Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 195 (2008) 113–132
www.elsevier.com/locate/entcs

An Institutional Theory for #-Components

Francisco Heron de Carvalho-Junior[1,2]

Departamento de Computação
Universidade Federal do Ceará
Fortaleza, Brazil

Rafael Dueire Lins[3]

Departamento de Eletrônica e Sistemas
Universidade Federal de Pernambuco
Recife, Brazil

Abstract

The # (hash) component model has been proposed to bring the advantages of a component-based perspective of software for the development of high performance computing applications, targeting computer architectures enabled for grid, cluster and capability computing. In simple terms, it is a component model for general purpose parallel programming targeting distributed architectures. This paper presents an institutional theory for #-components, which has originated the idea of introducing parameterized and recursive abstract component types in # programming systems, making possible a general notion of skeletal programming.

Keywords: Theory of institutions, category theory, components, parallel programming.

1 Introduction

Advances in technologies for developing software for high performance computing (HPC) applications, commonly originated from computational sciences and engineering, have been influenced by current trends in software *integration*, *distribution*, and *parallelization* [8]. The *component* technology, which has been successfully applied to business applications, has been considered a promising approach to meet those requirements, yielding the birth of several component models, architectures, and frameworks for HPC, including CCA and its compliant frameworks [3], P-COM [33], Fractal/ProActive [5], and many others [44]. Components deal with requirements of integration and distribution in a natural way, but parallel programming

[1] Thanks to CNPq for the financial support (grant 475826/2006-0).
[2] Email: heron@lia.ufc.br
[3] Email: rdl@ufpe.br

doi:10.1016/j.entcs.2007.08.029

based on the peer-to-peer pattern of components interaction is still not suitable for high performance software, mainly when there are non-trivial patterns of parallel synchronization among a collection of processes in architectures with deep hierarchies of parallelism and memory, potentially enabled for grid, cluster, and capability computing [14,1,17]. Indeed, outside the context of components technology, parallel programming artifacts that can exploit the performance of these architectures, such as message passing libraries [20,23], still provide poor *abstraction*, requiring a fair amount of knowledge on architectural details and strategies of parallelism that go far beyond the reach of users in general [28]. Higher level approaches, such as functional programming languages [43] and parallel scientific computing libraries [18] do not merge efficiency with generality. Skeletal programming has been considered a promising alternative, but has reached low dissemination [16]. Parallel programming paradigms that reconcile portability and efficiency with generality and abstraction are still looked for [7,40].

The current trend of parallelism support in components infrastructures has been to encapsulate parallel synchronization inside components, sometimes introducing a minimal set of orthogonal extensions at the coordination level to enable parallel execution. Thus, parallelism is not fully treated as a coordination level concern, as expected. Such approach is influenced by the common trend of taking processes as units of software decomposition in the same dimension of concerns, thus making software engineering disciplines too hard to be applied to parallel programming. Processes and concerns must be placed at orthogonal dimensions of software decomposition [13]. The # component model takes the *hypothesis of orthogonality* as a premise, proposing an alternative for component based parallel programming, inspired in the coordination model of Haskell$_{\#}$ [11], a parallel extension to the functional language Haskell [41]. Most possibly, any component model and parallel programming artifact may be interpreted in terms of the # component model.

This paper introduces an institutional theory for #-components, which has provided valuable insights about abstraction mechanisms that could be supported by # compliant programming systems. In fact, it has suggested that #-components may be categorized by component classes represented by recursive component types, with support to polymorphism based on universal and existential bounded quantification [37]. A generalized use of skeletal programming [16], a promising approach for promoting abstraction in parallel programming, has been made possible from such perspective. A basic background in Category Theory and Petri nets is recommended for readers of this paper.

Section 2 introduces the basic principles behind the # component model. Section 3 includes the formalization of the # component model using Theory of Institutions. Section 4 concludes this paper, describing ongoing and lines for further works regarding formalization, specification, and verification of #-components.

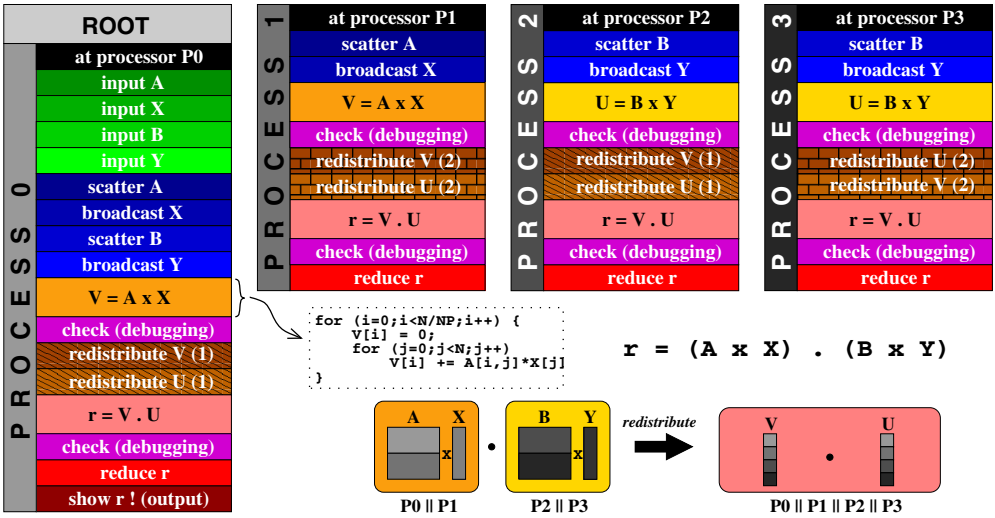


Fig. 1. Slicing a Simple Parallel Program by Concerns

2 The # Component Model: Principles and Intuitions

Motivated by the inadequacy of usual forms of peer-to-peer component interactions for parallel programming, extensions have been proposed to current component artifacts. In general, they free component frameworks from concerns about parallelism at coordination level, leaving synchronization encapsulated inside components, tangling functional code. Despite covering a wide spectrum of parallel programs, they do not reach the generality of message-passing. Indeed, it is common to find papers on HPC components that include "support for richer forms of parallelism" in the list of lines for further investigation. The # component model comes from a deductive generalization of channel-based parallel programming for supporting a general and efficient notion of parallel component. Its origins lie in Haskell$_{\#}$ [11], a parallel extension to the functional language Haskell. The following paragraphs introduce the fundamental principles behind the # component model: *the separation of concerns through process slicing*; and *orthogonality between processes and concerns as units of software decomposition*. Some familiarity with parallel programming is needed to understand the # component model from the intuition behind its basic principles. Figures 1 and 2 sketch a simple parallel program that illustrates the idea of slicing processes by concerns[4]. Let $\mathbf{A}$ and $\mathbf{B}$ be $n \times n$ matrices and X and Y be vectors. It computes $(\mathbf{A} \times X^T) \bullet (\mathbf{B} \times Y^T)$.

In fact, we have searched for the fundamental reasons that make software engineering disciplines too hard to be applied to parallel programming, and concluded that they reside on the trend to mix *processes* and *concerns* in the same dimension of software decomposition, due to the traditional process-centric perspective of parallel programming practice. Software engineering disciplines assume concerns as basic units of software decomposition [34]. We advocate that processes and concerns are

[4] The use of the term *slicing* is not accidental. The problem of slicing of programs according to some criterion was proposed originally in the nineteen seventies and has motivated several research directions [42].

orthogonal concepts. Without any loss of generality, aiming at clarifying the intuitions behind the enunciated *orthogonality hypothesis*, let $\mathcal{P}$ be an arbitrary parallel program formed by a set of processes that synchronize through message-passing. Each process may be split into a set of slices, each one related to a concern. In Figure 1, four processes are sliced into its constituent concerns. Examples of typical concerns are:

(a) a piece of code that represents some meaningful calculation, for example, a local matrix-vector multiplication;

(b) a collective synchronization operation, which may be represented by a sequence of *send/recv* operations;

(c) a set of non-contiguous pieces of code including debugging code of the process;

(d) the identity of the processing unit where the process executes;

(e) the location of a process in a given process topology.

The reader may be convinced that there is a hierarchical dependency between process slices. For instance:

(a) the slice representing collective synchronization operation is formed by a set of slices representing send/recv point-to-point operations;

(b) a local matrix-vector multiplication slice may include a slice that represents the local calculation performed by the process and another one representing the collective synchronization operation that follows it.

If one takes all the processes into consideration, it is easy to see the existence of concerns that cross-cut processes. For example:

(a) the concern of parallel matrix-vector multiplication includes all slices, for each involved process, that defines the (local) role of the process in the overall operation;

(b) the concern of process-to-processor allocation is formed by the set of slices that defines the identities of processors where each process executes.

From the perspective of the parallel program, most of slices inside individual processes do not make sense when observed in isolation. Individually, they are not concerns in the overall parallel program.

