An Institutional Theory for #-Components

Francisco H. de Carvalho Junior¹, Rafael D. Lins², Ana T. de C. Martins¹

¹Departamento de Computação – Universidade Federal do Ceará
Campus do Pici, Bloco 910 – Fortaleza, Brazil

²Departamento de Eletrônica e Sistemas – Universidade Federal de Pernambuco
Av. Acadêmico Hélio Ramos s/n – Recife, Brazil

{heron,ana}@lia.ufc.br, rdl@ufpe.br

Abstract. The # (hash) component model has been proposed for large scale parallel programming in high performance computing. This paper presents an institutional theory for #-components, which has given insight that originated the ideas of parallel programming with parameterized and recursive component types, with support to a general notion of skeletons.

1. Introduction

Advances in technologies for programming multi-disciplinary, multi-physics, multi-scale, and high-performance software for computational sciences and engineering [7] have been influenced by current trends in software integration, distribution, and parallelization. The component technology, which has been used successfully in business applications, has been considered a promising approach to meet those requirements, yielding the birth of several component models, architectures, and frameworks for high performance computing (HPC), including CCA and its compliant frameworks [3], P-COM [25], Fractal/PROACTIVE [4], and many others [31]. Components deal with requirements of integration and distribution in a natural way, but parallel programming based on the peer-to-peer pattern of components interaction is still not suitable for high performance software, mainly when there are non-trivial forms of interaction of parallelism [1, 12] targeted at architectures with hierarchical parallelism, potentially enabled for grid, cluster, and capability computing [12, 1, 15]. Outside the context of components, parallel programming artifacts that can exploit the performance of high-end distributed architectures, such as message passing libraries like MPI, 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. Higher level approaches, such as functional programming languages and parallel scientific computing libraries do not merge efficiency with generality. Skeletal programming has been considered a promising alternative, but has reached low dissemination [14]. Parallel programming paradigms that reconcile portability and efficiency with generality and abstraction are still looked for [6].

HPC components models and architectures have proposed parallel extensions for current component technology [1], without reaching the generality of message-passing parallel programming, however. In general, they are not concerned with parallelism at coordination level, encapsulating parallel synchronization inside components. Besides that,
they have been 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 [11]. 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# [9], a parallel extension to the functional language Haskell. Most possibly, any component model may be interpreted in terms of abstractions 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 frameworks. In fact, it has suggested that #-components may be categorized by component classes represented by recursive component types, with support to polymorphism based on existential bounded quantification [28]. A generalized use of skeletal programming [14], a promising approach for promoting abstraction in parallel programming, has been made possible from such perspective. The ability to deal with concepts from Category Theory and Petri nets is required to read 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.

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# [9], 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. Let $A$ and $B$ be $n \times n$ matrices and $X$ and $Y$ be vectors. It computes $(A \times X^T) \bullet (B \times Y^T)$.

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

1The 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 [30].
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 [26]. We advocate that processes and concerns are orthogonal concepts. Without any loss of generality, aiming at clarifying the intuitions behind the enunciated orthogonality hypothesis, let \( P \) 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 \texttt{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 \texttt{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 and non-functional. Focused on concerns, programmers may build #-components by combining other #-components through overlapped composition. 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 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 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 HPC. It is also supported by Fractal [8]. 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. The component type of a #-component is defined by the set of interfaces of its units. A #-program is an executable #-component.

The # component model goes far beyond the idea of raising connectors to the status of first-class citizens [27, 29], by promoting them to components, leading to uniformity of concepts. For example, a communication channel could be implemented as a
Fractal also supports connectors as components through composite bindings, but primitive bindings are not yet components. The # connectors are exogenous[23], 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. Institutions were proposed by J. A. Goguen and R. M. Burstall [22]. 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 [18]. Institutions have been applied in the semantics of algebraic specification systems [22], type systems theory [19, 20], and logical systems [21].

