Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Electronic Notes in Theoretical Computer Science 260 (2010) 47–72
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Compositional Specification of Parallel Components Using *Circus*

Francisco Heron de Carvalho-Junior[1,2]

Departamento de Computação
Universidade Federal do Ceará
Fortaleza, Brazil

Rafael Dueire Lins[3]

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

Abstract

The # (hash) component model aims to take advantage of a component-based perspective of software for the development of high-performance computing applications targeted at parallel distributed architectures. This paper presents an approach for specifying #-components using *Circus*, to provide the ability of reasoning about behavioral and functional properties of #-components and their composition, as well as to partially generate code of their units through the application of successive semi-automatic refinement steps. On the *Circus* side, the # component model provides a new compositional approach to combine a *Circus* specification to form new ones, widening its applicability.

1 Introduction

The dissemination of parallel architectures, such as clusters, grids and multi-core processors rapidly increased the widespread interest in high performance computing (HPC) applications. Thus, such platforms attracted the investments of the software industry. Today, peak performance demands of programmers a good knowledge of HPC techniques for parallel and distributed programming tuned with computer architectures. This narrows the possibility of the development of general purpose parallel programming in widespread platforms.

[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.2009.12.031

The success of the technology of components in the commercial scenario [37] yielded new component models and frameworks for HPC applications, such as CCA and its compliant frameworks [5], Fractal/ProActive [10], and P-COM [28]. However, there are still difficulties to be overcome, such as the development of a general notion of parallel component and more suitable connectors for efficient parallel synchronization.

The # component model was developed to improve the practice of developing HPC parallel software. The ability to be deployed in a pool of computing nodes of a parallel platform and addressing non-functional concerns are inherent to the so called #-components. Based on a framework architecture recently proposed [13], a # programming system called HPE (*The Hash Programming Environment*) was designed and prototyped on top of the notion of #-components.

The implementation of parallel programs is considered an error prone task. In particular, to deal with synchronization bugs is an important aspect of programming with low level message passing libraries like MPI, mainly when programming for scientific and engineering application domains, due to the complexity of interactions in optimized implementations of high-level mathematics involved in simulations. Moreover, it is also very important to ensure the correctness of computations, which are not trivial for non-specialist programmers. The authors believe that formal methods may constitute an important tool for addressing these issues.

This paper presents an approach for the specification of #-components using *Circus*, a language for behavioral and functional specification of concurrent programs that supports code generation by semi-automatic refinement steps. The reasons to adopt *Circus* are presented throughout this paper. This is an initial step to integrate existing tools for working with *Circus* specifications (model checking, refinement, code generation, and type checking) [3] with HPE, providing support for translating behavioral parts of specifications of #-components using *Circus* onto Petri nets. This integration will provide an environment for the analysis of formal properties, performance evaluation, and the safe implementation of parallel programs. On the side of *Circus*, this paper contributes with a new modularization technique, orthogonal to processes, intended for incremental building of large scale specifications written using this formalism.

The Structure of the Paper

The # component model introduces a number of new concepts attempting to reach expressiveness for describing parallel programming abstractions and to provide the necessary level of abstraction to be independent of parallel platforms and parallelism-enabling infrastructures on top of them. By assuming that probably readers are not familiar with some of these concepts, and despite the fact that this is not the first paper to introduce the # component model [15,12,16,13], the Section 2, devoted to describe and formalize it, occupies a substantial space in this paper. It firstly introduces the notion of #-component by assuming some knowledge of the reader about basic structure of parallel programs as a set of interacting processes. For that, it is used a simple example from which #-components are ex-

tracted by using process slicing by concerns. Then, in Section 2.1, now assuming the reader knowledge about existing component models, the # component model is briefly compared to other component models in terms of expressiveness. Section 2.2 informally provides a more general perspective of overlapping composition, preparing the reader to understand HOCC (Hash Overlapping Composition Calculus) in Section 2.3, which is intended to define overlapping composition formally. HOCC abstracts away concepts that are dependent on particular # programming systems, which are defined in Section 2.4, by taking an arbitrary category, named U, in its definition. One may define this category in order to instantiate a formal model of a particular # programming system. This is the approach used in Section 3 to define a specification language for #-components on top of *Circus*. A background on basic concepts of category theory that are used in this paper and references to the additional literature on this subject are provided in Appendix A. The reason to not include category theory background in the main body of the paper is that the authors do not assume deep knowledge about category theory to read this paper, by providing additional explanation about the use of categoric concepts. Section 2.5 attempts to present the reasons that motivated the authors to propose the use of *Circus* for specification of #-components. Finally, Section 2.6 presents an example of parallel programming design using #-components whose specification will be presented in Section 4 for exemplifying the approach proposed in Section 3. Section 3, which describes the contribution of this paper as pointed out before, shows how # components can be specified using *Circus*. For that, it would be sufficient to define the category U, since HOCC already defines the interpretation of overlapping composition operations over such category. However, for better understanding, Section 3 describes intuitively the interpretation of overlapping composition operations in *Circus* specifications, also proposing syntactical extensions to *Circus* for supporting overlapping composition. Section 4 provides an example of specification for the example presented in Section 2.6 and using the approach presented in Section 3. Section 5 concludes this paper, discussing previous research with # programming systems that motivates the adoption of *Circus*, and points at lines for further works.