The # component model moves parallel programming to a concern-oriented perspective. A *#-component* realizes an application concern, functional or non-functional one. Focused on concerns, programmers may build #-components by combining other #-components through *overlapped composition*. The *units* of a #-component C correspond to the slices of processes of the intended parallel program that define the role of each process in the cooperation to realize the concern of C. The reader must be convinced that the concrete interpretation of a unit is abstract at the perspective of the # component model, since it may correspond to slices of any nature, covering functional and some kinds of non-functional concerns. The units of overlapped #-components are combined to form the units of the #-component being composed. Let u be a unit. The units that are combined to

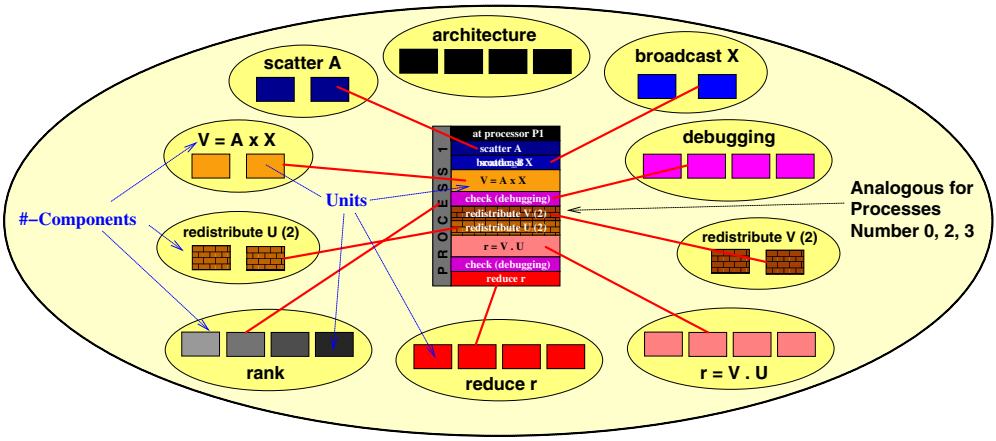


Fig. 2. Forming a Unit by Combining Units of Overlapped #-Components

form u are called slices of u. Slices have a hierarchical structure, like process slices. Sharing between components comes from *fusion of slices* when combining units. For example, two units of #-components whose concerns are linear algebra operations may fuse slices that represent vector or matrix operands, specifying that they are applied to the same instance of data structure. Sharing of data structures is a fundamental issue in components-based HPC. It is also supported by Fractal [9]. *Rooted directed acyclic graphs* have captured the hierarchical structure of slices in units, describing their *signature*. The *protocol* of a unit is specified by a labelled Petri net, whose corresponding formal language says in which order processes may execute their functional slices (labels of the Petri net), which may be interpreted as blocks of code or procedure calls. Petri nets allow the analysis of formal properties and performance evaluation of # programs. An *interface* is defined as a set of units that complies to the same *signature* and *protocol*. In fact, it is the *type* of such set of units. The *component type* of a #-component is defined by the set of interfaces of its units. In more intuitive terms, a #-component differs from usual components by its ability to be deployed in a distributed environment, with each unit being its representant in an individual computer, and by its ability to deal with a rich set of non-functional concerns. A #-program is an executable #-component.

The # component model goes far beyond the idea of raising connectors to the status of first-class citizens [36,39], by promoting them to components, leading to uniformity of concepts. For example, a communication channel could be implemented as a #-component CHANNEL. Fractal also supports connectors as components through *composite bindings*, but *primitive bindings* are not yet components. The # connectors are *exogenous* [30], like in P-COM, while they are *endogenous* in CCA and Fractal.

3 The Institutional Theory of #-Components

A precise and rigorous mathematical characterization of the # component model was developed, based on the Theory of Institutions and Category Theory. Institu-

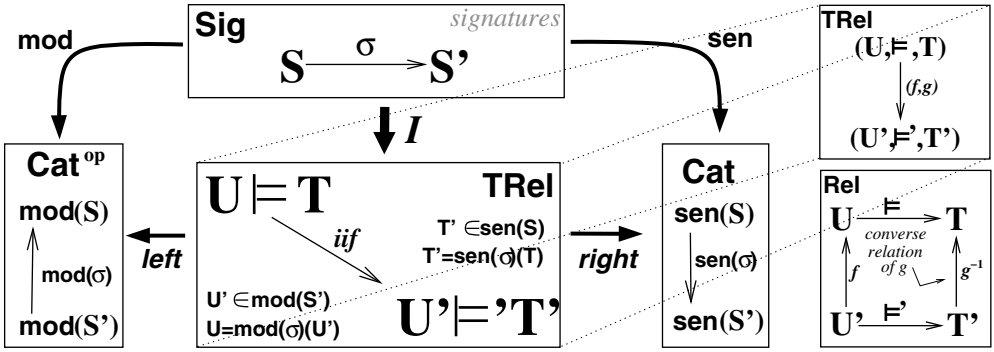


Fig. 3. Institutions and Their Satisfaction Condition

tions were proposed by J. A. Goguen and R. M. Burstall [27]. Their main purpose have been to formalize the idea that satisfaction in logical systems is invariant under change of notation. An interesting variant of institutions are π-institutions, proposed by J. L. Fiadeiro [22]. Institutions have been applied in the semantics of algebraic specification systems [27], type systems theory [24,25], and logical systems [26].

Definition 3.1 An *institution* $\mathcal{K}$ is a quadruple $\langle \mathbf{Sig}, \mathbf{mod}^{\mathcal{K}}, \mathbf{sen}^{\mathcal{K}}, \models^{\mathcal{K}} \rangle$, where:

- **Sig** is a category of *signatures*;
- $\mathbf{mod}^{\mathcal{K}} : \mathbf{Sig} \rightarrow \mathbf{Cat}^{op}$ is a functor that map signatures to the category of *models* with that signature.
- $\mathbf{sen}^{\mathcal{K}} : \mathbf{Sig} \rightarrow \mathbf{Cat}$ is a functor that maps each signature to the category of *sentences* over that signature.
- Let Σ be a **Sig**-object, the relation $\models^{\mathcal{K}}_{\Sigma}$ associates Σ-*models* with Σ-*sentences*. The *satisfaction condition* showed in the commutative diagram in Figure 3, where institutions are defined in terms of the category of twisted relations (**TRel**), must be satisfied.

The Theory of Institutions was adopted in this work because it concisely captures the notions of compatibility between protocols, which gives rise to interface morphisms, and the idea to interpret #-components as models of *component types*, for supporting skeletal programming. This work is deeply inspired by the idea of "types as theories" [25]. Institutions are a sophisticated formal machinery. Any attempt to present further details about them in this paper would be incomplete. For those readers interested in the use of institutions in specification languages and type systems theory, we refer to [27] and [24]. The book by J. L. Fiadeiro is another alternative [22]. Connections of Institutions with logical systems are addressed in reference [26].

3.1 A Formal Characterization for Concerns

First, it is necessary to provide an abstract characterization for *software concerns*. Unfortunately, it is not possible to provide a rigorous and general definition of

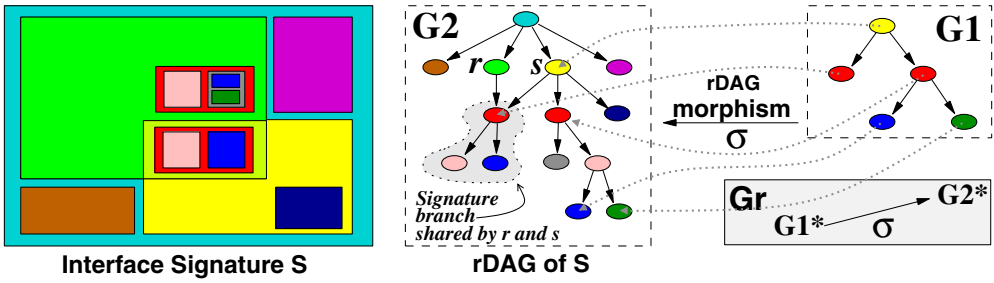


Fig. 4. Signatures and rDAGs

concerns separated from a particular language. But it is possible to characterize certain classes of concerns with special interest in software development. For example, functional concerns may be defined by *computable functions*, specified using some specification language. Richer software specification languages may support specific kinds of non-functional concerns, such as security and demands for computing resources. Concerns may be abstracted from implementation details, at several levels. The functional concern of solving a linear system, for example, may be implemented using several techniques, on top of many programming artifacts. #-components intend to be software "materializations" for concerns. The # component model adopts an abstract characterization of concerns, viewed as elements of a collection named *Concerns*. The metaphor of colors will be used to characterize concerns. Indeed, it is said that a unit has the color of the concern it represents.

3.2 The Institution of Interfaces ($\mathcal{I}$)

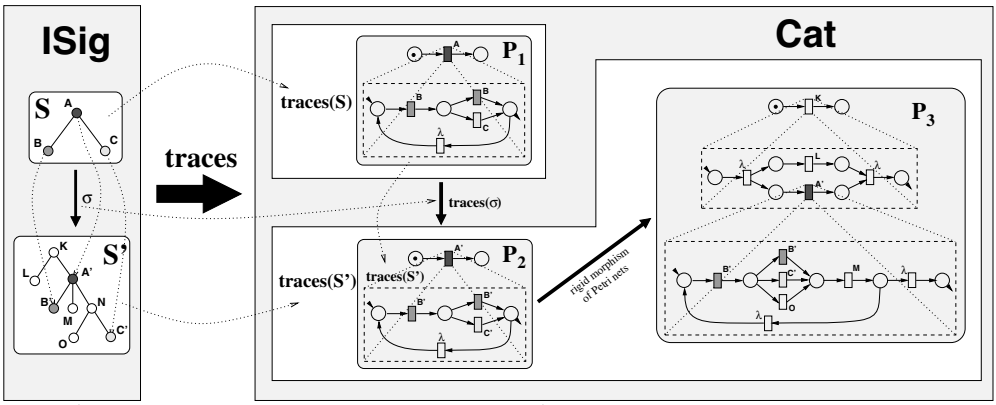
Definition 3.2 [Interface Signatures]

An interface signature is a pair $\langle G, \gamma \rangle$, where $G{=}\langle N, A, \partial_0, \partial_1 \rangle$ is a Rooted Directed Acyclic Graph (rDAG), and $\gamma{:}N{\to}Concerns$ is a function to color nodes with concerns. A rDAG is: (1) an empty graph; or (2) a non-empty directed acyclic graph with a distinguished root node $r \in N$.

