A Categorical Characterization for the Compositional Features of the # Component Model

Francisco Heron de Carvalho Junior *
Departamento de Computação
Universidade Federal do Ceará
Bloco 910, Campus do Pici
Fortaleza, Brazil
heron@lia.ufc.br

Rafael Dueire Lins †
Departamento de Eletrônica e Sistemas
Universidade Federal de Pernambuco
Rua Acadêmico Hélio Ramos, s/n
Recife, Brazil
rdl@ufpe.br

ABSTRACT

The # programming model attempts to address the needs of the high performance computing community for new paradigms that reconcile efficiency, portability, abstraction and generality issues on parallel programming for high-end distributed architectures. This paper provides a semantics for the compositional features of # programs, based on category theory.

1. INTRODUCTION

Due to the advent of clusters and grids, the processing power of large-scale distributed architectures is now accessible for a wider number of academic and industrial users, most of them non-specialists in computers and programming. This new context in high-performance computing (HPC) has brought new challenges to computer scientists. Contemporary parallel programming technologies that can exploit the potential performance of distributed architectures, such as message passing libraries like MPI and PVM, still require a fair amount of knowledge on the architecture and the strategy of parallelism used. This knowledge goes far beyond the reach of naive users [7]. The high-level approaches available today do not join efficiency with generality. The scientific community still looks for a parallel programming paradigm that reconciles portability and efficiency with generality and a high-level of abstraction [4].

The # component model attempts to meet the needs of the HPC community [5], by moving parallel programming from a process-based perspective to a concern-oriented one based on components, separating concerns of specification of computations from concerns related to their coordination, and providing a number of features for abstraction in topological composition of components. This paper provides a categorical [2] foundation for the # component model, focused on the semantics of its compositional features.

2. THE # COMPONENT MODEL

The # component model moves parallel programming from a *process-based* perspective towards a *concern-oriented* one. Without any loss of generality, in the former perspective, a parallel program may be seen as a set of processes that synchronize by means of communication channels. Application *concerns* [10] are scattered across the implementation of processes. In fact, a process may be decomposed in a set of *slices*, each one describing the role of the process with respect to a concern. In the latter outlook, ***components*** are programming abstractions that address concerns. In # programming, a component is described as a set of units organized in a network topology through synchronization channels that connect their interfaces. The interface of a unit comprises a set of input and output ports, whose activation order is dictated by a *protocol*, specified using a formalism with the expressiveness of labeled Petri nets. Component units have direct correspondence to processes slices. In fact, a # process is defined by the unification of a set of units from distinct components. Each unit from a component corresponds to a slice that describes the role of a process with respect to its concern. It is not difficult to see that *processes are orthogonal to concerns* (Figure 1) and that concern-oriented parallel programming fits better modern software engineering methodologies.

In # programming, the concerns about computations and the ones related to their coordination are separated in composed and simple components, respectively. Composed components comprise the ***coordination medium*** of # programs, while simple components comprise their ***computation medium***. Components may be combined with other components yielding new components, through *nesting* or *overlapping* composition. Nesting composition occurs when a simple/composed component is assigned to a unit of another composed component. Overlapping composition occurs when units from disjoint composed components are unified. Component models of today allow only nesting composition [1, 3] and does not support separation of cross-cutting concerns. The # model also brings the support to *skeletal programming* [6] to component-based programming. The essence of # programming is to provide compositional features for raising the level of abstraction in dealing with basic channel-based parallel programming. It is supposed that any parallel programming artifact may be defined in terms of # programming abstractions. This section provided only an overview of the # component model features. Details about compositional and abstraction issues in # programming will be given in the categorical semantics presented in the next sections, where concepts like *compositional interfaces* and *interface*

*Supported by FUNCAP and CNPq.
†Supported by CNPq.