2 The # Component Model

Parallel components have been proposed for several computational frameworks for developing applications in high-performance computing (HPC) [37]. Most of those frameworks are derived from existing component models successfully applied by the software industry, which are not concerned with parallel processing in their design, or component models specifically designed to the needs of HPC, such as CCA [5] and Fractal [9], also based on existing component models. They introduce new features to enable the description of a limited set of patterns of parallelism. For the sake of simplicity, these features are completely orthogonal to the underlying component model, in such a way that component infrastructures stays "out of the way" with parallelism. Such approaches have not reached the level of expressiveness and efficiency of message passing libraries such as MPI, making the search for more

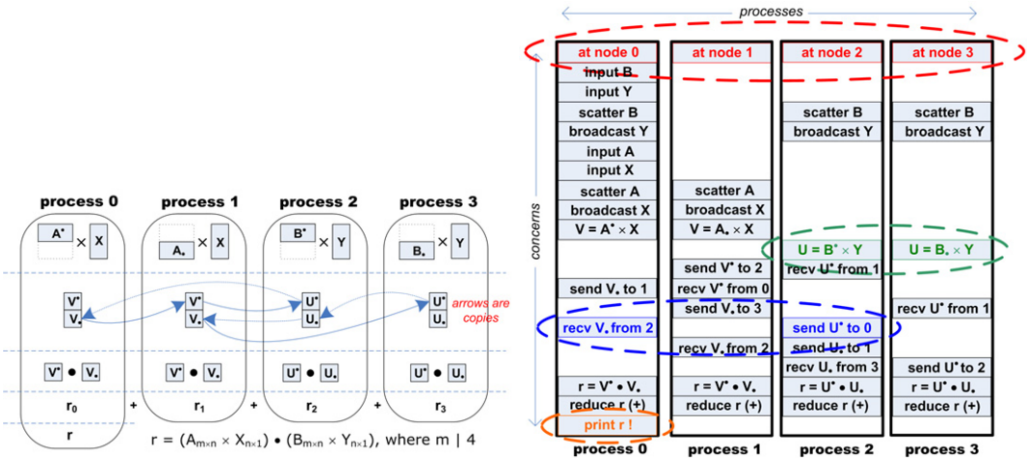

Fig. 1. From Processes to #-Components

expressive parallelism with components an important research theme for those who work with CCA and Fractal [1,8,6]. The # component model proposes a notion of components that is intrinsically parallel and shows how they can be combined to form new components and applications, recursively. A #-component may be seen as a generalization of the usual notions of component, where a component is formed by a set of component parts, called *units*, each one deployed in a node of a parallel computer. Usual component models assume that a component is a software entity that resides in a single address space.

Figure 1 intends to provide a naive notion of #-components by assuming the knowledge of the reader about the basic structure of parallel programs, viewed as a set of interacting processes. For that, it is used a parallel program that calculate the linear algebra operation

$$A \times \widehat{x} \bullet B \times \widehat{y}$$

, where $A_{m\times n}$ and $B_{m\times k}$ are matrices and $\widehat{x}_{n\times 1}$ and $\widehat{y}_{k\times 1}$ are vectors. For that, the parallel program is formed by N processes that are coordinated in two groups, named p and q, with M and P processes, respectively. In Figure 1, $M = P = 2$, $p = \{\textbf{process 0}, \textbf{process 1}\}$ and $q = \{\textbf{process 2}, \textbf{process 3}\}$. In the first stage of the computation, the processes in group p calculate the matrix-vector product $\widehat{v} = A \times \widehat{x}$, while the processes in group q calculate $\widehat{u} = B \times \widehat{y}$, where $\widehat{v}_{m\times 1}$ and $\widehat{u}_{m\times 1}$ are intermediary vectors. Figure 1(a) illustrates the partitioning of matrices and vectors and the messages exchanged. $M^{\bullet}$ denotes the upper rows of the matrix M, where $M_{\bullet}$ denotes their lower rows. The definition is analogous for vectors, by taking them as matrices with a single column. Thus, the input matrices A and B are partitioned by rows, while the input vectors $\widehat{x}$ and $\widehat{y}$ are respectively replicated across the processes in groups p and q. Thus, after the first stage, the elements of the result vectors $\widehat{v}$ and $\widehat{u}$ are respectively distributed across the processes in groups p and q. Since it is necessary to calculate the dot product $\widehat{v} \bullet \widehat{u}$ using all the N processes, it is a good practice to improve data locality by distributing vectors $\widehat{v}$

and $\widehat{u}$ across all the N processes. This is performed in the second stage, where the arrows denote exchange of messages between processes. Finally, the dot product is calculated in the third stage, returning the resulting scalar by summing the partial results calculated by each process.