Interface signatures are rDAGs whose nodes have a color that denotes their concerns. Let S be an interface signature and r be a slice of S. An interface slice r of S is the branch of S, also an rDAG, that includes all nodes of S that are reachable from r. The hierarchy of slices in processes is captured by rDAGs. Also, notice that nodes with the same color may have sets of sons with distinct sets of colors, respectively, denoting different implementations for a given concern. Figure 4 illustrates interface signatures. In particular, it shows how they capture *fusion of slices*, by allowing interface slices to be shared. For example, an interface slice representing a sparse matrix may be shared by all interface slices that represent computations over that matrix.

Definition 3.3 [Interface Signatures Morphisms]

Let **Gr** be the category of graphs. Let $S_1{=}\langle G_1, \gamma_1 \rangle$ and $S_2{=}\langle G_2, \gamma_2 \rangle$ be interface signatures. An interface signature morphism $\sigma{:}S_1{\to}S_2$ exists if and only if there is a morphism $\sigma^*{:}G_1^*{\to}G_2^*$ in **Gr** and $\gamma_1 = \gamma_2 \circ \sigma_N^*$ (color preservation), where G^* is



S and S' are interface signatures (**ISig**-objects), while $\sigma{:}S{\to}S'$ is a **ISig**-morphism. The protocol P_1 is an example of $\mathbf{sen}^{\mathcal{I}}(S)$-object, while P_2 and P_3 are examples of $\mathbf{sen}^{\mathcal{I}}(S')$-objects. The reader may be attempted to the hierarchical structures of the underlying Petri nets of P_1, P_2, and P_3. The picture emphasizes that P_2 is the image of P_1 in $\mathbf{sen}^{\mathcal{I}}(S')$ through the morphism $\mathbf{sen}^{\mathcal{I}}(\sigma)$, which maps protocols in $\mathbf{sen}^{\mathcal{I}}(S)$ to its corresponding protocols in $\mathbf{sen}^{\mathcal{I}}(S')$. Also, there is a general Petri net morphism from P_2 to P_3. All morphisms inside $\mathbf{sen}^{\mathcal{I}}(S)$ and $\mathbf{sen}^{\mathcal{I}}(S')$ are general Petri net morphisms.

Fig. 5. Interface Signatures and Protocols

the freely generated graph of G.

Theorem 3.4 *Interface signatures and their morphisms forms a category, named* $\mathbf{Sig}^{\mathcal{I}}$.

Figure 4 also depicts an example of a morphism between interface signatures (σ) that emphasize that they do not need to preserve roots. The contravariant functor $\mathbf{mod}^{\mathcal{I}}{:}\mathbf{Sig}^{\mathcal{I}}{\to}\mathbf{Cat}^{op}$ maps an interface signature S onto the category of *units* with interface signature S, called S-units. Intuitively, a *unit* is a *process slice* that implements the concern specified by an interface signature. The # programming systems are responsible to define the real nature of what is called a *process slice* and what is meant by a process slice that is a model of an interface signature, but it must be ensured that if an interface signature S with root s says that s has slices $s_1, \ldots, s_n$ as its children, a S-unit U must implement the concern that colors s by combination of units U_i, for $1 \leq i \leq n$, each one complying to the the interface slice s_i. The units U_i are the *unit slices* of U. A unit has a *behavior* that may be viewed as a control flow that defines valid orders for activating a subset of its unit slices. Thus, the behavior of a unit will be described by a set of traces of the form $t_1 t_2 \ldots t_k$ that are valid with respect to its control flow, where each t_i represents the activation of a unit slice. The behavior of a unit will be treated as a terminal Petri net formal language. It is important to emphasize that the meaning of the activation of a unit, when viewed as a unit slice, may vary according to the kind of concern that it addresses. For instance, if a unit slice denotes a computation or synchronization operation, its activation may correspond to a call to a subroutine. Notice that unit slices that denote several kinds of non-functional concerns, such as allocation of processes to processors, do not require an activation semantics. Unit slices with an activation semantics will be called *executable slices*.

Let S be an interface signature. *Tracing sentences* over S, or S-traces, are defined by protocols. A protocol is a hierarchical Petri net, whose transitions are

labelled with interface slices (nodes of the rDAG) that denotes activable concerns. The hierarchical structure of Petri nets of protocols obeys the hierarchy of slices of the interface signature. Therefore, if a transition is labelled with a node s whose ancestor node is s', then:

(i) s' does not label any transition in the protocol; or

(ii) s' is the label of some substitution transition and s belongs to the Petri net page that refines s'.

It is still necessary to give an *interpretation* for tracing sentences over *behavior* of units. Let U be a S-unit and L_t be the formal language that define the behavior of U (set of traces). T is *true* for L_t iif $L_t \subseteq \ell(T)$, where $\ell(T)$ denotes the terminal Petri net language of the protocol T. Let T and T' be tracing sentences over S. T' is deducible from T, written $T \vdash T'$, iif $\ell(T) \subseteq \ell(T')$. As usual, the transitive relation $\vdash^*$ is inductively defined over $\vdash$. The functor $\mathbf{sen}^{\mathcal{I}} : \mathbf{Sig}^{\mathcal{I}} \rightarrow \mathbf{Cat}$ maps an interface signature S to the category of tracing sentences over S, and an interface signature morphism $\sigma : S \rightarrow S'$ to the corresponding inclusion morphism. The morphisms in the category $\mathbf{sen}^{\mathcal{I}}(S)$, for some interface signature S, are *rigid morphisms* of Petri nets [6], in order to ensure existence of products of labelled Petri nets. Let $\sigma : S \rightarrow S'$ be an interface signature morphism and T be a S-trace. $\mathbf{sen}^{\mathcal{I}}(\sigma)(T)$ may be obtained from T by simple application of homomorphism σ_N over transition labels of T. It can be demonstrated that $\mathbf{sen}^{\mathcal{I}}(\sigma)(T)$ is isomorphic to T in categories of Petri nets, since they only differ by the identification of labels. The relation $\models^{\mathcal{I}}_S$: $\mathbf{mod}^{\mathcal{I}}(S) \times \mathbf{sen}^{\mathcal{I}}(S)$ associates S-units with S-traces. Indeed, $U \models^{\mathcal{I}}_S T$ iif T is true for the behavior of U. The union of all $\models^{\mathcal{I}}_S$, for some S, is denoted by $\models^{\mathcal{I}}$.

Theorem 3.5 (The Institution $\mathcal{I}$) *The quadruple* $\langle \mathbf{Sig}^{\mathcal{I}}, \mathbf{sen}^{\mathcal{I}}, \mathbf{mod}^{\mathcal{I}}, \models^{\mathcal{I}} \rangle$ *forms the institution of interfaces, named* $\mathcal{I}$.

Proof. Let $\sigma : S \rightarrow S'$ be an interface signature morphism, $T \in \mathbf{sen}^{\mathcal{I}}(S)$, $U' \in \mathbf{mod}^{\mathcal{I}}(S')$, $U \equiv \mathbf{mod}^{\mathcal{I}}(\sigma)(U')$, and $T' \equiv \mathbf{sen}^{\mathcal{I}}(\sigma)(T)$. It must be proved that $U \models^{\mathcal{I}}_S T$ iif $U' \models^{\mathcal{I}}_S T'$, or the commutative of the diagram in Figure 3. ($\Rightarrow$) First, suppose that $U \models^{\mathcal{I}}_S T$. It follows, from the interpretation of units as slices of processes, that if $t \in \ell(T)$, then t is a trace in the behavior of U. Let $t = s_1 \ldots s_k$. Since the Petri net $\mathbf{sen}^{\mathcal{I}}(\sigma)(T)$, or T', is obtained from T by homomorphism of labels according to σ, then $t' = s'_1 \ldots s'_k \in \ell(T')$, where s'_i is the image of s_i with respect to σ_N. Thus, since σ preserve concerns and, by $\mathbf{mod}^{\mathcal{I}}(\sigma)$, traces of U' are mapped to traces of U by inverse application of the homomorphism σ_N, where each symbol in a trace of U' that is not in the image of σ_N is discarded in the corresponding trace of U, t' is a trace in the behavior of U'. By consequence, $U' \models^{\mathcal{I}}_S T'$. ($\Leftarrow$) Conversely, suppose that $U' \models^{\mathcal{I}}_S T'$. Let $t' = s'_1 \ldots s'_k \in \ell(T')$ be a trace of U'. Since $T' \equiv \mathbf{sen}^{\mathcal{I}}(\sigma)(T)$, by isomorphism between T and $\mathbf{sen}^{\mathcal{I}}(\sigma)(T)$, T may be obtained from T' by replacing s for each label s' in the transitions of T', such that $\sigma_N(s) = s'$. From this fact, it follows directly that each $t \in \ell(T)$ may be obtained from $t' \in \ell(T')$ by application of the homomorphism σ_N. By $\mathbf{mod}^{\mathcal{I}}(\sigma)$, t is a trace of U. Thus, $U \models^{\mathcal{I}}_S T$. □