Definition 1 An institution $\mathcal{K}$ is a quadruple $(\Sigma\text{IG}, \text{mod}^{\mathcal{K}}, \text{sen}^{\mathcal{K}}, |^{\mathcal{K}})$, where:

- $\Sigma\text{IG}$ is a category of signatures;
- $\text{mod}^{\mathcal{K}} : \Sigma\text{IG} \rightarrow \text{CAT}^{\text{op}}$ is a functor that map signatures to the category of models with that signature.
- $\text{sen}^{\mathcal{K}} : \Sigma\text{IG} \rightarrow \text{CAT}$ is a functor that maps each signature to the category of sentences over that signature.
- Let $\Sigma$ be a $\Sigma\text{IG}$-object, the relation $|^{\mathcal{K}}$ associates $\Sigma$-models with $\Sigma$-sentences. The satisfaction condition showed in the commutative diagram in Figure 3, where institutions are defined in terms of the category of twisted relations ($\text{TREL}$), 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” [20]. 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 [22] and [19]. The book by J. L. Fiadeiro is another alternative [18]. Connections of Institutions with logical systems are addressed in reference [21].
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 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 2 (Interface Signatures)** An interface signature is defined by a pair $\langle G, \gamma \rangle$, where $G=\langle N, A, \partial_0, \partial_1 \rangle$ is a Rooted Directed Acyclic Graph (rDAG), and $\gamma: N \rightarrow \text{Concerns}$ is a function to paint nodes with concerns. A rDAG is: (1) an empty graph; or (2) a non-empty directed acyclic graph with a distinguished root node $r \in N$, from which there is a path to any other node.

Interface signatures are rDAGs whose nodes, representing interface slices, are painted with a color that denotes their concerns. Let $S$ be an interface signature and $r$ be a node in the rDAG of $S$. The interface slice of $S$ with root $r$ (or, the interface slice $r$) is the rDAG that includes all nodes of $S$ that are reachable from $r$. Intuitively, rDAGs capture the hierarchy of slices in processes, described in the previous sections, allowing a concern to have several implementations. Indeed, if two roots of interface slices are painted with the same color, their respective sets of children may be painted with distinct colors. Interface signatures also capture sharing, since nodes in rDAGs may have several parents. For example, an interface slice representing a sparse matrix may be shared by all interface slices that represent computations over that matrix. Figure 4 illustrates interface signatures, showing how they capture fusion of slices.

**Definition 3 (Interface Signatures Morphisms)** Let $\text{GR}$ 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
Figure 5: Interface Signatures and Protocols

\[ \sigma : S_1 \rightarrow S_2 \text{ is a morphism } \sigma^* : G_1^* \rightarrow G_2^* \text{ in } GR \text{ and } \gamma_1 = \gamma_2 \circ \sigma_N^* \text{ (color preservation), where } G^* \text{ is the freely generated graph of } G. \]

**Theorem 1** Interface signatures and their morphisms forms a category, named ISIG.

Figure 4 also depicts an example of a morphism between interface signatures that emphasize that they do not need to preserve roots. The contravariant functor \( \text{mod}^{\sigma} : \text{ISIG} \rightarrow \text{CAT}^{\text{op}} \) 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 \# compliant programming frameworks 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 interface slice \( s_i \) of \( S \). 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 (executing) its executable unit slices. The behavior of a unit will be described by a set of traces of the form \( t_1t_2\ldots t_k \) 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.

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 nodes of \( S \). The hierarchical structure of the Petri nets of protocols obeys the hierarchy of interface slices. Therefore, if a transition is labelled with a node \( s \) whose ancestor node is \( s' \), then (1) \( s' \) does not label any transition in the protocol or (2) \( s' \) 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 \( L_\ell \) be the formal language that define the behavior of \( U \) (set of traces). \( T \) is true for \( L_\ell \) iif \( L_\ell \subseteq \ell(T) \), 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 \(T \vdash T'\), if \(\ell(T) \subseteq \ell(T')\). As usual, the transitive relation \(\vdash^*\) is inductively defined over \(\vdash\). The functor \(\text{sen}^T: \text{ISIG} \to \text{CAT}\) maps an interface signature \(S\) to the category of tracing sentences over \(S\), and an interface signature morphism \(\sigma: S \to S'\) to the corresponding inclusion morphism. The morphisms in the category \(\text{sen}^T(S)\), are rigid morphisms of Petri nets [32, 5], in order to ensure existence of products of labelled Petri nets. Let \(T\) be a trace sentence (protocol) over \(S\). \(\text{sen}^T(\sigma)(T)\) may be obtained from \(T\) by simple application of homomorphism \(\sigma_N\) over transition labels of \(T\). Figure 5 illustrates this fact with \(P_2 = \text{sen}^T(\sigma)(P_1)\). Notice that \(\text{sen}^T(\sigma)(T)\) is isomorphic to \(T\) in categories of Petri nets, since they only differ by the identification of labels. The relation \(\models_{\mathcal{S}}^T\); \(\text{mod}^T(S) \times \text{sen}^T(S)\) associates \(S\)-units with \(S\)-traces. Indeed, \(U \models_{\mathcal{S}}^T T\) if \(T\) is true for the behavior of \(U\). The union of all \(\models_{\mathcal{S}}^T\), for some \(S\), is denoted by \(\models^T\).