In Figure 1(b), the processes that form the parallel program described in the last paragraph are sliced according to the notion of *software concern*, whose definition vary broadly in the literature [30]. For the purposes of this paper, it is sufficient to take a concern as anything about the software that one wants to be able to reason about as a relatively well-defined entity. Software engineers classify concerns in functional and non-functional ones. The former ones define the units of functionality of the software, while the later ones are related to any aspect that affect the performance of the software. A slice of a process will correspond to the piece of its implementation related to a concern that is considered relevant to reason about. In the parallel program of the example, the relevant concerns include synchronization, communication and computation operations and allocation of processes onto processors. Most of them involve the participation of slices of many processes, such as the four slices that define allocation of processes to processors, the two slices of processes 2 and 3 that perform the matrix-vector product $U = B \times Y$ in parallel, and that ones defining communication channels (*send* and *recv* pairs). Such teams of cooperative slices define the units of #-components. By cooperative, we mean that each slice in a team does not constitute a complete concern. It does not make sense to think about such slices as isolated entities. Only together they define complete software concerns that are intended to be addressed by #-components. In SCMD (Single Component Multiple Data) programming, CCA frameworks uses the abstraction of *cohort of components* to implement the notion of cooperative slices, but they keep implementation of parallel interaction between components of a cohort encapsulated inside the components. Thus, such components are not independent of each other, breaking an important principle of components. In Figure 1(a), candidates to be #-components are represented by the dashed ellipses. Thus, a *unit* defines the *role* of a process with respect to the *concern* addressed by the #-component. The example also shows that #-components can deal with *non-functional* concerns, such as mapping of processes onto processors. #-components may be recursively combined by overlapping composition, discussed in Section 2.2.

Compared to the traditional practice of parallel programming, a # parallel programmer works at the perspective of *concerns*, while a traditional one works at the perspective of *processes*, tending either to encapsulate individual slices in modules, separated from their cooperating slices, or to encapsulate cooperating slices in a single module, calling the appropriate slice according to the process identification like in SPMD style. Both are supported in CCA frameworks that implement SCMD programming, but break down modularization principles as pointed out in the last paragraph. The authors advocate that having *processes* and *concerns* in the same dimension of the software decomposition process make hard to harmonize software engineering and parallel programming [15].

2.1 Comparison to Other Component Models

What are the fundamental differences between the # component model and other models ? Essentially, a #-component offers a more general notion of component than usually found in traditional component models. It makes neither assumption on the concrete nature of the components nor on the connectors used to bind them. CCA, for instance, requires that components implement certain interfaces, expressed by a specific language called SIDL (Scientific Interface Definition Language). Some of these interfaces are required for accessing the services of the CCA framework. Other are required for connecting components by means of *bindings* between *uses ports* and *provides ports*. Thus, a connector between CCA components is a binding between ports with the same interface, like in CORBA, Fractal, and most of component models. The # component model gives the responsibility of defining the concrete nature of components and connectors to the # programming systems, which must define a set of *component kinds* to classify #-components according to their intended meaning, as usual component models do by defining a single component kind. Component kinds may be viewed as a *domain specific language* (DSL) if the # programming system is designed with a specific application domain in mind. Moreover, connectors may be taken as kinds of #-components in a # programming system, making possible that programmers define new primitive connectors from scratch or by composing other connectors, like proposed by Reo[4]. This is possible because, in a distributed environment, a connector may be viewed as a software entity whose intent is to connect components that reside in disjoint address spaces, a notion that is supported by a #-component in a natural way. Hypothetically, it is possible to support components of many component infrastructures, based on different component models, in a # programming system, by taking their specific notions of components and connectors as isolated component kinds.

2.2 Overlapping Composition of #-Components

The example presented in the introduction of Section 2 only attempts to provide some intuition about the slicing of processes of a parallel program for decomposing it in concerns. This is the main principle behind the notion of #-component. In fact, the example helps us to show how to move from a process-based decomposition of a parallel program, that is the current practice in parallel programming, onto a concern-oriented based decomposition, that is closer to software engineering artifacts. In this section, the opposite direction is taken by informally introducing a model of #-components that supports the basic principles described in the example and from which other properties about the model can be extracted.

A #-component is composed by a finite set of units, whose cooperation defines the concern addressed by the #-component. #-components can be composed hierarchically, using *overlapping composition*, forming a new #-component. Figure 2 illustrates the notation used to represent #-components, as ellipses, and their units, as rectangles. It illustrates the concepts introduced in this paragraph. In a configuration of a #-component, the #-components to be composed are called *direct inner*