Let S be an interface signature. An interface I with signature S is defined by a S-presentation $\langle S, \Theta \rangle$ over the institution $\mathcal{I}$, where Θ is a set of tracing sentences over S. Since Petri net languages are closed under intersection, Θ may refer to the labelled Petri net whose language is the intersection of the Petri net languages of the protocols in Θ. Let $I = \langle S, \Theta \rangle$ be an interface. A S-unit U has interface I (U satisfies I) iff $U \models_S \Theta$. The set of all S-units that satisfy the tracing sentence Θ is the *denotation* of Θ, defined as $\Theta^* = \{U \mid U \models_S^{\mathcal{I}} \Theta\}$. Let Ω be a set of units. Then, $\Omega^* = \{T \mid \exists U \in \Omega.\ U \models_S^{\mathcal{I}} T\}$. Θ^{**} may be written as $\Theta^\bullet$. The S-theory presented by the interface I is $\langle S, \Theta^\bullet \rangle$. The category of interfaces is denoted by $\mathbf{Theo}(\mathcal{I})$, the category of theories over $\mathcal{I}$. Under such interpretation, interfaces may be used to classify units, as intended. From the institutional theory, an interface morphism is a presentation morphism $\Psi : \langle S, \Theta \rangle \to \langle S', \Theta' \rangle$ induced from the interface signature morphism $\Psi_S : S \to S'$, such that $\Psi_\Theta(\Theta) \subseteq \Theta'^\bullet$. The *denotation* of Ψ is defined by the forgetful functor $\Psi^* : \Theta'^* \to \Theta^*$.

The institution $\mathcal{I}$ is *liberal*. Given an interface morphism $\Psi : I_1 \to I_2$, where $I_1 = \langle S_1, \Theta_1 \rangle$ and $I_2 = \langle S_2, \Theta_2 \rangle$, let $\Psi^* : \Theta_2^* \to \Theta_1^*$ be its *denotation*, $\Psi_S : S_1 \to S_2$ be the corresponding interface signature morphism, and U_1 be a model in Θ_1^*. Then, there is a model U_2 in Θ_2^* that is said to be *free* over U_1 with respect to Ψ^*. Formally, there is some morphism $i : U_1 \to \Psi^*(U_2)$ such that given any $j : U_1 \to \Psi^*(U_2')$, there is a unique morphism $h : U_2 \to U_2'$ such that $\Psi^*(h) \circ i = j$. Informally, U_2 is the best model in Θ_2^* that "extends" U_1, in the sense that any other extension of U_1, represented by U_2', may be defined in terms of U_2. In fact, informally, we can take U_2 as the S_2-unit obtained from U_1 whose traces are the interleaving of the formal languages defined by the traces U_1 and by the closure of the alphabet formed by the symbols representing the slices of U_2 that are not in U_1. In such case, $\Psi^*(U_2) = U_1$. Thus, since $i = id_{U_1}$, h is unique.

3.2.1 A Subtype Relation for Interfaces

The notion of subtype interface comes from the notion of *subpresentation* in institutional theory. Let $\Psi : \langle S, \Theta \rangle \to \langle S', \Theta' \rangle$ be an interface morphism. Ψ is a subpresentation if $S \subseteq S'$ and $\Theta \subseteq \Theta'$, but such formulation still depends on a suitable notion of subsignature ($\subseteq$). The symbol $<:$, denoting subtype relation, will represent the inverse relation of $\subseteq$. For instance, let S and S' be interface signatures, and $\sigma : S \to S'$ be a interface signature morphism. Then, $S' <: S$ iif σ is a monomorphism that preserve roots.

Theorem 3.6 *Let $\Psi_a : I_x \to I_a$ and $\Psi_b : I_x \to I_b$ be* $\mathbf{Theo}(\mathcal{I})$*-morphisms, such that Ψ_a is a subtype morphism, i. e., $I_a <: I_x$. The categorical* pushout *of Ψ_a and Ψ_b, denoted by $\Psi_a \oplus \Psi_b$, exists.*

Proof. Let $I_x = \langle S_x, \Theta_x \rangle$, $I_a = \langle S_a, \Theta_a \rangle$, and $I_b = \langle S_b, \Theta_b \rangle$. The use of the institutional framework makes necessary only to show that the pushout between the corresponding signature morphisms $\sigma_a : S_x \to S_a$ and $\sigma_b : S_x \to S_b$ exists. For instance, let $S_x{=}\langle G_x, \gamma_x \rangle$, $S_a{=}\langle G_a, \gamma_a \rangle$, and $S_b{=}\langle G_b, \gamma_b \rangle$. The pushout between σ_a and σ_b is defined by two arrows $\sigma_a' {:} S_a \to S_a +_{S_x} S_b$ and $\sigma_b' {:} S_b \to S_a +_{S_x} S_b$, where

$S_a{+}_{S_x}S_b{=}\langle G_a{+}_{G_x}G_b, \gamma_a{+}_{\gamma_x}\gamma_b\rangle$. $G_a{+}_{G_x}G_b$ denotes the vertex of the pushout between $\sigma_a^*{:}G_x^*{\to}G_a^*$ and $\sigma_b^*{:}G_x^*{\to}G_b^*$ in **Gr** by forgetting transitive relation. $G_a{+}_{G_x}G_b$ is also a *rDAG*, since morphisms under consideration preserve roots and direction of arcs are preserved by graph morphisms. $\gamma_a{+}_{\gamma_x}\gamma_b$ is a pushout of morphisms in **Set**, which involves their source and target objects. Since **Gr** and **Set** has all pushouts, we can conclude that it is always possible to build the pushout of two **ISig**-morphisms. □

3.3 The Institution of Component Types (C)

An institution of component types $\mathcal{C}$ may be built over $\mathcal{I}$. The category $\mathbf{Sig}^{\mathcal{C}}$, for signatures of component types, has finite sets of interface signatures as objects and total functions between sets of interface signatures as morphisms, whose mappings are induced from the interface signatures involved. More formally, Let C_1 and C_2 be $\mathbf{Sig}^{\mathcal{C}}$-objects and $\Phi{:}C_1{\to}C_2$ be a $\mathbf{Sig}^{\mathcal{C}}$-morphism. If $\Phi(S_1){=}S_2$, then $S_1 \in C_1$, $S_2 \in C_2$, and there is a $\mathbf{Sig}^{\mathcal{I}}$-morphism $\sigma{:}S_1{\to}S_2$.

Let $C{=}\langle S_1, S_2, \ldots, S_k\rangle$ be a signature of component type. The category of models of C, or *#-components* with component type signature C, and the category of sentences over C are respectively defined as:

$$\mathbf{mod}^{\mathcal{C}}(C){=}\sum_{i=1}^{k}\mathbf{mod}^{\mathcal{I}}(S_i) \quad \textit{and} \quad \mathbf{sen}^{\mathcal{C}}(C){=}\prod_{i=1}^{k}\mathbf{sen}^{\mathcal{I}}(S_i)$$

where $\prod$ and $\sum$ denote categorical k-ary product and coproduct, respectively. Since $\mathbf{mod}^{\mathcal{C}}$ is a contravariant functor, a model of C is $M = \langle U_1, U_2, \ldots, U_k\rangle$, where each U_i is a S_i-unit. A sentence over a component type signature is represented by $\Theta = \langle T_1, T_2, \ldots, T_k\rangle$, where each T_i is a S_i-trace. The relation $\models^{\mathcal{C}}_{C}$: $\mathbf{mod}^{\mathcal{C}}(C) \times \mathbf{sen}^{\mathcal{C}}(C)$, for every component type signature C, associates #-components with signature C with sentences over C. It is induced from the relation $\models^{\mathcal{I}}_{S}$, for interface signatures. For instance, let $C = \langle S_1, S_2, \ldots, S_k\rangle$ be a signature of component type, $\Theta = \langle T_1, T_2, \ldots, T_k\rangle$ be a sentence over C, and $M = \langle U_1, U_2, \ldots, U_k\rangle \in \mathbf{mod}^{\mathcal{C}}(C)$ be a #-component of signature C. $M \models^{\mathcal{C}}_{C} \Theta$ iif $U_i \models^{\mathcal{I}}_{S_i} T_i$, for $1 \leq i \leq k$.

Let $\Phi : C \to C'$ be a $\mathbf{Sig}^{\mathcal{C}}$-morphism. Let $C{=}\langle S_1, \ldots, S_n\rangle$ and $C' = \langle \sigma_1(S_1), \ldots, \sigma_n(S_n), S'_{n+1}, \ldots, S'_{n+m}\rangle$, where σ_i, for $1{\leq}i{\leq}n$, are $\mathbf{Sig}^{\mathcal{I}}$-morphisms that define the mappings of the total function Φ. Let $\Theta = \langle T_1, \ldots, T_n\rangle$ be a trace over C. Then, $\mathbf{sen}^{\mathcal{C}}(\Phi)(\Theta) = \langle \mathbf{sen}^{\mathcal{I}}(\sigma_1)(T_1), \ldots, \mathbf{sen}^{\mathcal{I}}(\sigma_n)(T_n), \emptyset_{n+1}, \ldots, \emptyset_{n+m}\rangle$, where $\emptyset_i$ denotes the empty protocol (an empty Petri net), is the corresponding trace over C'. Since $\emptyset_i$ is the initial object of the category of protocols over the interface signature S_i, the intuition behind such definition says that $\mathbf{Sig}^{\mathcal{C}}$-morphisms do not impose any protocol restriction in the implementation of #-components with respect to units of interface signatures that are not in their image.

