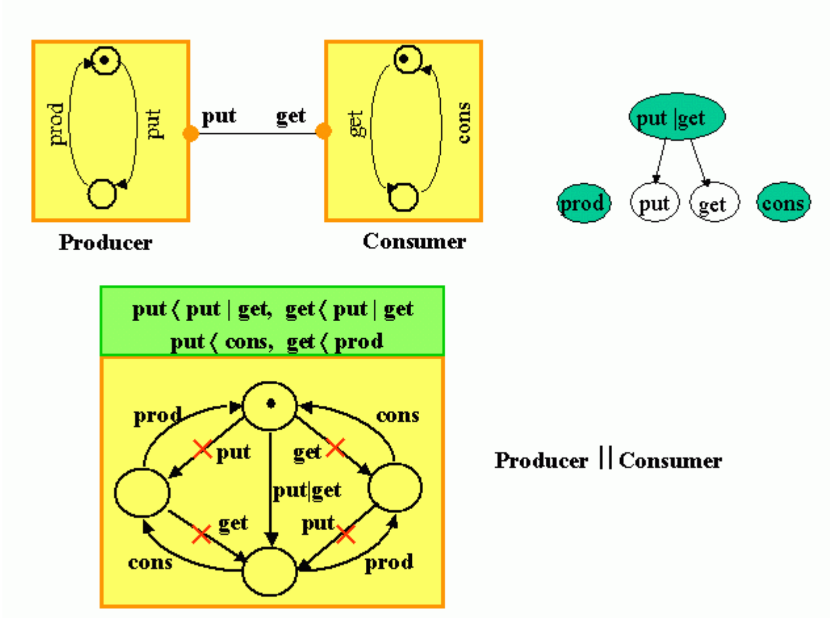

Fig. 4: Composition: producer/consumer.

depends on the set of integrated components. Sub-systems of a given system may perform incomplete interactions that are not legal interactions of the system. For instance, consider a system consisting of three components with a connector $\{a_1, a_2, a_3\}$ such that all its interactions are incomplete. The interaction $a_1|a_2$ is legal in the sub-system consisting of the first two components while it is illegal in the system. In the latter, $a_1 | a_2$ is incomplete and non maximal. It must synchronize with a_3 to produce the maximal incomplete interaction $a_1|a_2|a_3$.

For a given system, only complete and maximal incomplete interactions are considered as legal.

Definition 8 (Interaction safety). *Given an interaction model $IM = (IC, IC^+)$, define the priority order $\prec$ on incomplete interactions such that $\alpha_1 \prec \alpha_2$ if $\alpha_1 \in IC^{--}$ and $\alpha_2 \in IC^{-} \setminus IC^{--}$. A system with interaction model IM is* interaction safe *if its restriction with $\prec$ can perform only complete or maximal incomplete interactions.*

Notice that the rule defining the priority order $\prec$ is similar to the completeness priority rule of definition 7. For a given system, incomplete interactions that are maximal in IC have the same status as complete interactions with respect to non maximal incomplete interactions. Nevertheless, the priority order $\prec$ depends on the considered system as legality of incomplete actions depends on the interaction model considered.

We give below results for checking whether a model is interaction safe.

Dependency graph: Consider a system $S[K]$ consisting of a set of interacting components K with interaction model $IM = (IC, IC^+)$. For $c \in C$ (C is the set of the connectors of IC) we denote by $I^+_{\min}(c)$ the set of the minimal complete interactions of c, and write $I^+_{\min}(C)$ for $\{i \in I^+_{\min}(c)\}_{c \in C}$.

The dependency graph of $S[K]$ is a labelled bipartite graph with two sets of nodes: the components of K, and nodes labelled with elements of the set $\{(c, \alpha(c)) \mid c \in C \ \wedge\ I^+_{\min}(c) = \emptyset\} \cup \{(c, \alpha) \mid c \in C \ \wedge\ \alpha \in I^+_{\min}(c)\}$, where $\alpha(c)$ is the maximal interaction of c (involving all the elements of c).