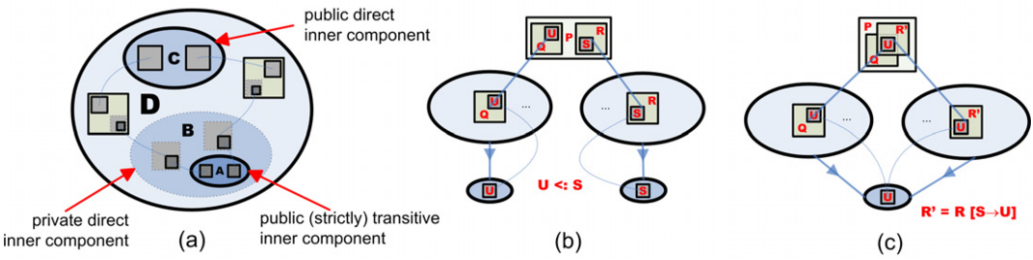

Fig. 2. #-Components

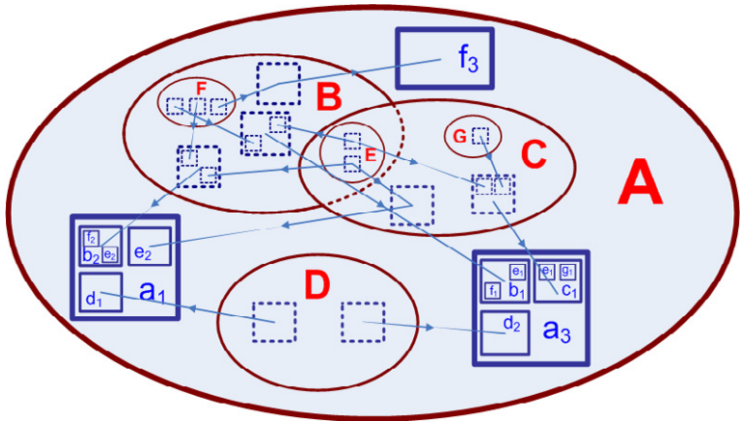

Fig. 3. Deep Hierarchy in Overlapping Composition of a #-Component

components. The units of the new #-component are obtained by *folding* units of its direct inner components. A set of folded units becomes the slices of the unit of the new #-component. A unit of some direct inner component that was not *folded* is said to be *lifted*. It becomes a unit of the new #-component. #-components that are formed by overlapping other #-components are called *composite* ones. Otherwise, they are called *primitive*. In overlapping composition, a #-component is a *transitive inner component* of a #-component C if it is a *direct inner component* of C or a *transitive inner component* of some direct inner component of C. It is *strictly transitive* whenever it is not a direct inner component of C. It is also useful to classify inner components as *public* and *private*. Public inner components of a #-component C are *accessible* in the configuration of a #-component C' where C is a direct inner component. A public inner component may be set to be private, becoming *inaccessible* in a configuration where the #-component enclosing is used. Accessibility of strictly transitive inner components in a configuration is necessary to make possible the configuration of their sharing between direct inner components, as explained with help of an example in the next paragraph.

Figure 3 presents an illustrative overlapping composition of #-components **A**, **B**, **C**, **D**, **E**, and **F**. **A** is a composite, formed by overlapping composition of **B**, **C**, and **D**. **B** and **C** are also composites, formed respectively by primitives **C** and **E**, and **F** and **E**. Notice that **E** is a shared inner component of **B** and **C**. By definition, two inner components are shared if, and only if, their corresponding units are shared

$$
\begin{array}{lll}
\texttt{t} ::= & x & \\
& \lambda x{:}\texttt{T}.\ \texttt{t} & \textit{abstraction} \\
& \texttt{t}\ \texttt{t} & \textit{application} \\
& \mathbf{join}_{\kappa}\ \texttt{t}\ \texttt{t} & \textit{joining} \\
& \mathbf{fold}\ \oplus\ u_1\ u_2\ \texttt{t} & \textit{unification} \\
& \langle \mathcal{U}, \kappa \rangle\,,\ \mathcal{U} \subset \texttt{obj}\,(\mathsf{U}) & \textit{\#-component} \\
 & & \\
\texttt{v} ::= & \lambda x{:}\texttt{T}.\ \texttt{v} & \textit{abstraction} \\
& \langle \mathcal{U}, \kappa \rangle\,,\ \mathcal{U} \subset \texttt{obj}\,(\mathsf{U}) & \textit{\#-component}
\end{array}
$$

Fig. 4. HOCC - The # Overlapping Composition Calculus (Syntax)

in units of the enclosing configuration. Figures 2(b) and 2(c) attempt to show how sharing of slices can be obtained between units Q and R. In Figure 2(b), units U and S are slices of units Q and R, respectively. Also, U is a refinement of S (U <: S). For this reason, U may supersede S in R, forming R' in the configuration of Figure 2(c), where U is now a slice of units of Q and R'. The concept of refinement is left abstract here. It is only a way to say that it is safe to use unit U in the context where unit S is being used. A concrete notion of refinement is defined by the # programming system. Well known examples of refinement relations are subtyping relations between objects in a object-oriented language or refinement relations between specifications in process algebras and specification languages. In particular, further sections will show that the work described in this paper is interested in refinement notions between *Circus* processes.

2.3 HOCC - A Calculus for the Overlapping Composition of #-Components

Figure 4 presents the syntax of a calculus of terms to formalize the overlapping composition of #-components, called the *# overlapping composition calculus* (HOCC).