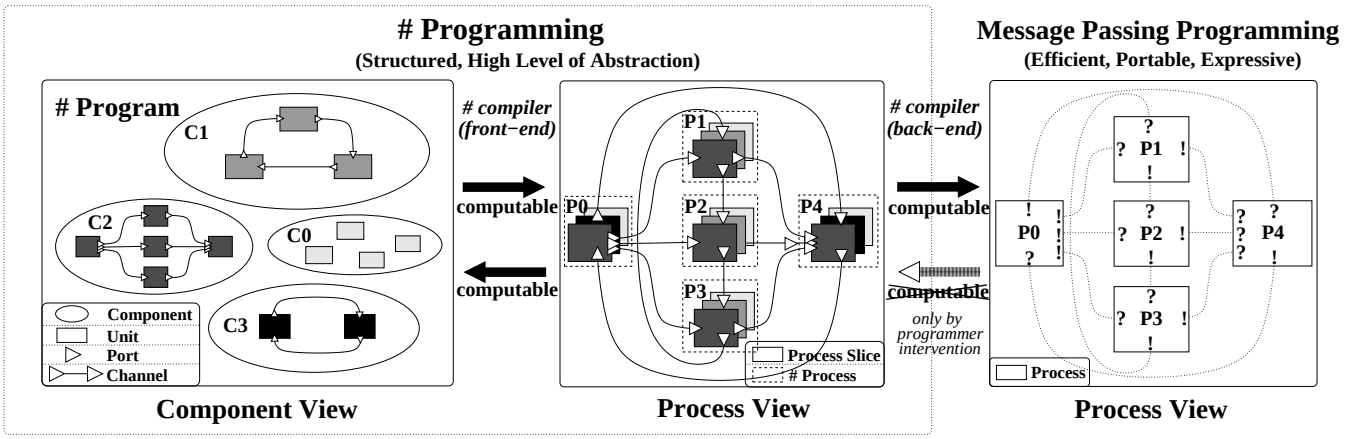

Figure 1: Components versus Processes

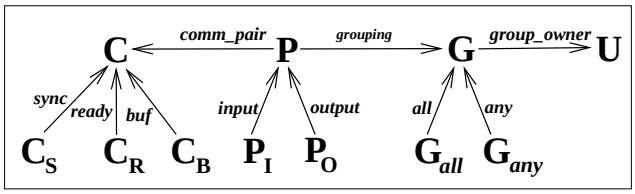

Figure 2: $G_{\mathcal{H}}$ (Graph of Sketch $\mathcal{H}$)

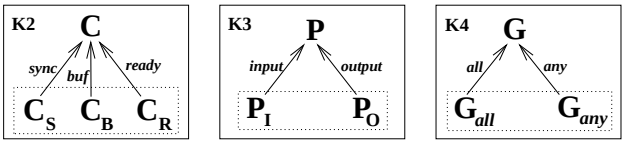

Figure 3: $L_{\mathcal{C}}$ (Cocones of Sketch $\mathcal{C}$)

abstractions are introduced.

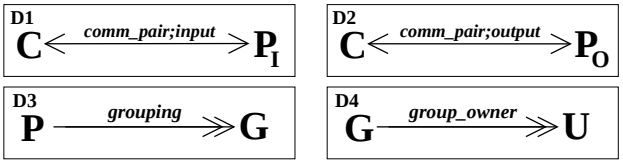

Figure 4: $D_{\mathcal{C}}$ (Diagrams of Sketch $\mathcal{C}$)

3. THE CATEGORICAL # MODEL

For better understanding of this section, knowledge about basic category theory and graph theory concepts are required. However, some intuition behind the introduced formal concepts will be provided whenever possible. The concepts of *sketch* [2] and *institution* [8] are the only advanced categorical concept employed herein. Sketches were firstly proposed for the specification of certain mathematical structures. They play the same role as traditional techniques from first-order logic and universal algebra, but they seem to be more appropriate for dealing with multi-sorted structures and with models in categories other than sets. Institutions were proposed by Goguen and Burnstal for providing a unified theory for algebraic specification systems, which in general differs by the underlying logical system used for expressing properties.

3.1 The Category of Units

The sketch $\mathcal{H}$ is defined by $(G_{\mathcal{H}}, D_{\mathcal{H}}, \emptyset, K_{\mathcal{H}})$. The graph $G_{\mathcal{H}}$, the diagrams $D_{\mathcal{H}}$, and the cocones $K_{\mathcal{H}}$ are presented in Figures 2, 4, and 3, respectively. A # component is defined by a *model* (sketch homomorphism) of the sketch $\mathcal{H}$ on SET, the category of sets, satisfying the commutative diagram in Figure 5, where UNIT is the category of units, defined further, and M, M' are # components. Therefore, a homomorphism μ between # components M and M' is a homomorphism of models, defined by a natural transformation $\mu : M \rightarrow M'$. The category of # components, named HASH, has components as objects and homomorphisms between components as morphisms. As usual for categories of functors (components are functors), the vertical composition of natural transformations defines composition in HASH.