Theorem 3.7 (The Institution $\mathcal{C}$) *The quadruple* $\langle \mathbf{Sig}^{\mathcal{C}}, \mathbf{sen}^{\mathcal{C}}, \mathbf{mod}^{\mathcal{C}}, \models^{\mathcal{C}}\rangle$ *forms the institution of components types, named* $\mathcal{C}$.

Proof. Let $\Phi : C \to C'$ be a $\mathbf{Sig}^{\mathcal{C}}$-morphism, $\Theta \in \mathbf{sen}^{\mathcal{C}}(C)$, $M' \in \mathbf{mod}^{\mathcal{C}}(C')$, $M \equiv \mathbf{mod}^{\mathcal{C}}(\Phi)(M')$, and $\Theta' \equiv \mathbf{sen}^{\mathcal{C}}(\Phi)(\Theta)$. It must be

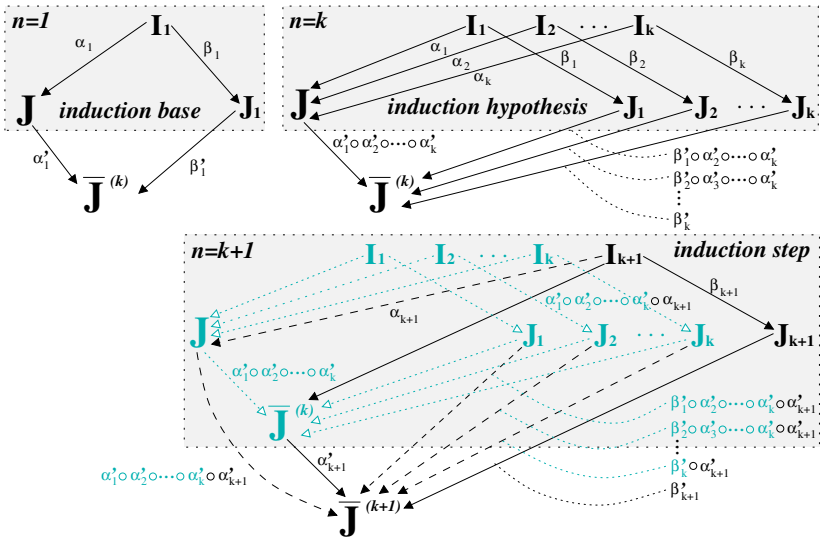


Fig. 6. Diagram for the Proof of Theorem 3.8

proved that $M \models^{\mathcal{C}}_{C} \Theta$ iif $M' \models^{\mathcal{C}}_{C} \Theta'$. Let $C = \langle S_1, \ldots, S_n \rangle$, $M' = \langle U'_1, \ldots, U'_m \rangle$, and $\Theta = \langle T_1, \ldots, T_n \rangle$. By $\Phi = \{\sigma_1, \ldots, \sigma_n\}$, we can take $C' = \langle \sigma_1(S_1), \ldots, \sigma_n(S_n), S'_{n+1}, \ldots, S'_{n+m} \rangle$, $M = \langle \mathbf{mod}^{\mathcal{I}}(\sigma_1)(U'_1), \ldots, \mathbf{mod}^{\mathcal{I}}(\sigma_n)(U'_n) \rangle$ (for simplicity, we take $\langle U'_1, \ldots, U'_n \rangle$, for $n \leq m$, as the units in the image of $\mathbf{mod}^{\mathcal{C}}(\Phi)$), $\Theta' = \langle \mathbf{sen}^{\mathcal{I}}(\sigma_1)(T_1), \ldots, \mathbf{sen}^{\mathcal{I}}(\sigma_n)(T_n), \emptyset_{n+1}, \ldots, \emptyset_{n+m} \rangle$. ($\Rightarrow$) Suppose that $M \models^{\mathcal{C}}_{C} \Theta$. Thus $\mathbf{mod}^{\mathcal{I}}(\sigma_i)(U'_i) \models^{\mathcal{I}}_{S_i} T_i$, for $1 \leq i \leq n$. From $\mathcal{I}$, we can conclude that $U'_i \models^{\mathcal{I}}_{S_i} \mathbf{sen}^{\mathcal{I}}(\sigma_i)(T_i)$, for $1 \leq i \leq n$. Since $U \models^{\mathcal{I}}_{S} \emptyset$, for all S-unit U, we can conclude that $M' \models^{\mathcal{C}}_{C} \Theta'$. ($\Leftarrow$) Conversely, suppose that $M' \models^{\mathcal{C}}_{C} \Theta'$. Thus, $\langle U'_1, \ldots, U'_m \rangle \models^{\mathcal{C}}_{C} \langle \mathbf{sen}^{\mathcal{I}}(\sigma_1)(T_1), \ldots, \mathbf{sen}^{\mathcal{I}}(\sigma_n)(T_n), \emptyset_{n+1}, \ldots, \emptyset_{n+m} \rangle$. From $\mathcal{I}$ and $U_i \models^{\mathcal{I}}_{S_i} \mathbf{sen}^{\mathcal{I}}(\sigma_i)(T_i)$, for $1 \leq i \leq n$, $\mathbf{mod}^{\mathcal{I}}(\sigma_i)(U'_i) \models^{\mathcal{I}}_{S_i} T_i$, for $1 \leq i \leq n$. Thus, $M \models^{\mathcal{C}}_{C} \Theta$. □

Presentations over the institution $\mathcal{C}$ denote *component types*. In fact, a component type C is denoted by a set of interfaces $\{\langle S_i, \Theta_i \rangle \mid i = 1, 2, \ldots, n\}$. A #-component N is a model of the component type C iif it is a model over its underlying presentation C, meaning that $N \models^{\mathcal{C}}_{C} \Theta$, for all $\Theta \in \Theta_C$, written $N \models^{\mathcal{C}}_{C} \Theta_C$. The category of all component types is denoted by $\mathbf{Theo}(\mathcal{C})$, the category of theories over $\mathcal{C}$. A component type morphism is a presentation morphism $\Psi : \langle S_C, \Theta_C \rangle \rightarrow \langle S'_C, \Theta'_C \rangle$ induced from the interface signature morphism $\Phi : S_C \rightarrow S'_C$, such that: $\Theta \in \Theta_C \Rightarrow \Psi(\Theta) \in \Theta'^{\bullet}_{C}$. A ***component class*** represented by the component type C is defined by the set of models that are *free* with respect to the component type (theory) morphism $\Psi : \emptyset \rightarrow C$, where $\emptyset$ denotes the empty component type. In fact, the component class includes the #-components of component type C that are considered *initial*.

3.3.1 The Subtype Relation for Component Types

Let C_1 and C_2 component types. The relation $C_2 <: C_1$ (C_1 is a subtype of C_2) is defined by an injective function $\Phi : C_1 \rightarrow C_2$ between its sets of interfaces. Also,

if $\Phi(I_1) = I_2$, for $I_1 \in C_1$ and $I_2 \in C_2$, then $I_2 <: I_1$. The following theorem is important for the formalization of parameterized and recursive component types.

Theorem 3.8 *Let $\Phi_a : C_x \to C_a$ and $\Phi_b : C_x \to C_b$ be* **Theo**$(\mathcal{C})$*-morphisms, such that Φ_b is a subtype morphism, i. e., $C_a <: C_x$. The pushout $\Phi_a \oplus \Phi_b$ exists.*