A *term* (metavariable t) denotes a *configuration*. The terms *variable*, *abstraction*, and *application* borrow their names and meaning from the λ-calculus. The terms *joining* and *folding* define the basic operations in the *overlapping composition* of configurations. The term *#-component* is defined as a pair, where $\mathcal{U}$ denotes a finite set of units and k denotes the kind of the #-component. The elements of $\mathcal{U}$ are objects of the category U, whose objects are units and whose arrows are unit homomorphisms. The category U is left abstract, being concretely defined by the # programming system. The element κ belongs to $\mathbb{K}$, denoting the kinds of #-components supported by the # programming system. The relation $\mathfrak{I} : \mathbb{K} \times \mathbb{K}$, has elements $\kappa_1 \Rrightarrow \kappa_2$, for $\kappa_1, \kappa_2 \in \mathbb{K}$, which define that a #-component of kind κ_1 may be a inner component of a #-component of kind κ_2.

Let u_1 and u_2 be units of a #-component, probably obtained by joining two #-components C_1 and C_2, respectively owners of u_1 and u_2. Their roles can be combined in the application of a term **fold** with a folding operator $\oplus$, defining a new unit in the new #-component. In the category U, $u = u_1 \oplus u_2$ is defined by the colimit of a given *commutative diagram* D including u_1 and u_2, where u is the vertex

of the colimit. More intuitively, u is the "simplest" unit that satisfies the property defined by D. In fact, since D includes both u_1 and u_2, u preserves the structures of units u_1 and u_2 in its constitution. This is illustrated in the *commutative diagrams* D_1 and D_2 of Figures 5(a) and 5(b) for the folding examples of Figures 2(b) and 2(c), respectively. In Figure 5(a), units Q and R are folded without sharing using the discrete commutative diagram D_1 that includes them. In Figure 5(b), Q and R are folded by sharing units U and S, which is possible due to the morphism from S to U in D_2, denoting the refinement relation U <: S of Figure 2(b).

In a # programming system, the category U could be defined in such way that its objects are *classes* of *objects* in some object-oriented language and $\oplus$ is a language constructor to form a new *class* which has *objects* of the operand *classes* as *properties*. In such case, the units of a #-component are *objects* and the configuration of this #-component define their units as *classes*. In the current implementation of HPE, such language can be any language supported by the Mono/.NET platform. For the aims of this paper, Section 3.1 defines that objects of U are *Circus* processes and that morphisms define a particular refinement relation between these processes. Thus, $\oplus$ combines processes u_1 and u_2 to form a new *Circus* process. Section 3 shows how #-components can be specified using *Circus* and the semantics of overlapping composition of *Circus* specifications.

Figure 6 presents a *call-by-value* evaluation semantics for configurations, by

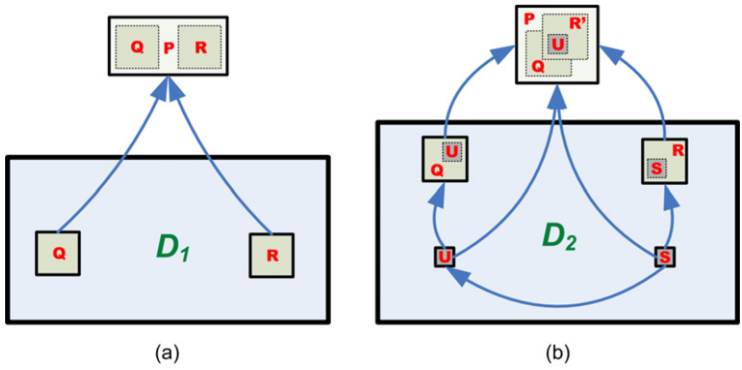

Fig. 5. A Categorical Perspective of Folding Two Units - (a) Without Sharing and (b) With Sharing

$$\frac{\mathtt{t}_1 \to \mathtt{t}'_1}{\mathtt{t}_1\ \mathtt{t}_2 \to \mathtt{t}'_1\ \mathtt{t}_2}\ \textit{(E-App1)} \qquad \frac{\mathtt{t}_2 \to \mathtt{t}'_2}{\mathtt{v}_1\ \mathtt{t}_2 \to \mathtt{v}_1\ \mathtt{t}'_2}\ \textit{(E-App2)} \qquad \frac{\mathtt{t}_1 \to \mathtt{t}'_1}{\mathbf{fold}\ \oplus\ u_1\ u_2\ \mathtt{t}_1 \to \mathbf{fold}\ \oplus\ u_1\ u_2\ \mathtt{t}'_1}\ \textit{(E-fold1)}$$

$$(\lambda x{:}\mathtt{T}.\mathtt{t}_{12})\ \mathtt{v}_2 \to [x \longmapsto \mathtt{v}_2]\ \mathtt{t}_{12}\ \textit{(E-AppAbs)} \qquad \frac{\mathtt{t}_1 \to \mathtt{t}'_1}{\mathbf{join}_\kappa\ \mathtt{t}_1\ \mathtt{t}_2 \to \mathbf{join}_\kappa\ \mathtt{t}'_1\ \mathtt{t}_2}\ \textit{(E-Join1)} \qquad \frac{\mathtt{t}_2 \to \mathtt{t}'_2}{\mathbf{join}_\kappa\ \mathtt{v}_1\ \mathtt{t}_2 \to \mathbf{join}_\kappa\ \mathtt{v}_1\ \mathtt{t}'_2}\ \textit{(E-Join2)}$$