The edges are labelled with actions of A as follows:

Let $(c, \alpha) = (\{a_1, \ldots, a_n\}, \alpha)$ be a node of the graph and assume that for an action a_i of c, $owner(a_i) \in K$ is the component which is owner of action a_i. For all actions a_i of c occurring in α, add an edge labelled with a_i from $owner(a_i)$ to (c, α). For all actions a_i of c, add an edge labelled with a_i from (c, α) to $owner(a_i)$ if a_i is offered in some *incomplete state* of $owner(a_i)$, that is, a state in which no complete or maximal action is offered.

The graph encodes the dependency between interacting actions of the components in the following manner. If a component has an input edge labelled a_i from a node $(\{a_1, \ldots, a_n\}, \alpha)$, then for a_i to occur in some interaction of $\{a_1, \ldots, a_n\}$ containing α it is necessary that all the actions labelling input edges of $(\{a_1, \ldots, a_n\}, \alpha)$ interact.

We call a circuit in the dependency graph *non trivial* if it encompasses more than one component node.

Example 2 (Producer/consumer). Consider a producer providing data to two consumers. Interaction is by rendez-vous and takes place if at least one of the two consumers can get an item. The interaction model is described by $C = \big\{\{put, get_1, get_2\}\big\}$ and $IC^+ = \{put \mid get_1, put \mid get_2, put \mid get_1 \mid get_2\}$. The dependency graph is shown in figure 5.

Definition 9 (Cooperativity). *Let a and b be labels of input and output edges of a component k in the dependency graph of $S[K]$. We say that a component $k \in K$ is cooperative with respect to (a, b) if from any state of $B[k]$ with a transition labelled a there exists a transition labelled b.*

$k \in K$ is cooperative *in a circuit γ in the dependency graph if it is cooperative wrt. (a, b), where a and b are the arcs of γ entering and leaving k, respectively.*

Theorem 1 (Interaction safety). *A system model is* interaction safe *if its dependency graph contains a non-empty sub-graph G such that (1) G contains all its predecessors, (2) any component in G is deadlock-free, and in any elementary circuit γ of G, either (3a) there exists a component k that is cooperative in γ and whose successor node in γ is a binary interaction, or (3b) the set of components k in γ whose successor node is not a binary interaction, is not empty, and all components in this set are cooperative in γ.*

Proof. Assume that the system is in an incomplete state, that is, a state from which only incomplete actions are possible. Then each component in G offers some incomplete action since it is deadlock-free. We consider the sub-graph G'

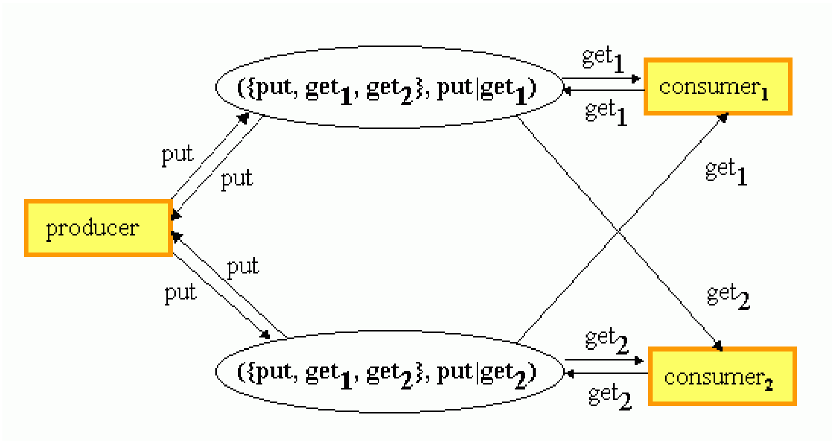

Fig. 5: Dependency graph for the producer/two consumer example.

of G that represents dependencies in the current state: G' has an edge from an interaction node (c, α) to a component node k if k is actually waiting for α in the current state; G' has the same edges from component to interaction nodes as G. G' has the same set of components as G since any component of G is awaiting at least one incomplete action.