Proof. Let $C_x = \langle S_x, \Theta_x \rangle$, $C_a = \langle S_a, \Theta_a \rangle$, and $C_b = \langle S_b, \Theta_b \rangle$. It is only necessary to prove the existence of the pushout $\Phi_a \oplus \Phi_b$, where $\Phi_a{:}S_x \to S_a$ and $\Phi_b{:}S_x \to S_b$ are the **CSig**-morphisms underlying Φ_a and Φ_b. The pushout $\Phi_a \oplus \Phi_b$ is a pair of arrows $\Phi_a{'}{:}S_a \to S_a \oplus_{S_x} S_b$ and $\Phi_b{'}{:}S_b \to S_a \oplus_{S_x} S_b$, where $S_a \oplus_{S_x} S_b$ is induced from pushout in **Set** and **ISig**, since component type signatures are sets (of interface signatures). Thus, interfaces in C_x, through Φ_a and Φ_b, define equivalence classes of interfaces in the disjoint union of the sets C_a and C_b. Interfaces that belong to the same equivalence class are fused to form a unique interface of $S_a \oplus_{S_x} S_b$, from the computation of pushouts of arrows in Φ_a and Φ_b. It is showed how such interfaces are calculated in a general case of equivalence class. For instance, let $\{I_1, \ldots, I_n\}$ be interfaces in C_x mapped by Φ_a to the same interface J in C_a, i. e., $\Phi_a(I_i) = J$, for $1 \leq i \leq n$. Since Φ_b is a subtype morphism, it is an injective function. Thus, for the same set of interfaces, $\Phi_b(I_i) = J_i$, where $J_i \in C_b$, for $1 \leq i \leq n$. The corresponding **Theo**$(\mathcal{I})$-morphisms are $\alpha_i : I_i \to J$ and $\beta_i : I_i \to J_i$, for $1 \leq i \leq n$. This is the most general case where a set of interfaces, represented by $J, J_1, \ldots, J_n$, will be fused to form a new interface $\bar{J}$ in $S_a +_{S_x} S_b$. In fact, $\bar{J}$ is the vertex of the colimit of the diagram representing the arrows $\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_n$ and their respective source and target objects. We want to demonstrate that such colimit always exist. For that, we show how to compute it from the computation of pushouts in **Theo**$(\mathcal{I})$, where one of the arrows is a subtype morphism. Theorem 3.6 has shown the existence of these pushouts. In what follows, we proceed by induction on n. The reader may look at the diagram in Figure 6 for better visualization of the induction. For **n=1 (base)**, the pushout of the arrows $\alpha_1{:}I_1 \to J$ and $\beta_1{:}I_1 \to J_1$ exists in **Theo**$(\mathcal{I})$, since β_1 is a subtype morphism in **Theo**$(\mathcal{I})$ because Φ_a is a subtype morphism in **Theo**$(\mathcal{C})$. It is represented by the triple $\langle \bar{J}^{(1)}, \alpha_1', \beta_1' \rangle$. For **n=k (hypothesis)**, suppose that the colimit of the diagram with arrows $\alpha_1, \ldots, \alpha_k, \beta_1, \ldots, \beta_k$ is represented by $\langle \bar{J}^{(k)}, \alpha_1' \circ \alpha_2' \circ \ldots \circ \alpha_k', \beta_1' \circ \alpha_2 \circ \ldots \circ \alpha_k, \ldots, \beta_2' \circ \alpha_3 \circ \ldots \circ \alpha_k, \beta_k' \rangle$. For **n=k+1 (induction step)**, the colimit of the diagram with arrows $\alpha_1, \ldots, \alpha_{k+1}, \beta_1, \ldots, \beta_{k+1}$, represented by $\langle \bar{J}^{(k+1)}, \alpha_1' \circ \alpha_2' \circ \ldots \circ \alpha_{k+1}', \beta_1' \circ \alpha_2 \circ \ldots \circ \alpha_{k+1}, \ldots, \beta_2' \circ \alpha_3 \circ \ldots \circ \alpha_{k+1}, \beta_{k+1}' \rangle$, can be computed from the colimit of the diagram with arrows $\alpha_1, \ldots, \alpha_k, \beta_1, \ldots, \beta_k$ (dotted lines), whose existence is ensured by the induction hypothesis, again by application of a pushout of two **Theo**$(\mathcal{I})$-morphisms, where one of them is a subtype morphism. For instance, the arrows $\beta_{k+1}'{:}J_{k+1} \to \bar{J}^{(k+1)}$ and $\alpha_{k+1}'{:}\bar{J}^{(k)} \to \bar{J}^{(k+1)}$ comes from the pushout $\beta_{k+1} \oplus (\alpha_1' \circ \ldots \circ \alpha_k' \circ \alpha_{k+1})$, as depicted in Figure 6, where β_{k+1} is known to be a subtype morphism. □

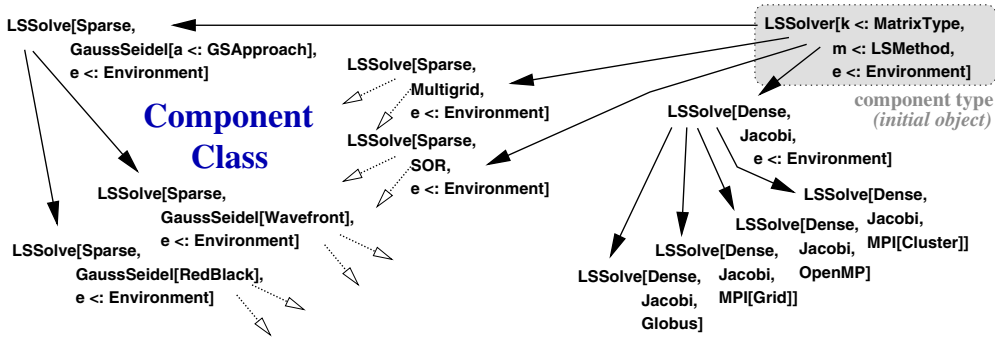


Fig. 7. Component Class for LSSolver

3.4 Skeletal Programming and Parameterized Component Types

The abstraction of skeletons has been widely studied in the context of parallel programming research since the beginning of 1990's, after the seminal work of Murray Cole [15]. Skeletons attempt to capture common patterns of parallel computation, whose implementation may be tuned to specific architectures. In [16], a retrospect of the research on skeletal programming is presented, focusing on the analysis of the reasons that have made difficult the dissemination of skeletal programming in widespread programming environments. This paper adopts a general characterization of skeletons, by relating them to parameterized abstract component types, or simply parameterized component types.

The simpler form of abstraction in # programming resembles routine calls from linear algebra libraries [19]. For example, a user of the #-component LSSolver, a linear system solver comprising N homogenous units, does not need to be aware of synchronization operations performed when one of its units are activated. Thus, a version of LSSolver whose implementation takes advantage of peculiar features of the target parallel architecture may be chosen dynamically, at startup time. Moreover, implementations for LSSolver may be adapted to specific processor topologies, and density properties of the characteristic matrix. Such kind of abstraction will be approached in # programming systems by means of *abstract component types*, with existential quantification semantics [37]. Thus, instead to deal directly with overlapping composition of #-components, programmers may form applications by combining abstract component types to form new abstract component types. In addition, primitive #-components may be developed and declared to inhabit a given abstract component type for a given execution context. At the startup of a deployed abstract component type, a #-component is formed using the tuned versions for the current execution environment and context. Figure 7 illustrates a component class for LSSolver.

programming also supports *skeletal programming* through parametrization of abstract component types that denote common parallel interaction patterns, such as the skeletons proposed by Cole [15] and many others proposed in several subsequent works [10,38]. Such skeletal component types are inhabited by skeleton implementations (#-components) tuned for each parallel architecture. The compo-

nent perspective of skeletons in # programming is inspired by partial topological skeletons of Haskell$_{\#}$ [12], a previous work of the authors that proposed skeletons inspired in collective communication operations of MPI that could be combined by nesting and overlapping. Such idea is very similar to skeletons of eSkel [16], which do not allow skeleton composition. More recently, HOC's (Higher-Order Components)[2] have also been proposed to meet skeletons and components, firstly implemented in the ProActive/Fractal framework for *Web Services* programming [21]. The parameterized component types have been formalized using categorical constructions over the institution $\mathcal{C}$, as following.

Definition 3.9 [Parameterized Component Type]

A parameterized *component type* is defined by a cocone $C[X] = \langle C, \alpha : X \to \Delta_C \rangle$, where $X : I \to \mathbf{Theo}(\mathcal{C})$ is the basis diagram of the cocone and $\Delta_C : I \to \mathbf{Theo}(\mathcal{C})$ is the *constant diagram* over C.

Let C be a component type. The diagram X, with shape I, denotes a subset of the component types that form C that are set to be parameters of C, denoted by $C[X]$. The parameters are denoted as X_i, for $i \in I$. Informally, a parameter X_i may be instantiated by any component C_i, that belongs to another diagram C with the same shape I, for which there is a $\mathbf{Theo}(\mathcal{C})$-morphism $\Phi_i : X_i \to C_i$, such that $C_i <: X_i$ (C_i is subtype of X_i). The choice of a suitable notion of subtyping relation is one of the current research topics with component types in # programming. This paper abstracts from a particular subtype notion.

Definition 3.10 [Parameter Instantiation]

Let $C[X] = \langle C, \alpha : X \to \Delta_C \rangle$ be an arrow component type with shape I. Let $P : I \to \mathbf{Theo}(\mathcal{C})$ be a diagram, and $v : X \to P$ be a diagram morphism (*application*). The morphism v specifies the substitutions for the component type variables in X by component types in P. The application of $C[X]$ through v is the component type $C[v]$, the pushout in the diagram of Figure 9(a).

The nomenclature adopted emphasizes that $C[v]$ is uniquely determined from the parameterized component type $C[X]$ through the instantiation v. The existence of pushout $C[v]$ is ensured by the finite cocompleteness of the institution $\mathcal{C}$.