$$\{\kappa_1 \Rrightarrow \kappa, \kappa_2 \Rrightarrow \kappa\} \subseteq \mathfrak{I}\ \ \mathbf{join}_\kappa\ \langle \mathcal{U}_1, \kappa_1 \rangle\ \langle \mathcal{U}_2, \kappa_2 \rangle \to \langle \mathcal{U}_1 \cup \mathcal{U}_2, \kappa \rangle\ \textit{(E-Join3)} \qquad \frac{u = u_1 \oplus u_2}{\mathbf{fold}\ \oplus\ u_1\ u_2\ \langle \mathcal{U}, \kappa \rangle \to \langle (\mathcal{U} - \{u_1, u_2\}) \cup \{u\}, \kappa \rangle}\ \textit{(E-fold2)}$$

Fig. 6. HOCC - The # Overlapping Composition Calculus (Semantics)

using a notation inspired by [33]. It shows how to build a #-component from a configuration by applying overlapping composition operations.

2.4 The # Programming Systems

A # programming system interprets #-components and their comprising units in terms of the usual units of software composition (ex: classes, interfaces, abstract data types, configuration files, and so on). It defines a finite set of supported *component kinds* as abstractions for building blocks of applications in some domain that the # programming system targets. In fact, a # programming system makes concrete the definitions of the category U, the set $\mathbb{K}$, and the relation $\mathfrak{I}$ discussed in the previous section.

As a comparison, usual component models might be interpreted as # programming systems with only one component kind defined as a particular notion of software module (a .NET assembly, a Java Bean, a CCA component, for example). A # programming system must provide a library of primitive #-components and/or facilities to build them from scratch according to the semantics of the different kinds of components.

HPE (*Hash Programming Environment*) is a # programming system that is being implemented as a plug-in to the IBM Eclipse Platform. HPE is being instantiated for general purpose parallel programming targeting at clusters of multiprocessors, supporting seven component kinds: *computations*, *data structures*, *synchronizers*, *architectures*, *environments*, *applications*, and *qualifiers* [13].

The specification approach proposed by this paper may be applied to a subset of kinds supported by a # programming system whose #-components may be specified using *Circus*. It includes those kinds of functional #-components, but it may also include some kinds of non-functional ones whose #-component may also be specified using *Circus* since they also involve computations. For example, a functional #-component C that is intended to solve a system of linear equations using some iterative method may be overlapped to a non-functional #-component C' which concurrently perform some operations over the data structures of C to accelerate its convergence to the solution. Despite C' is implemented as computations over a data structure that is shared with C, C' is not conceptually a functional #-component since it exists only to affect performance of another functional #-component.

2.5 Behavior Protocols and Exogenous Coordination: Motivating Circus

In the design of # programming systems, it is usual to classify units of some kinds of #-components as *actions*, such as units of synchronizers and computations in HPE. Action units denote operations that must be performed in some partial order. A #-component may have action and non-action units in its constitution. Moreover, action units may also appear in #-components that address non-functional concerns. Thus, a unit of a composite #-component may be formed by folding a set of action and non-action units that come from its inner components. In our work with # programming systems, since Haskell$_{\#}$ [14], we have adopted the approach

to give to programmers linguistic abstractions to describe the order in which action units are activated, defining the behavior of the #-components exogenously and promoting separation between coordination and computation. For this reason, the #-component model has been presented in previous works as a coordination model for parallel components [11]. Behavior expressions has been adopted as the formalism to describe the order of activation of actions, with semantics in CSP [22] and a possibility of translation onto Petri nets [17]. In order to achieve the expressive power of Petri nets for describing traces of actions, a class of synchronized regular expressions has been adopted [23].

In Haskell$_{\#}$, synchronization between units of a #-component has been defined by means of lazy streams transmitted through unidirectional and strongly typed communication channels [11], making possible a complete behavior description of a Haskell$_{\#}$ component at the *coordination level* that may be translated onto Petri nets, allowing formal proof of behavioral properties using automatic tools like INA and PEP, as well as performance evaluation. Formal reasoning would be also possible at the *computation level*, since Haskell is a pure non-strict functional language. The ability to support formal reasoning about the behavior of components and their interaction at *coordination level* may be supported by specific # programming systems, by using the techniques inherited from Haskell$_{\#}$. However, since any programming language can be used at the *computation level* of # programming systems, it is not possible to think about a complete formal reasoning environment integrating formal descriptions at coordination (behavioral properties) and computation (functional properties) levels in # programming systems. This is the motivation to propose, in this paper, the use of *Circus* for specification of configurations of #-components, since it gives the ability to describe both behavioral and functional aspects of concurrent systems.

2.6 A Simple Example of Parallel Program Built from #-Components

This section outlines an implementation of the example depicted in Figure 1 using #-components, which will be used further on to exemplify the specification of #-components using *Circus*.