Let M be a component. $M(\mathbf{U})$ is a set of *units*. $M(\mathbf{G})$ is a set of references to *groups of ports*. $M(\mathbf{P})$ is a set of *ports*. $M(\mathbf{C})$ is a set of *communication channels*. $M(\mathbf{C_B})$, $M(\mathbf{C_R})$, and $M(\mathbf{C_S})$ are sets of *buffered*, *ready* and *synchronous* channels, respectively. The Cocone $K1$ states that $M(\mathbf{C_B})$, $M(\mathbf{C_R})$, and $M(\mathbf{C_S})$ are disjoint subsets of $M(\mathbf{C})$. $M(\mathbf{G}_{any})$ and $M(\mathbf{G}_{all})$ are sets of references to groups of ports of kind *any* and *all*, respectively. The Cocone $K3$ yields that $M(\mathbf{G}_{any})$ and $M(\mathbf{G}_{all})$ are disjoint subsets of $M(\mathbf{G})$. $M(\mathbf{P_I})$, and $M(\mathbf{P_O})$ are sets of *input* and *output* ports, respectively. The Cocone $K2$ states that $M(\mathbf{P_I})$, and $M(\mathbf{P_O})$ are disjoint subsets of $M(\mathbf{P})$. The function $M(\textbf{\textit{grouping}})$ associates ports to

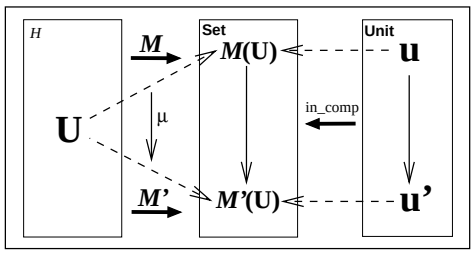

Figure 5: Sketch Restriction

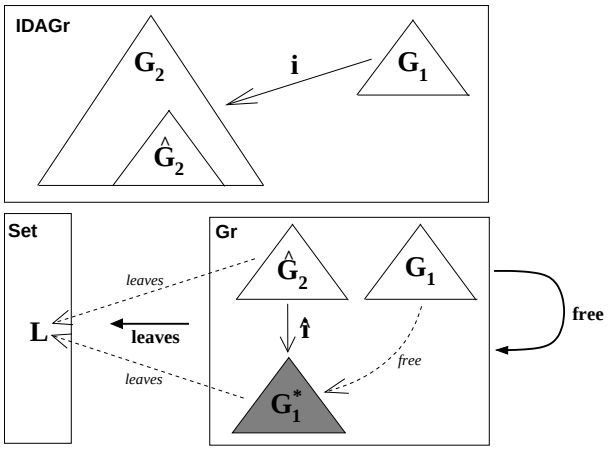

Figure 6: The functor i_G

their groups of ports. Diagram $D3$ is an epimorphism, forbidding empty groups. The function $M(\textbf{\textit{group\_owner}})$ defines the unit for which a group of ports belongs to. Diagram $D4$ ensures that it is an epimorphism too, forbidding units with an empty set of ports. The function $M(\textbf{\textit{comm\_pair}})$ defines the channel where a port is a communication pair. Together, $D1$, $D2$ and $K3$ ensure that channels are unidirectional. The restriction in Figure 5 ensures that the mapping between units induced by HASH-homomorphisms obeys morphisms in the category of units (UNIT).

The next sections define categories for *units* and *interfaces*. The relation between interfaces and units resembles the relation between signatures and algebras in universal algebra. For this reason, an *institution* will be employed for characterizing the relation between *interface signatures* and *units*.

3.2 The Category of Interface Signatures

The objects of category INTERFACESIG represent *interface signatures*. They are defined as tuples $\langle G, E, \nu \rangle$. G is a finite, connected, directed, and acyclic graph $\langle V, A, \partial_0, \partial_1 \rangle$ with exactly one root node. Graphs of this kind are referred as IDAG's. *Leave nodes* represent ports and *branch nodes* represent *interface slices*. The category of IDAG's is IDAGR, with a special notion of morphism that will be defined further on. As usual, GR denotes the category of graphs. E is the set of *exposed nodes* of G ($E \subset \textbf{nodes}(G)$). **Each path in G must have exactly one exposed node**. $\nu : V \to \mathbb{N}$ is a total function that maps nodes of G onto a *stream nesting factor*.