**Theorem 2 (The Institution \(\mathcal{I}\))** The quadruple \((\text{ISIG}, \text{sen}^T, \text{mod}^T, \models^T)\) forms the institution of interfaces, named \(\mathcal{I}\).

**Proof.** Let \(\sigma: S \to S'\) be an interface signature morphism, \(T, U \in \text{sen}^T(S), U' \in \text{mod}^T(S')\), \(U \equiv \text{mod}^T(\sigma)(U')\), and \(T' \equiv \text{sen}^T(\sigma)(T)\). It must be proved that \(U \models_{\mathcal{S}}^T T\) if \(U' \models_{\mathcal{S}}^T T'\), or the commutative of the diagram in Figure 3. \((\Rightarrow)\) First, suppose that \(U \models_{\mathcal{S}}^T T\). 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 \(\text{sen}^T(\sigma)(T)\), or \(T'\), is obtained from \(T\) by homomorphism of labels according to \(\sigma\), then \(t' = s_1' \ldots s_k' \in \ell(T')\), where \(s_i'\) is the image of \(s_i\) with respect to \(\sigma_N\). Thus, since \(\sigma\) preserve concerns and, by \(\text{mod}^T(\sigma)\), traces of \(U'\) are mapped to traces of \(U\) by inverse application of the homomorphism \(\sigma_N\), where each symbol in a trace of \(U'\) that is not in the image of \(\sigma_N\) is discarded in the corresponding trace of \(U\), \(t'\) is a trace in the behavior of \(U'\). By consequence, \(U' \models_{\mathcal{S}}^T T'\). \((\Leftarrow)\) Conversely, suppose that \(U' \models_{\mathcal{S}}^T T'\). Let \(t' = s_1' \ldots s_k' \in \ell(T')\) be a trace of \(U'\). Since \(T' \equiv \text{sen}^T(\sigma)(T)\), by isomorphism between \(T\) and \(\text{sen}^T(\sigma)(T)\), \(T\) 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 \(\sigma_N\). By \(\text{mod}^T(\sigma)\), \(t\) is a trace of \(U\). Thus, \(U \models_{\mathcal{S}}^T T\).

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 \(\Theta\) is a set of tracing sentences over \(S\). Since Petri nets are closed under intersection, \(\Theta\) will be also referred as the Petri net obtained from the intersection of the set of protocols in \(\Theta\), without loss of generality. Let \(I = \langle S, \Theta\rangle\) be an interface. A \(S\)-unit \(U\) has interface \(I\) (\(U\) satisfies \(I\)) if \(U \models_S \Theta\). The set of all \(S\)-units that satisfy the tracing sentence \(\Theta\) is the denotation of \(\Theta\), defined as \(\Theta^* = \{U \mid U \models_S \Theta\}\). Let \(\Omega\) be a set of units. Then, \(\Theta^* = \{T \mid \exists U \in \Omega. U \models_{\mathcal{S}}^T T\}\). \(\Theta^{**}\) may be written as \(\Theta^*\). The \(S\)-theory presented by the interface \(I\) is \(\langle S, \Theta^*\rangle\). The category of interfaces is denoted by \(\text{THEO}(\mathcal{I})\), 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 \(\Psi(\Theta) \subseteq \Theta'\). The denotation of \(\Psi\) 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, \(\sigma: S_1 \to S_2\) be the corresponding interface signature morphism, and \(U_1\) be a model in \(\Theta_1^*\). Then, there is a model \(U_2\) in \(\Theta_2^*\) that is said to be free over \(U_1\) with respect to \(\Psi^*\). 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 \(\Theta_2^*\).
that “extends” $U_1$, in the sense that any other one ($U_2'$) 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 $i = \text{id}_{U_1}$, $h$ 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 $\Psi : \langle S, \Theta \rangle \rightarrow \langle S', \Theta' \rangle$ be an interface morphism. $\Psi$ 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 $\ll$, denoting subtype relation, will represent the inverse relation of $\subseteq$. For instance, let $S$ and $S'$ be interface signatures, and $\sigma : S \rightarrow S'$ be a interface signature morphism. Then, $S' \ll S$ if $\sigma$ is a monomorphism that preserve roots.