Figure 7 presents the hierarchy of components of the #component **app**, of abstract type AppExample, with kind *application*, that implements the parallel program described in the introduction of this section. It is composed by overlapping #-components corresponding to the operations involved in the three steps of the computation. For instance, the #-components **axv** and **byu**, of abstract type MatVecProduct, represent the parallel matrix-vector multiplications. The #-components **rV** and **rU**, of abstract type ScatterMxN, represent the redistribution of the resulting vectors across all processes. Finally, the #-component **vur**, of abstract type VecVecProduct, represents the parallel dot product of the redistributed vectors. **axv**, **byu**, and **vur** are *computations*, while **rV** and **rU** are *synchronizers*. The public inner components of **axv**, **rV**, **rU**, and **vur** represent the *data structures* processed by them. Some of them are shared. The enumerated units **p** and **q** have four slices, corresponding to the units of the inner components

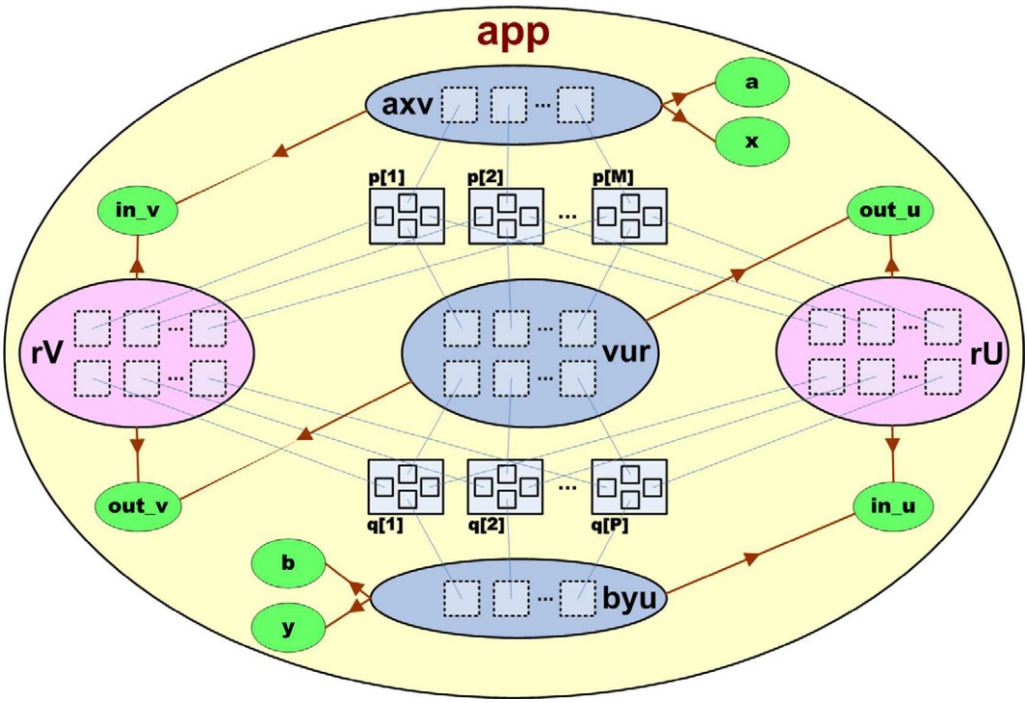

Fig. 7. Hierarchy of Components of the Application Example

axv, **byu**, **rV**, **rU**, and **vur**. Such slices denote actions. For example, the order of activation for slices of processes **p** is defined by the following CSP expression in the configuration of the #-component:

$$\mathbf{axv}.calculate; (\mathbf{rV}.a \;|||\; \mathbf{rU}.b)\,; \mathbf{vur}.calculate$$

. In this expression, the sequential steps, separated by semi-colons, correspond to the three stages described in the introduction of Section 2. Notice that the redistributions of vectors $\hat{v}$ and $\hat{u}$, where all processes participate, may be executed in parallel.

3 #-Components and Circus Specification

Circus [2] was proposed as a unified language for presenting specifications, designs, and programs. It combines Z [38], CSP [22], specification constructs usually found in refinement calculi [31] and Dijkstra's language of guarded commands [18].

A system specified using *Circus* is composed by a set of processes that interact through communication channels, as in CSP. Z schema constructs are used to describe the internal state of a process, which is encapsulated since channels are the only means for processes to communicate with their environments. More formally, as Z, a *Circus* system specification is formed by a list of paragraphs, as formalized in the abstract syntax of Figure 8. A paragraph can be a *Z paragraph* (constants and global types), a *channel* or *channel set* definition, or a *process* declaration.