Let I_1 and I_2 be interface signatures. A morphism $\imath : I_1 \to I_2$, in INTERFACESIG, is defined by a tuple $\langle i_G, i_E, i_\nu \rangle$. The IDAGR-morphism $i_G : G_1 \to G_2$ maps G_1 to a branch (also a IDAG) of G_2, called $\hat{G}_2$, in such way that there is an GR-morphism $\hat{\imath}_G : \hat{G}_2 \to \textbf{free}(G_1)$ that preserves leaves, e.g. $\textbf{leaves}(\hat{G}_2) = \textbf{leaves} \circ \textbf{free}(G_1)$. This is illustrated in Figure 6. The function i_E maps exposed nodes from the interface signatures. It must satisfy $i_E(E_1) \subseteq E_1$, which is a sufficient condition to ensure the preservation of exposed nodes between I_1 and I_2. $i_\nu(\nu_1) = \nu_2$, satisfying the commutativity of the diagram in Figure 7 (preservation of stream nesting factors).

3.3 The Institution of Units

In terms of the theory of institutions, a unit is essentially a model for an interface signature, as well as Σ-algebras are models for an algebra signature Σ. Units augment interface signatures with a notion of *behavior*, defined by a protocol that generates a formal language whose alphabet is composed by the set of exposed nodes of the interface signature of the unit.

Let I be an arbitrary interface signature. The functor Sen : INTERFACESIG $\to$ SET maps I to the set E^* (Kleene closure of E). A unit is defined by a tuple $\langle P, R, \delta, \pi \rangle$, where P is a set of ports, R is a set of slice references, $\delta : P \to \{input, output\}$ is a total function defining direction of ports, and π is a protocol. A protocol expression is defined by the syntactic class Π, whose definition is

$$\begin{aligned}\Pi_i ::= {} & \textbf{seq}\ \{\Pi_1, \Pi_2, \ldots, \Pi_k\} \mid \textbf{par}\ \{\Pi_1, \Pi_2, \ldots, \Pi_k\} \mid \\ & \textbf{alt}\ \{\Pi_1, \Pi_2, \ldots, \Pi_k\} \mid p_i? \mid p_o! \mid \textbf{do}\ \mathrm{r} \mid s+ \mid s-\end{aligned}$$

where $p_i \in \{p \in P \mid \delta(p) = Input\}$, $p_o \in \{p \in P \mid \delta(p) = Input\}$, $r \in R$, and s is a semaphore symbol. The protocol combinators **seq**, **par** and **alt** denote respectively *sequence*, *concurrency* and *alternative*. The symbols ! and ? are the basic primitives for sending and receiving data in distributed synchronization channels, respectively. The primitive **do**, the *unfolding* primitive, denotes the protocol underlying the slice reference r. It can be viewed as a macro expansion operator. The symbols + and - mean the usual signal (V) and wait (P) semaphore primitives. Protocols of units may generate a terminal Petri net formal language on the alphabet P [9]. The motivation for Petri nets is to build dynamic models for the behavior and interaction of units, making possible the analysis of formal properties and performance evaluation of programs using Petri net tools and their variants. The formal language generated by a protocol π is denoted by $\Lambda(\pi)$.

A unit U complies with an interface I if the following condition holds: (1) $P = E \cap \textbf{leaves}(G)$ (ports are exposed leaf exposed nodes of G), and (2) $R = E - \textbf{leaves}(G)$ (slice references are exposed non-leaf nodes of G). The covariant functor Mod : INTERFACESIG $\to$ CAT^{op} maps each interface signature I onto the category of units that complies with I.

Let I be an interface signature. The relation $\models_I$: $Sen(I) \times Mod(I)$ associates units with the activation sequences supported by their protocols. For instance, let $w \in Sen(I)$ be a

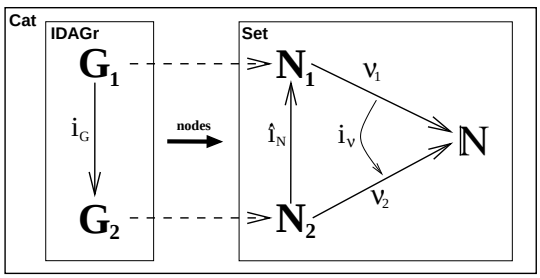

Figure 7: Commutative Diagram for i_ν

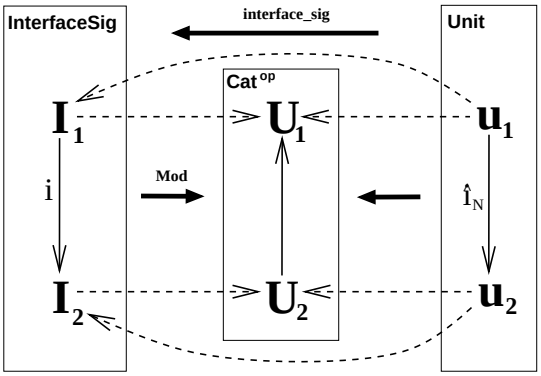

Figure 8: The Category of Units

word in E^*, U be a unit, and π be the protocol of U. Then, $U \models_I w$ iif $w \in \Lambda(\pi)$.