3.5 Recursive Component Types

Recursive component types are useful to describe self-similar process topologies that are common in parallel programs. Figure 8 presents the recursive parameterized component type TreeFilterDivide, whose signature comprises three interfaces: *ISendTree*, denoting a unit (process slice) that successively splits input data of type D_1 in a pair of data chunks of the same type, transmitting each final chunk to a unit of interface *ICompute*; *ICompute*, which denotes units that takes a chunk of data of type D_1 as input and maps it to a chunk of data of type D_2 as output; and *IRecvTree*, which denotes units that receive a set of chunks of data of type $D2$ and join them in a single chunk of the same type, by successive application of a joining function over pairs of data chunks. TreeFilterDivide has five formal parameters: D_1 and

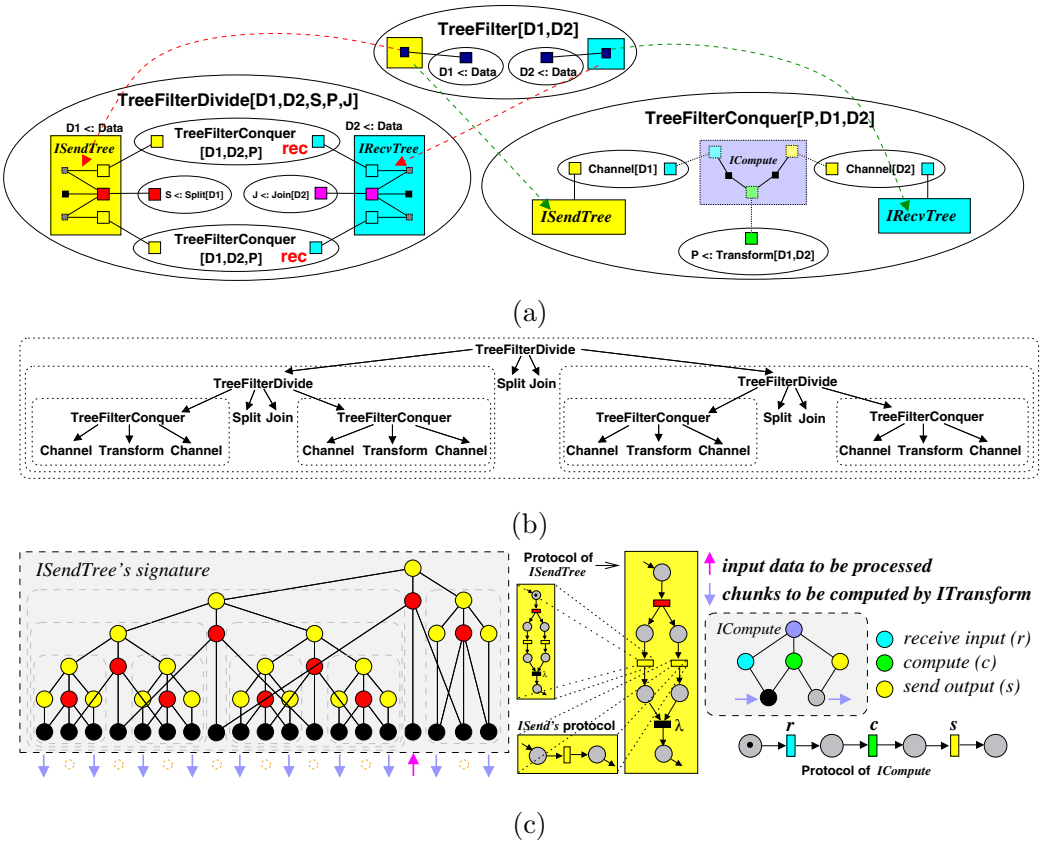
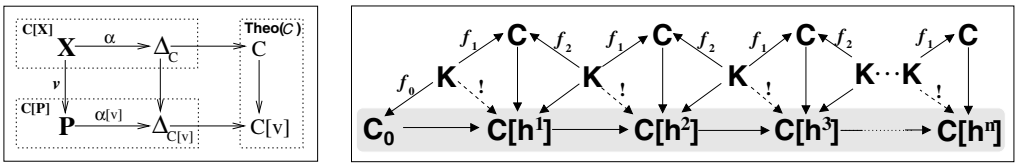


Fig. 8. The Recursive Topology of the TreeFilter Component Type

D_2 specifies the input and output data types; S specifies the splitting function over data chunks of type D_1; J specifies the joining function over data chunks of type D_2; P specifies a function for transforming a chunk of data D_1 to a chunk of type D_2. In fact, the resultant recursive parameterized component type is a skeleton for a well-known divide-and-conquer pattern of parallelism. Figure 8(a) illustrates the recursive configuration of TreeFilterDivide. The component type TreeFilter is configured to be the recursion basis. This is possible because TreeFilterDivide and TreeFilterConquer are both subtypes of TreeFilter, making possible to apply TreeFilterDivide recursively over TreeFilter until the end of the recursion is reached, when TreeFilterConquer is finally applied over TreeFilter. Figure 8(b) presents a recursive unfolding of TreeFilterDivide, with depth 3. In Figure 8(c), it is presented the resultant hierarchy for interface signatures of *ISendTree*, from an recursive unfolding with depth 4, and *ICompute*. The protocols of *ISendTree* and *ICompute* (labelled Petri nets) are illustrated in the figure. In the signature of *ISendTree*, the leave nodes (black circles) are units slices that represent data structures of type D_1 and D_2. The reader may be asking about how recursion basis is reached when unfolding a configuration of a recursive component type. In fact, this is a concern of # programming systems and could be resolved dynamically, at startup or execution time.



(a) (b)

Fig. 9. Diagrams for Parameterized and Recursive Component Types

Let $\mathbf{Theo}(\mathcal{C})_{<:}$ be the wide subcategory of $\mathbf{Theo}(\mathcal{C})$ restricted to subtype morphisms. For formalizing recursive component types, some usual categorical machinery is needed. For instance, let $\mathbf{Rec}(\mathcal{C}) = \Delta^{\mathbf{4}} \uparrow (\Delta \uparrow (\mathbf{id} \uparrow \mathbf{id}))$, where $\uparrow$ denotes the *comma category*, Δ denotes the *diagonal functor* over $\mathbf{Theo}(\mathcal{C})_{<:}$, $\mathbf{id}$ is the *identity functor* over $\mathbf{Theo}(\mathcal{C})$, and $\Delta^{\mathbf{4}}$ denotes the *four-dimensional diagonal functor* over $\mathbf{Theo}(\mathcal{C})_{<:}$. For convenience, we represent a $\mathbf{Rec}(\mathcal{C})$-object by $O = \langle f_0{:}K{\to}C_0, f_1{:}K{\to}C, f_2{:}K{\to}C, f{:}K{\to}D\rangle$. Also, it is defined the endofunctor $\mathbf{F}$ over $\mathbf{Rec}(\mathcal{C})$, such that $\mathbf{F}(\langle f_0, f_1, f_2, f\rangle) = \langle f_0, f_1, f_2, \Theta(\langle f_0, f_1, f_2, f\rangle)\rangle$, where $\Theta(\langle f_0, f_1, f_2, f\rangle) = f_2 \circ (f_1 \oplus f)_r + f_0$ and $(f \oplus f')_r$ denotes the left arrow of the *pushout* between f and f' in $\mathbf{Theo}(\mathcal{C})$, whose existence may be ensured by a suitable subtyping relation.

Definition 3.11 [Recursive Component Type] The fixed point of $\mathbf{F}$, denoted by $\mu(\mathbf{F})$, denotes the category of recursive component types over $\mathbf{Theo}(\mathcal{C})$.

Figure 9(b) illustrates the construction of a recursive component type from the arrows $f_0 : K \to C_0$, $f_1 : K \to C$, and $f_2 : K \to C$. It is important to notice that f_0 and f_2 are $\mathbf{Theo}(\mathcal{C})_{<:}$-morphisms, denoting subtyping relationships. C_0 denotes the *basis component type* of the recursion, while C is the component type where the recursion is applied. In the proposed example, TreeFilterConquer is C_0, while TreeFilterDivide is C. K denotes a component type that is in the structure of C, through f_1, but which is also a subtype of C_0 and C, through f_0 and f_2, respectively. In the example, TreeFilter is K. Thus, C may be applied recursively over K in the structure of C until the recursion basis is reached, when C_0 is finally applied over K.

3.6 The Relation of # Programming with Aspects, Hyperspaces, and Features

Since the # component model is mainly focused on separation of concerns that cross cut the processes of a parallel program, the reader may consider it helpful to understand the relation between modularity in # programming and modularity in some recently proposed artifacts for separation of cross-cutting concerns in software, such as *aspects* [29], *hyperspaces* [35], *features* [4], and so on. For instance, aspects may be encapsulated in #-components. In this perspective, *joinpoints* are the execution points before, after, and around activation of slices in protocols. *Advices* correspond to the interleaving of activations of slices from #-aspects (aspect slices) with other slices in protocols. *Weaving* is related to the process of composition of a parallel program from the overlapping composition of a set of #-components that

represent components, in the usual sense, and aspects. However, modularity in # programming is closer to mechanisms of *multi-dimensional separation of concerns*, where hyperspaces and features are included. In fact, a *hyperslice* that cross cuts a set of classes is like a #-component that cross cuts a set of processes, in such a way that a *unit* of a hyperslice corresponds to a unit of a #-component. The combination of hyperslices to form hypermodules is captured by overlapping composition of #-components, where hyperslices and hypermodules are both viewed as #-components. The relation of features with aspects and hyperspaces have been detailed studied in [32]. Features are closer to hyperspaces, by making refinements correspondent to hyperslices. Transitively, it is easy to see the relation between features and #-components. Also, the idea of synthesizing efficient programs that meet some specification by looking for the best implementation of features has a clear relation to the idea of skeletal programming through existential polymorphism with component types in # programming.