**Theorem 3** Let $\Psi_a : I_a \rightarrow I_a$ and $\Psi_b : I_a \rightarrow I_b$ be $\text{THEO}(\mathcal{I})$-morphisms, such that $\Psi_a$ is a subtype morphism, i.e., $I_a \ll: I_a$. The categorical pushout of $\Psi_a$ and $\Psi_b$, denoted by $\Psi_a \oplus \Psi_b$, exists.

**Proof.** Let $I_a = \langle S_a, \Theta_a \rangle$, $I_a = \langle S_b, \Theta_b \rangle$, 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 $\sigma_a : S_a \rightarrow S_b$ and $\sigma_b : S_b \rightarrow S_b$ exists. For instance, let $S_a = \langle G_a, \gamma_a \rangle$, $S_a = \langle G_a, \gamma_a \rangle$, and $S_b = \langle G_b, \gamma_b \rangle$. The pushout between $\sigma_a$ and $\sigma_b$ is defined by two arrows $\sigma'_b : S_a \rightarrow S_a + S_b, \sigma'_b : S_a \rightarrow S_a + S_b$, and $\sigma'_a : S_a \rightarrow S_a + S_b$, where $S_a + S_b = \langle G_a + \gamma_a, G_b, \gamma_a + \gamma_b \rangle$. $G_a + \gamma_a, G_b$ denotes the vertex of the pushout between $\sigma'_a : G_a \rightarrow G_a$ and $\sigma'_b : G_b \rightarrow G_b$ in $\text{GR}$ by forgetting transitive relation. $G_a + \gamma_a, G_b$ is also a rDAG, since morphisms under consideration preserve roots and direction of arcs are preserved by graph morphisms. $\gamma_a + \gamma_a, \gamma_b$ is a pushout of $\gamma_a$ in $\text{SET}$. Such construction allows to represent a component type signature $C$ simply by a set of interface signatures $\langle S_1, S_2, \ldots, S_k \rangle$, an $\text{I} \sigma$-morphism between component type signatures by a total function between sets of interface signatures, whose mappings are interface signature morphisms. The category of models of $C$, or $\text{#-components}$ with component type signature $C$, and the category of sentences over $C$, respectively, are:

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

where $\prod$ and $\sum$ denote categorical $k$-ary product and coproduct, respectively. Since, $\text{mod}^\mathcal{I}$ 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
\[\theta = \langle T_1, T_2, \ldots, T_k \rangle, \text{ where each } T_i \text{ is a } S_i\text{-unit.} \]

The relation \(\models^C : \text{mod}^C(C) \times \text{sen}^C(C)\), for every component type signature \(C\), associates \#-components with signature \(C\) with sentences over \(C\). It is induced from the relation \(\models^S\), 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 \(M = \langle U_1, U_2, \ldots, U_k \rangle \in \text{mod}^C(C)\) be a \#-component of signature \(C\). \(M \models^C \theta \) if \(U_i \models^S T_i\), for \(1 \leq i \leq k\).

Let \(\phi : C \to C'\) be a CSIG-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 \(\sigma_i\), for \(1 \leq i \leq n\), are ISIG-morphisms that define the mappings of the total function \(\phi\). Let \(\theta = \langle T_1, \ldots, T_n \rangle\) be a trace over \(C\). Then, \(\text{sen}^C(\phi)(\theta) = \langle \text{sen}^C(\sigma_1)(T_1), \ldots, \text{sen}^C(\sigma_n)(T_n), \emptyset_{n+1}, \ldots, \emptyset_{n+m} \rangle\), 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 CSIG-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. In fact, it is intended to capture the notion of hidden unit in \#-components.

**Theorem 4 (The Institution C)** The quadruple \(\langle \text{CSIG}, \text{sen}^C, \text{mod}^C, \models^C \rangle\) forms the institution of component types, named \(C\).