If according to *(3a)* one of the components k is cooperative in some non trivial elementary circuit γ G', and the successor node (c, α) of k in γ is a binary interaction, then k and the successor of (c, α) can interact via the complete or maximal interaction α.

Otherwise, all non trivial circuits in G' satisfy condition *(3b)*. Let k be some component in a strongly connected sub-graph of G' not having any predecessors. Such a sub-graph exists since any component is node of some non-trivial circuit. Let γ be a non-trivial circuit in G' containing k, and consider some non-binary interaction node (c, α) in γ. Let k' be an arbitrary predecessor node of (c, α) in G'. By the choice of k, k' and (c, α) are in some non-trivial circuit γ' of G'. γ' satisfies *(3b)*, which implies that k' is cooperative in γ'. That is, all predecessors of (c, α) are cooperative, such that the complete or maximal interaction α is enabled.

In both cases, at least one complete or maximal interaction is enabled, which means that any non-maximal incomplete interaction is disabled in $(B, \prec)$. ■

Intuitively, the hypotheses of Theorem 1 make sure that any circular dependency between the occurrence of strict interactions is broken by some cooperative component. Notice that by definition components are cooperative with respect to (a, a) for any action a. If the dependency graph has a backwards-closed subgraph all of whose elementary circuits are self-loops with the same label then the model is interaction safe.

Example 3 (Producer/consumer). For example 2, the only subgraph G satisfying the backward closure requirement is the whole dependency graph. Let $n_1 = (\{put, get_1, get_2\}, put \shortmid get_1)$ and $n_2 = (\{put, get_1, get_2\}, put \shortmid get_2)$. Γ_G contains two non-trivial elementary circuits $\gamma_1 = (producer, n_1, consumer_2, n_2)$ and $\gamma_2 = (producer, n_2, consumer_1, n_1)$. Since the *producer* is trivially cooperative wrt. the pair (put, put), condition *(3a)* is satisfied. If all three components are deadlock-free, the system is interaction safe.

Deadlock-Freedom We give some results about deadlock-freedom preservation for transitions systems with priorities. Similar results have been obtained for timed transition systems with priorities in [5].

Definition 10 (Deadlock-freedom). *A transition system is called* deadlock-free *if it has no sink states. A system is deadlock-free if the transition system with priorities representing it is deadlock-free.*

Proposition 4 (Composability). *Deadlock-freedom is preserved by priority orders that is, if B is deadlock-free then $(B, \prec)$ is deadlock-free for any priority order $\prec$.*

Proposition 5 (Compositionality). *Deadlock-freedom is preserved by composition that is, if $(B_1, \prec^1)$ and $(B_2, \prec^2)$ are deadlock-free then $(B_1, \prec^1)\|(B_2, \prec^2)$ is deadlock-free.*

Proof. Follows from the fact that composition of behaviors preserves deadlock-freedom and from the previous proposition. ■

Proposition 6. *Any system obtained by composition of deadlock-free components is deadlock-free.*

4 Execution Model

Execution models constitute the third layer. They implement constraints which superposed to interaction constraints further restrict the behavior of a system by reducing non determinism. They differ from interaction models from a pragmatic point of view. Interaction models restrict behavior so as to meet global functional properties, especially properties ensuring harmonious cooperation of components and integrity of resources. Execution models restrict behavior so as to meet global performance and efficiency properties. They are often timed and specific to execution platforms. In that case, they describe scheduling policies which coordinate system activities by taking into account the dynamics of both the execution platform and of the system's environment.

We assume that execution models are also described by priority orders, and discuss two interesting uses of execution models.