4 Conclusions and Lines for Further Works

This paper presented an institutional theory for #-components. Besides to provide important insights on the nature of #-components and to allow proving important properties about them, such approach has also the advantage of introducing abstraction mechanisms from algebraic specification and type theory into # programming systems. For instance, this leads to the idea of supporting skeletal programming using recursive polymorphic component types.

Work on progress address some pragmatic issues regarding the implementation of type systems for #-components, such as decidability and generality issues that always rise when talking about type systems supporting bounded quantification [37]. Besides that, it is intended to extend the institutional framework presented in this paper to deal with specific kinds of concerns. As discussed along the paper, the current approach abstracts from the nature of concerns addressed by #-components. For example, a simple approach to support functional concerns could be to enrich the specification of units of functional component types with *pre-conditions*, *post-conditions*, and *invariants*, trying to apply specification matching techniques [45] for verification of combination of units in the overlapped composition of #-components. Such approach suggests the application of notions of *behavioral subtyping* [31] onto component type systems.

References

[1] Allan, B. A., R. C. Armstrong, A. P. Wolfe, J. Ray, D. E. Bernholdt and J. A. Kohl, *The CCA Core Specification in a Distributed Memory SPMD Framework*, Concurrency and Computation: Practice and Experience **14** (2002), pp. 323–345.

[2] Alt, M., J. Dünnweber, J. Müller and S. Gorlatch, *HOCs: Higher-Order Components for Grids*, in: *Workshop on Component Models and Systems for Grid Applications (in ICS'2004)* (2004).

[3] Armstrong, R., D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker and B. Smolinski, *Towards a Common Component Architecture for High-Performance Scientific Computing*, in: *The 8th IEEE International Symposium on High Performance Distributed Computing* (1999).

[4] Batory, D., J. N. Sarvela and A. Rauschmayer, *Scaling Step-Wise Refinement*, in: *Proceedings of the International Conference on Software Engineering*, 2003.

[5] Baude, F., D. Caromel and M. Morel, *From Distributed Objects to Hierarchical Grid Components*, in: *International Symposium on Distributed Objects and Applications* (2003).

[6] Bednarczyk, M. A. and A. Borzyszkowski, *General Morphisms of Petri Nets*, Lecture Notes in Compute Science **1113** (1999), pp. 190–199.

[7] Bernholdt D. E., J. Nieplocha and P. Sadayappan, *Raising Level of Programming Abstraction in Scalable Programming Models*, in: *IEEE International Conference on High Performance Computer Architecture (HPCA), Workshop on Productivity and Performance in High-End Computing (P-PHEC)*, Madrid, Spain (2004), pp. 76–84.

[8] Bramley, R., R. Armstrong, L. McInnes and M. Sottile, *High-Performance Component Software Systems*, SIAM **49** (2005), p. .

[9] Bruneton, E., T. Coupaye and J. B. Stefani, *Recursive and Dynamic Software Composition with Sharing*, in: *European Conference on Object Oriented Programming (ECOOP'2002)* (2002).

[10] Campbell, D. K. G., *Towards the Classification of Algorithmic Skeletons*, Technical Report YCS 276, Department of Computer Science, University of York (1996).
URL citeseer.nj.nec.com/campbell96towards.html

[11] Carvalho Junior, F. H. and R. D. Lins, *Haskell$_{\#}$: Parallel Programming Made Simple and Efficient*, Journal of Universal Computer Science **9** (2003), pp. 776–794.

[12] Carvalho Junior, F. H. and R. D. Lins, *Topological Skeletons in Haskell$_{\#}$*, in: *International Parallel and Distributed Processing Symposium (IPDPS)* (2003), 8 pages.

[13] Carvalho Junior, F. H. and R. D. Lins, *Separation of Concerns for Improving Practice of Parallel Programming*, INFORMATION, An International Journal **8** (2005).

[14] Chiu, K., "An Architecture for Concurrent, Peer-to-Peer Components," Ph.D. thesis, Department of Computer Science, Indiana University (2001).

[15] Cole, M., "Algorithm Skeletons: Structured Management of Paralell Computation," Pitman, 1989.

[16] Cole, M., *Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming*, Parallel Computing **30** (2004), pp. 389–406.

[17] Dongarra, J., *Trends in High Performance Computing*, The Computer Journal **47** (2004), pp. 399–403.

[18] Dongarra, J., I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon and A. White, "Sourcebook of Parallel Computing," Morgan Kauffman Publishers, 2003 .

[19] Dongarra, J., I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon and A. White, "Sourcebook of Parallel Computing," Morgan Kauffman Publishers, 2003.

[20] Dongarra, J., S. W. Otto, M. Snir and D. Walker, *A Message Passing Standard for MPP and Workstation*, Communications of ACM **39** (1996), pp. 84–90.

[21] Dünnweber, J., Gorlatch S., F. Baude, L. Legrand and N. Parlavantzas, *Towards Automatic Creation of Web Services for Grid Component Composition*, in: *CoreGRID Workshop on Grid Systems, Tools and Environments*, 2005.

[22] Fiadeiro, J. L., "Categories for Software Engineering," Springer, 2005.

[23] Geist, G., A. Beguelin, J. Dongarra, W. Jiang, R. Manchek and V. S. Sunderam, *PVM: Parallel Virtual Machine - A User's Guide and Tutorial for Networked Parallel Computing*, MIT Press, Cambridge (1994).

[24] Goguen, J., *Higher-Order Functions Considered Unnecessary for Higher-Order Programming*, in: D. A. Turner, editor, *Research Topics in Functional Programming*, Addison-Welsey, Reading, MA, 1990 pp. 309–351.
URL citeseer.ist.psu.edu/goguen90higher.html

[25] Goguen, J., *Types as Theories*, in: G. M. Reed, A. W. Roscoe and R. F. Wachter, editors, *Topology and Category Theory in Computer Science*, Oxford, 1991 pp. 357–390.
URL citeseer.ist.psu.edu/503432.html

[26] Goguen, J., "Logica Universalis - Towards a General Theory of Logic," Springer, 2005 pp. 113–133.

[27] Goguen, J. and R. Burnstal, *Institutions: Abstract Model Theory for Specification and Programming*, Journal of ACM **39** (1992), pp. 95–146.

[28] Gorlatch, S., *Send-Recv Considered Harmful? Myths and Truths about Parallel Programming*, ACM Transactions in Programming Languages and Systems **26** (2004), pp. 47–56.

[29] Kiczales, G., J. Lamping, Menhdhekar A., Maeda C., C. Lopes, J. Loingtier and J. Irwin, *Aspect-Oriented Programming*, , **1241** (1997), pp. 220–242.

[30] Lau, K., P. V. Elizondo and Z. Wang, *Exogenous Connectors for Software Components*, Lecture Notes in Computer Science (Proceedings of 2005 International SIGSOFT Symposium on Component-Based Software Engineering - CBSE'2005) **3489** (2005), pp. 90–108.

[31] Leavens, G. T., *Concepts of Behaviral Subtyping and a Sketch of Their Extension to Component-Based Systems*, in: *Foundations of Component Based Systems* (2000), pp. 113–135.

[32] Lopez Herrejon, R. E., D. Batory and W. Cook, *valuating Support for Features in Advanced Modularization Technologies*, Lecture Notes in Computer Science (Proceedings of ECOOP'2005) **3586** (2005), pp. 169–194.

[33] Mahmood, N., G. Deng and J. C. Browne, *Compositional development of parallel programs.*, in: L. Rauchwerger, editor, *LCPC*, Lecture Notes in Computer Science **2958** (2003), pp. 109–126.

[34] Milli, H., A. Elkharraz and H. Mcheick, *Understanding Separation of Concerns*, in: *Workshop on Early Aspects - Aspect Oriented Software Development (AOSD'04)*, 2004, pp. 411–428.

[35] Ossher, H. and P. Tarr, *Multi-Dimensional Separation of Concerns and the Hyperspace Approach*, in: *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development* (2000), University of Twente, Enschede, The Netherlands.

[36] Perry, D. E., *The Inscape Environment*, in: *The Proceedings of the 11th International Conference on Software Engineering* (1989).

[37] Pierce, B., "Types and Programming Languages," The MIT Press, 2002.

[38] Rabhi, F. A. and S. Gorlatch, "Patterns and Skeletons for Parallel and Distributed Computing," Springer, 2002.

[39] Shaw, M., *Procedure Calls are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status*, in: *International Workshop on Studies of Software Design*, Lecture Notes in Computer Science (1994).

[40] Skjellum, A., P. Bangalore, J. Gray and Bryant B., *Reinventing Explicit Parallel Programming for Improved Engineering of High Performance Computing Software*, in: *International Workshop on Software Engineering for High Performance Computing System Applications* (2004), pp. 59–63, Edinburgh.

[41] Thompson, S., "Haskell, The Craft of Functional Programming," Addison-Wesley Publishers Ltd., 1996.

[42] Tip, F., *A Survey of Program Slicing Techniques*, Journal of Programming Languages **3** (1995), pp. 121–189.

[43] Trinder, P. W., H.-W. Loidl and R. F. Pointon, *Parallel and Distributed Haskells*, Journal of Functional Programming **12** (2002), pp. 469–510.

[44] van der Steen, A. J., *Issues in Computational Frameworks*, Concurrency and Computation: Practice and Experience **18** (2006), pp. 141–150.

[45] Zaremsky, A. M. and J. M. Wing, *Specification Matching of Software Components*, in: *3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1995.