**Proof.** Let \(\phi : C \to C'\) be a CSIG-morphism, \(\theta \in \text{sen}^C(C), M' \in \text{mod}^C(C')\), \(M \equiv \text{mod}^C(\phi)(M')\), and \(\theta' \equiv \text{sen}^C(\phi)(\theta)\). It must be proved that \(M \models^C \theta\) if \(M' \models^C \theta'\). Let \(C = \langle S_1, \ldots, S_n \rangle\), \(M' = \langle U'_1, \ldots, U'_n \rangle\), and \(\theta = \langle T_1, \ldots, T_n \rangle\). By \(\phi = \{\sigma_1, \ldots, \sigma_n\}\), we can take \(C' = \langle \sigma_1(S_1), \ldots, \sigma_n(S_n), S'_{n+1}, \ldots, S'_{n+m} \rangle\), \(M = \langle \text{mod}^C(\sigma_1)(U'_1), \ldots, \text{mod}^C(\sigma_n)(U'_n) \rangle\) (for simplicity, we take \(\langle U'_1, \ldots, U'_n \rangle\), for \(n \leq m\), as the units in the image of \text{mod}^C(\phi)), \(\theta' = \langle \text{sen}^C(\sigma_1)(T_1), \ldots, \text{sen}^C(\sigma_n)(T_n), \emptyset_{n+1}, \ldots, \emptyset_{n+m} \rangle\). \(\Rightarrow\) Suppose that \(M \models^C \theta\). Thus \(\text{mod}^C(\sigma_i)(U'_i) \models^S T_i\), for \(1 \leq i \leq n\). From \(\mathcal{I}\), we can conclude that \(U'_i \models^S \emptyset_i\), for all \(S\)-unit \(U\). So, we can conclude that \(M' \models^C \theta'\). \(\Leftarrow\) Conversely, suppose that \(M' \models^C \theta'\). Thus, \(\langle U'_1, \ldots, U'_n \rangle \models^C \langle \text{sen}^C(\sigma_1)(T_1), \ldots, \text{sen}^C(\sigma_n)(T_n), \emptyset_{n+1}, \ldots, \emptyset_{n+m} \rangle\). From \(\mathcal{I}\)
and $U_i \models_T^2 \text{sen}^T(\sigma_i)(T_i)$, for $1 \leq i \leq n$, $\text{mod}^T(\sigma_i)(U'_i) \models_T^2 T_i$, for $1 \leq i \leq n$. Thus, $M \models_C^2 \Theta$.

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 $C$ if it is a model over its underlying presentation $\mathcal{C}$, meaning that $N \models_C^2 \Theta$, for all $\Theta \in \Theta_C$, written $N \models_C^2 \Theta_C$. The category of all component types is denoted by $\text{THEO}(\mathcal{C})$, 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: $\Theta \in \Theta_C \Rightarrow \Psi(\Theta) \in \Theta'_C$. The institution $\mathcal{C}$ is also liberal. It may be proved directly from the fact that $\mathcal{I}$ is liberal. 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 initial in a categorical setting.

### 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 5** Let $\Phi_a : C_x \rightarrow C_a$ and $\Phi_b : C_x \rightarrow C_b$ be $\text{THEO}(\mathcal{C})$-morphisms, such that $\Phi_b$ is a subtype morphism, i.e., $C_a < : C_x$. The pushout $\Phi_a \oplus \Phi_b$ exists.