Asynchronous vs. Synchronous Execution As explained in 2.2, synchronous execution adopts a very strong fairness assumption as in all computation steps components are offered the possibility to execute some quantum of computation. Our thesis is that synchronous execution can be obtained by appropriately restricting the first two layers. Clearly, it is possible to build synchronous systems by using specific interaction models to compose behaviors. This is the case for Statecharts, and synchronous languages whose semantics use parallel composition operators combined with unary restriction operators [17]. Nevertheless, their underlying interaction model uses non strict interaction and specific action composition laws which are not adequate for asynchronous execution.

In the proposed framework, systems consisting of the first two layers are not synchronous, in general. Interactions between components may be loose. Components keep running until they reach some state from which they offer a strongly synchronizing action. Thus, executions are rich in non-determinism resulting from the independence of computations performed in the components. This is the case for formalisms which point to point interaction, such as SDL and UML.

We believe that it is possible to define synchronous execution semantics for appropriate sub-sets of asynchronous languages. Clearly, these sub-sets should include only reactive components, that is, components with distinct input and output actions such that when an input occurs some output(s) eventually occur. The definition of synchronous execution semantics for asynchronous languages is an interesting and challenging problem.

Consider the example of figure 6, a system which is the serial composition of three strongly synchronized components with inputs i_j and outputs o_j, $j = 1, 2, 3$. Assume that the components are reactive in the sense that they are triggered from some idle (stable) state when an input arrives and eventually produce an output before reaching some idle state from where a new input can be accepted. For the sake of simplicity, components have simple cyclic behaviors alternating inputs and outputs.

The interaction model is specified by $\{o_1, i_2, o_2, i_3\} \prec \{i_1, o_3\}$, $\{o_1, i_2\} \prec o_1 \mid i_2$, $\{o_2, i_3\} \prec o_2 \mid i_3$. That is, we assume that i_1 and o_3 are complete as the system is not connected to any environment. In the product of the behaviors restricted with the interaction model each component can perform computation independently of the others provided the constraints resulting from the interaction model are met. This corresponds to asynchronous execution.

The behavior of the two layers can be further constrained by an execution model to become synchronous in the sense that a run of the system is a sequence of steps, each step corresponding to the treatment of an input i_1 until an output o_3 is produced. This can be easily enforced by the order $i_1 \prec o_1 \mid i_2 \prec o_2 \mid i_3 \prec o_3$. This order reflects the causality order between the interactions of the system. In fact, if all the components are at some idle state then all the components are awaiting for an input. Clearly, only i_1 can occur to make the first component evolve to a state from which $o_1 \mid i_2$ can occur. This will trigger successively $o_2 \mid i_3$

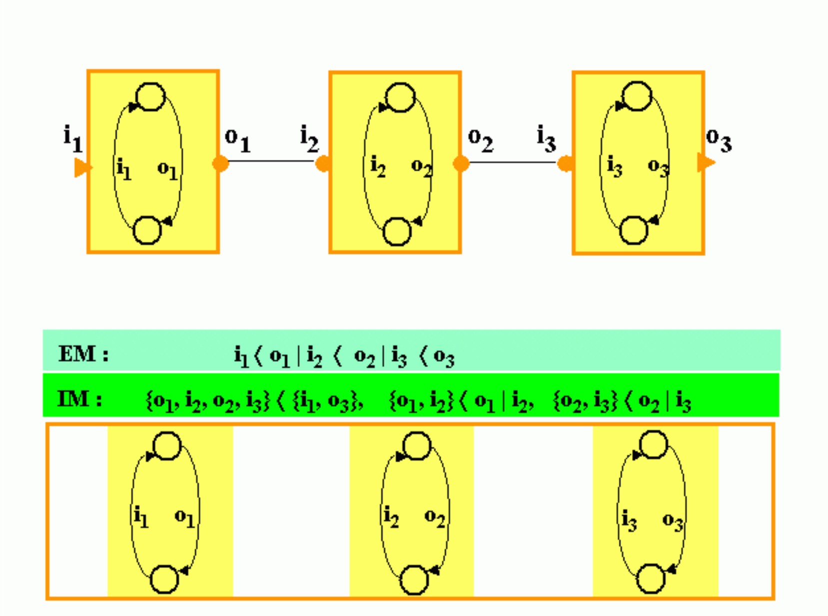

Fig. 6: Enforcing synchronous execution.

and finally o_3. Notice that i_1 cannot be executed as long as a computation takes place in some component.