The category InterfaceSig, the functors Sen and Mod and the relation $\models_I$, as defined before, forms an **institution**. In fact, for some morphism $i : I \rightarrow I'$ in InterfaceSig, the required *satisfaction condition* $u' \models_{I'} Sen(i)(w) \Leftrightarrow Mod(i)(u') \models_I w$ holds for each $u' \in |\ Mod(I')\ |$ and for each $w \in Sen(I)$

3.4 The Category of Units

The *institution* of units splits the category of units in classes of units that complies to the same signature. Also, it naturally expresses the unusual **behavior preservation property between units**. The functor **interface_sig** : Unit $\rightarrow$ InterfaceSig maps units to their interfaces. The commutative diagram in Figure 8 must be satisfied.

3.5 Composition of Components

The category HashFDDiag has all discrete diagrams in Hash as *objects* and the graph homomorphisms between them as *morphisms*. Let $h : \mathbf{D}_1 \rightarrow \mathbf{D}_2$ be a morphism in HashFDDiag , and let $\langle C_1, C_2 \rangle$ be a pair of nodes in $\mathbf{D}_1$ and $\mathbf{D}_2$, respectively. Then $h(C_1) = C_2$ implies that $f : C_1 \rightarrow C_2$ is a morphism in HashFDDiag.

Let $\boldsymbol{D}$ be a HashFDDiag-object formed by n components. $\boldsymbol{D}$ is the *start diagram* for overlapping them. The vertex of its co-limit is conventionally called M_0^D. For the sake of simplicity, M_0 may be used instead of M_0^D whenever this does not cause confusion. M_0 is called *initial component* of $\boldsymbol{D}$, obtained from the disjoint overlapping of the components in $\boldsymbol{D}$. From M_0, new components are formed by applying the composition operations *unification*, *factorization*, *replication* and *superseding*. The functor ***overlap*** : HashFDDiag $\rightarrow$

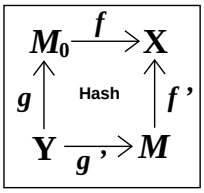

Figure 9: A Commutative Diagram for *overlap*

Cat associates a diagram $\boldsymbol{D}$ with the sub-category of Hash containing all objects M such that there are epimorphisms $f : M_0 \twoheadrightarrow X$, $f' : M \twoheadrightarrow X$, $g : Y \twoheadrightarrow M_0$, and $g' : Y \twoheadrightarrow M$, for some pair of objects X and Y, such that the diagram in Figure 9 commutes.

4. CONCLUSIONS

This paper introduced a categorical interpretation for the compositional features of # component model. Further works will use the formal framework introduced herein for formalizing concepts and proving properties regarding the structure of # components. Another source of work is to study the expressiveness of component models, by mapping # components onto components from other component models using functors and natural transformations. In fact, this is the main motivation for adopting category theory as an underlying mathematical foundation.

5. REFERENCES

[1] R. Armstrong et al. Towards a Common Component Architecture for High-Performance Scientific Computing. In *The Eighth IEEE International Symposium on High Performance Distributed Computing*. IEEE Computer Society, 1999.

[2] M. Barr and C. Wells. *Category Theory for Computing Science*. Prentice Hall, 1990.

[3] F. Baude, D. Caromel, and M. Morel. From Distributed Objects to Hierarchical Grid Components. In *International Symposium on Distributed Objects and Applications*. Springer-Verlag, 2003.

[4] Bernholdt D. E. 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)*, pages 76–84. Madrid, Spain, 2004.

[5] F. H. Carvalho Junior and R. D. Lins. The # Model for Parallel Programming: From Processes to Components with Insignificant Performance Overheads. In *Workshop on Components and Frameworks for High Performance Computing (CompFrame 2005)*, June 2005.

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

[7] J. Dongarra, *et al. Sourcebook of Parallel Computing*. Morgan Kauffman Publishers, 2003.

[8] J. Goguen and R. Burnstal. Institutions: Abstract Model Theory for Specification and Programming. *Journal of ACM*, 39(1):95–146, 1992.

[9] T. Ito and Y. Nishitani. On Universality of Concurrent Expressions with Synchronization Primitives. *Theoretical Computer Science*, 19:105–115, 1982.

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