**Proof.** Let $C_a = \langle S_a, \Theta_a \rangle$, $C_b = \langle S_b, \Theta_b \rangle$, and $C_x = \langle S_x, \Theta_x \rangle$. It is only necessary to prove the existence of the pushout $\Phi_a \oplus \Phi_b$, where $\Phi_a : S_x \rightarrow S_a$ and $\Phi_b : S_x \rightarrow S_b$ are the $\text{CSIG}$-morphisms underlying $\Phi_a$ and $\Phi_b$. The pushout $\Phi_a \oplus \Phi_b$ is a pair of arrows $\Phi_a : S_x \rightarrow S_a \oplus S_b$ and $\Phi_b : S_x \rightarrow S_a \oplus S_b$, where $S_a \oplus S_b$ is induced from pushout in $\text{SET}$ and $\text{ISIG}$, since component type signatures are sets (of interfaces signatures). Thus, interfaces in $C_x$, through $\Phi_a$ and $\Phi_b$, 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 $S_a \oplus S_b$, from the computation of pushouts of arrows in $\Phi_a$ and $\Phi_b$. 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 $\Phi_a$ to the same interface $J$ in $C_a$, i.e., $\Phi_a(I_i) = J$, for $1 \leq i \leq n$. Since $\Phi_b$ 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 $\text{THEO}(\mathcal{I})$-morphisms are $\alpha_i : I_i \rightarrow J$, and $\beta_i : I_i \rightarrow J_i$, 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 $J$ in $S_a \oplus S_b$. In fact, $J$ 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 $\text{THEO}(\mathcal{I})$, where one of the arrows is a subtype morphism. Theorem 3 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 $\alpha_1 : I_1 \rightarrow J$ and $\beta_1 : I_1 \rightarrow J_1$ exists in $\text{THEO}(\mathcal{I})$, since $\beta_1$ is a subtype morphism in $\text{THEO}(\mathcal{I})$ because $\Phi_a$ is a subtype morphism in $\text{THEO}(\mathcal{C})$. It is represented by the triple $\langle J^{(1)}, J_1^{(1)}, \beta_1 \rangle$. 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 $\langle J^{(k)}, \alpha_1 \circ \alpha_2 \circ \cdots \circ \alpha_k, \beta_1 \circ \alpha_2 \circ \cdots \circ \alpha_k, \ldots, \beta_1 \circ \alpha_3 \circ \cdots \circ \alpha_k, \beta_k \rangle$. 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 $\langle J^{(k+1)}, \alpha_1 \circ \alpha_2 \circ \cdots \circ \alpha_{k+1}, \beta_1 \circ \alpha_2 \circ \cdots \circ \alpha_{k+1}, \ldots, \beta_1 \circ \alpha_3 \circ \cdots \circ \alpha_{k+1}, \beta_{k+1} \rangle$, 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 $\text{THEO}(\mathcal{I})$-morphisms, where
one of them is a subtype morphism. For instance, the arrows $\beta_{k+1} : \bar{J}_{k+1} \to \bar{J}^{(k+1)}$ 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 $\beta_{k+1}$ is known to be a subtype morphism.

3.4. Skeletal Programming and Parameterized Component Types

The abstraction of skeletons has been widely studied in parallel programming research since the beginning of 1990’s, after the seminal work of M. Cole [13]. Skeletons attempt to capture common patterns of parallel computation, whose implementation may be tuned to specific architectures. In [14], 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 generalized notion of skeletons, where they are related to component types.

The simpler form of abstraction in # programming resembles routine calls from linear algebra libraries [16]. For example, a user of the #-component LSSOLVER, a linear system solver, does not need to be aware of synchronization operations performed when one of its units are activated. # programming also supports skeletal programming through component types, representatives of classes of components that address the same concern through implementations that are tuned to specific execution environments. For instance, implementations for LSSOLVER may be adapted to specific parallel architectures, process topologies, and density properties of a matrix $A$. Figure 7 presents a component class for LSSOLVER. Component types adopts a polymorphic type system based on a decidable instance of existential bounded quantification[28] for classification of #-components. A component type may be viewed as a generalized notion of skeleton, where #-components are skeleton implementations. The component perspective of skeletons in # programming is inspired by partial topological skeletons of Haskell# [10]. 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 [17]. Parameterized component types have been formalized using categorical constructions over the institution $C$.

**Definition 4 (Parameterized Component Type)** A parameterized component type is defined by a cocone $C[X] = \langle C, \alpha : X \to \Delta_C \rangle$, where $X : I \to \text{THEO}(C)$ is the basis diagram of the cocone and $\Delta_C : I \to \text{THEO}(C)$ is the constant diagram over $C$. 
Let \( C \) be a component type. The diagram \( X \), with shape \( I \), denotes a sub-set of the component types that form \( C \) that are set to be parameters of the parameterized version of \( C \), named \( C[X] \). 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 \( C_i <_C X_i \) (there is a \( \text{T}_\text{HEO}(C) \)-morphism \( \Phi_i : X_i \to C_i \)).

**Definition 5 (Parameter Instantiation)** Let \( C[X] = \langle C, \alpha : X \to \Delta C \rangle \) be a parameterized component type with shape \( I \). Let \( P : I \to \text{T}_\text{HEO}(C) \) be a diagram, and \( v : X \to P \) be a diagram morphism (application) in \( \text{T}_\text{HEO}(C) \). 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 Theorem 5.