Scheduling Policies as Execution Models We have shown in [1] that general scheduling policies can be specified as timed priority orders. The following example illustrates this idea for untimed systems.

We model fixed priority scheduling with pre-emption for n processes sharing a common resource (figure 7). The scheduler gives preference to low index processes.

The states of the i-th process are s_i (sleeping), w_i (waiting), e_i (executing), and e_i' (pre-empted). The actions are a_i (arrival), b_i (begin), f_i (finish), p_i (pre-empt), r_i (resume). To ensure mutual exclusion between execution states e_i, we assume that begin actions b_j are complete and synchronize with p_i for all $1 \leqslant i, j \leqslant n$, $i \neq j$. By the maximal progress priority rule, an action b_j cannot occur if some interaction $b_j \mid p_i$ is possible. Similarly, we assume that finish actions f_j are complete and synchronize with r_i for all $1 \leqslant i, j \leqslant n$, $i \neq j$. An action f_j cannot occur if some interaction $f_j \mid r_i$ is possible.

The system is not interaction safe, since the structural properties of theorem 1 cannot exclude the case where the system is in the incomplete state $(e_1', \ldots, e_n')$, that is, all processes are preempted. However, this is the only incomplete state of the system, and it is easy to show that it is not reachable from any other state:

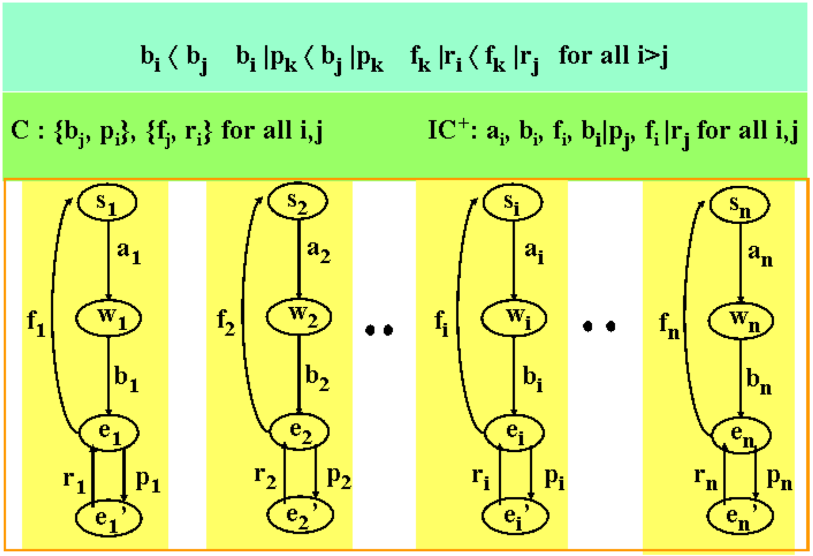

Fig. 7: Fixed-priority preemptive scheduling of processes.

as all actions p_i are incomplete, they are disabled by the completeness priority rule of definition 7 giving priority to the complete actions. Interactions $b_i \mid p_j$ are complete but keep component i, and thus the whole system, in a complete state. Therefore, initialized in a complete state the system always remains in a complete state, where interaction safety is guaranteed.

Scheduling constraints resolve conflicts between processes (b_i and r_i actions) competing for the acquisition of the common resource. They can be implemented by adding a third layer with the priority rules $b_i \prec b_j$, $b_i \mid p_k \prec b_j \mid p_k$, and $f_k \mid r_i \prec f_k \mid r_j$ for all k, and $i > j$.

It is easy to check that these constraints preserve mutual exclusion, in the sense that if the initial state respects mutual exclusion then mutual exclusion holds at any reachable state.

Notice that as the components are deadlock-free and the composition of the interaction and execution priority orders is a priority order, the obtained model is deadlock-free.

5 Discussion

The paper proposes a framework for component composition encompassing heterogeneous interaction and execution.

The framework uses a single powerful associative and commutative composition operator for layered components. Component layering seems to be instrumental for defining such an operator. Existing formalisms combine at the same level behavior composition and unary restriction operators to achieve interaction safety. Layered models allow separation of concerns. Behaviors and restrictions (represented by priority orders) are composed separately. This makes technically possible the definition of a single associative operator.

Interaction models describe architectural constraints on component behavior. Connectors relate interacting actions of different components. They naturally define the set of interactions of a system. The distinction between complete and incomplete interactions is essential for the unification of existing interaction mechanisms. It induces the property of interaction safety characterizing correctness of a model with respect to modeling assumptions about the possibility for interactions to occur independently of their environment. Such assumptions are implicit in existing formalisms. Their satisfaction is enforced on models at the risk of introducing deadlocks. The proposed composition operator preserves deadlock-freedom. Theorem 1 can be used to check interaction safety of models.

The distinction between interaction and execution models is an important one from a methodological point of view. Priority orders are a powerful tool for describing the two models. Their use leads to a semantic model consisting of behaviors and priorities which is amenable to correctness by construction. This is due to the fact that priorities are restrictions that do not introduce deadlocks to an initially deadlock-free system. More results about deadlock-freedom and liveness preservation can be found in [5].

References

1. K. Altisen, G. Gössler, and J. Sifakis. Scheduler modeling based on the controller synthesis paradigm. *Journal of Real-Time Systems, special issue on "control-theoretical approaches to real-time computing"*, 23(1/2):55–84, 2002.
2. F. Balarin, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, M. Sgroi, and Y. Watanabe. *Modeling and Designing Heterogeneous Systems*, volume 2549 of *LNCS*, pages 228–273. Springer-Verlag, 2002.
3. A. Benveniste, P. LeGuernic, and Ch. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16:103–149, 1991.
4. G. Berry and G. Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.
5. S. Bornot, G. Gössler, and J. Sifakis. On the construction of live timed systems. In S. Graf and M. Schwartzbach, editors, *Proc. TACAS'00*, volume 1785 of *LNCS*, pages 109–126. Springer-Verlag, 2000.
6. S. Bornot and J. Sifakis. An algebraic framework for urgency. *Information and Computation*, 163:172–202, 2000.
7. L. de Alfaro and T.A. Henzinger. Interface theories for component-based design. In T.A. Henzinger and C. M. Kirsch, editors, *Proc. EMSOFT'01*, volume 2211 of *LNCS*, pages 148–165. Springer-Verlag, 2001.

8. W.-P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositonal and Non-compositional Methods.* Cambridge University Press, 2001.
9. W.-P. de Roever, H. Langmaack, and A. Pnueli, editors. *Compositionality: The Significant Difference*, volume 1536 of *LNCS*. Springer-Verlag, 1997.
10. OMG Working Group. Response to the omg rfp for schedulability, performance, and time. Technical Report ad/2001-06-14, OMG, June 2001.
11. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.
12. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
13. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.
14. ITU-T. Recommendation Z.100. Specification and Design Language (SDL). Technical Report Z-100, International Telecommunication Union — Standardization Sector, Geneva, 1999.
15. L. Lamport. Specifying concurrent program modules. *ACM Trans. on Programming Languages and Systems*, 5:190–222, 1983.
16. E.A. Lee et al. Overview of the Ptolemy project. Technical Report UCB/ERL M01/11, University of California at Berkeley, 2001.
17. F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *proc. CONCUR*, volume 630 of *LNCS*. Springer-Verlag, 1992.
18. R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25(3):267–310, 1983.
19. R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.
20. SystemC. `http://www.systemc.org`.