### 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 \( \text{TreeFilterDivide} \), whose signature comprises three interfaces: \( \text{ISendTree} \), 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 \( \text{ICompute} \); \( \text{ICompute} \), 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 \( \text{IRecvTree} \), 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. \( \text{TreeFilterDivide} \) has five formal parameters: \( D_1 \) and \( D_2 \) 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 \( D_2 \). Figure 8(a) illustrates the recursive configuration of \( \text{TreeFilterDivide} \). The component type \( \text{TreeFilter} \) is configured to be the recursion basis with the recursive modifier. This is possible because \( \text{TreeFilterDivide} \) and \( \text{TreeFilterConquer} \) are both subtypes of \( \text{TreeFilter} \), making possible to apply \( \text{TreeFilterDivide} \) recursively over \( \text{TreeFilter} \) until the end of the recursion is reached, when \( \text{TreeFilterConquer} \) is finally applied over \( \text{TreeFilter} \). Figure 8(b) depicts configurations of component types \( \text{TreeFilter}, \text{TreeFilterDivide}, \text{TreeFilterConquer} \). The protocols of \( \text{ISendTree} \) and \( \text{ICompute} \) (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 # compliant programming frameworks. Notice that unfolding could be resolved dynamically.

Let $\text{Theo}(C)_{<}$ be the wide subcategory of $\text{Theo}(C)$ restricted to subtype morphisms\(^2\). For formalizing recursive component types, some usual categorical machinery is needed. For instance, let $\text{Rec}(C) = \Delta^4 \uparrow (\Delta \uparrow (\text{id} \uparrow \text{id}))$, where $\uparrow$ denotes the comma category, $\Delta$ denotes the diagonal functor over $\text{Theo}(C)_{<}$, $\text{id}$ is the identity functor over $\text{Theo}(C)$, and $\Delta^4$ denotes the four-dimensional diagonal functor over $\text{Theo}(C)_{<}$. For convenience, we represent a $\text{Rec}(C)$-object by $O = \langle f_0: K \rightarrow C_0, f_1: K \rightarrow C, f_2: K \rightarrow C, f: K \rightarrow D \rangle$. Also, it is defined the endofunctor $F$ over $\text{Rec}(C)$, such that $F(\langle f_0, f_1, f_2, f \rangle) = \langle f_0, f_1, f_2, \Theta(\langle f_0, f_1, f_2, f \rangle) \rangle$, where $\Theta(\langle f_0, f_1, f_2, f \rangle) = f_2 \circ (f_1 \oplus f) + f_0$ and $(f \oplus f')_c$ denotes the left arrow of the pushout between $f$ and $f'$ in $\text{Theo}(C)$, whose existence is ensured by Theorem 5 because $f$ is a subtype morphism, and $+$ denotes the coproduct in $\text{Rec}(C)$.

**Definition 6 (Recursive Component Type)** The fixed point of $F$, denoted by $\mu(F)$, denotes the category of recursive component types over $\text{Theo}(C)$.

Figure 9(b) illustrates the construction of a recursive component type from the arrows $f_0: K \rightarrow C_0$, $f_1: K \rightarrow C$, and $f_2: K \rightarrow C$. It is important to notice that $f_0$ and $f_2$ are $\text{Theo}(C)_{<}$-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 $K$.

4. Conclusions and Lines for Further Works

This paper presented an institutional theory for #-components. Besides providing important insights on the nature of #-components and to allow proving important formal properties about them, such approach has also the advantage of introducing into # programming the concept of abstraction from algebraic specification and institutional type theory. This leads to the idea of programming with recursive polymorphic component

\(^2\)It is a category, since identities in $\text{Theo}(C)$ are subtype morphisms and subtype relation is transitive.
types. Such features, combined to the intrinsic properties of #-components, yields an important contribution in the support for large scale programming in HPC.

Work on progress address some pragmatic issues regarding the implementation of type systems to be resolved for programming with component types, related to the decidability and generality issues that always rise when talking about existential bounded quantification in type systems [28]. Besides that, it is intended to extend the institutional framework presented in this paper to deal with specification of functional concerns. In fact, the current approach abstracts from the nature of concerns. For example, a simple approach could be to enrich the specification of units of functional component types with pre-conditions and post-conditions, trying to apply specification matching techniques [33] for verifying combination of units in the overlapped composition of component types. Such approach also suggests the application of notions of behavioral subtyping [24] onto component type systems.

References