Program ::= CircusPar*
CircusPar ::= Par | **channel** CDecl | **channel** N == CSExp | **process** $N \triangleq$ Proc
CDecl := SimpleCDecl | SimpleCDecl; CDecl
SimpleCDecl := N^+ | N^+:Exp | Schema-Exp
Proc ::= **begin** PPar* **state** Schema-Exp PPar* • Action **end** | Proc; Proc | Proc□Proc
| Proc ⊓ Proc | Proc⟦CSExp⟧Proc | Proc ||| Proc | Proc \ CSExp | Decl ⊙ Proc
| Proc⌊Exp$^+$⌋ | Process[$N^+ := N^+$] | Decl • Proc||Proc(Exp$^+$) | [N^+]Proc | Proc[Exp$^+$]
PPar ::= Par | $N \triangleq$ Action
Action ::= Schema-Exp | CSPAction | Command
CSPAction ::= *Skip* | *Stop* | *Chaos* | Comm → Action | Pred&Action | Action; Action
| Action□Action | Action ⊓ Action | Action⟦CExp⟧Action | Action ||| Action
| Action\CSExp | μN • Action | Decl • Action | Action(Exp$^+$)
Comm ::= N CParameter*
CParameter ::= ?N | ?N : Predicate | !Expression | .Expression
Command ::= N^+ : [Pred, Pred] | $N^+ := Exp^+$ | **if** GActions **fi** | **var** Decl • Action | **con** Decl • Action
GActions ::= Pred → Action | Pred → Action□GActions

Fig. 8. *Circus* Syntax [35]

A process declaration comprises a *name* and a *process definition*. The most basic form of a process specifies its local state, by means of Z paragraphs; a sequence of paragraphs that define actions, which can be Z paragraphs describing local state transitions or the combination of other actions using CSP combinators; and a nameless action, normally formed by the combination of the other actions, describing the behavior of the process. Processes can also be defined by combination of other processes, using the CSP constructors.

Circus has important contributions to # programming systems, regarding specification of parallel programs. Besides to make possible to think about an integrating environment for formal reasoning about behavioral and functional aspects of #-components belonging to a subset of kinds supported by the # programming system, as pointed out in Sections 2.4 and 2.5, it makes possible the generation of source code after the application of successive refinement steps, with help of some automatic support. It is planned to use this feature for generation of source code targeting specific architectures. The # component model also has contributions to *Circus*. The authors argue that the # component model may be a useful approach for modular description of *Circus* specifications.

The remaining portion of this section presents how *Circus* can be used to specify #-components, and how the overlapping composition combinators (**join** and **fold**) may be applied to combine *Circus* specifications of #-components, forming new ones. Moreover, syntactical extensions to *Circus* are proposed to support overlapping composition operations. The case study of Section 4 will attempt to present the ideas and syntactical extensions in a more intuitive way. The convention adopted is to use *slanted* font to refer to Circus concepts and *italic* to refer to concepts in the # component model.

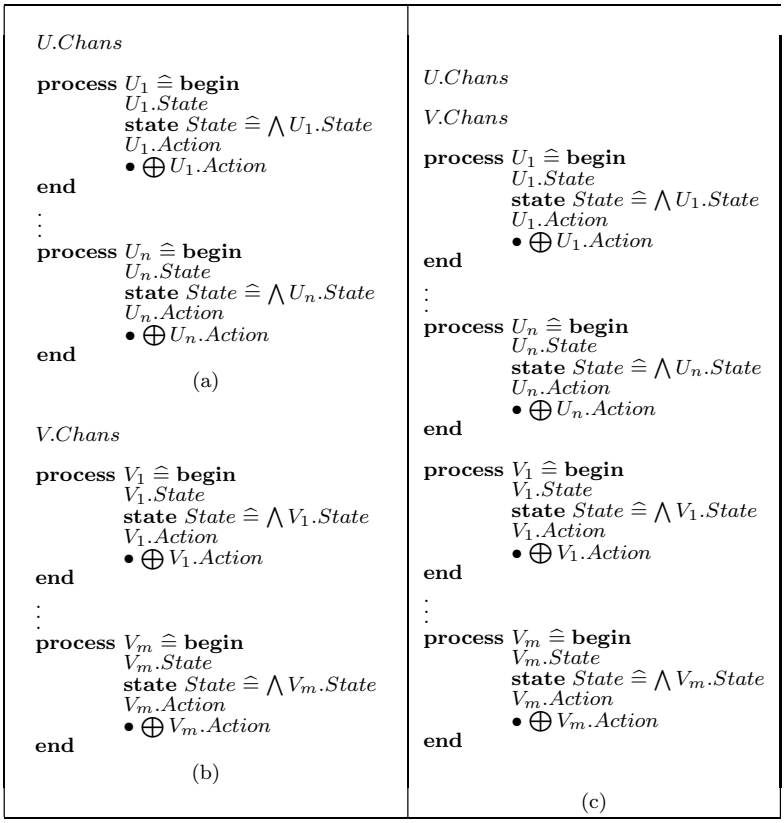

Fig. 9. Joining Schema

3.1 The Definition of the Category U

To present the formal definition of the representation of #-components as *Circus* specifications, it is only necessary to define the required category U, of units, as mentioned in Section 2.3. The definition of a #-component and the overlapping composition operators is already defined by HOCC in Section 2.3. A basic background in Category Theory [32,20] is provided in Appendix A, but readers that are not interested in deep formal details may ignore this section without compromising their understanding. The remaining sections present the ideas formalized in this section in a more intuitive sense.

Definition 3.1 Let Σ be a set of symbols denoting *action labels*. In fact, action label correspond to *events* of CSP processes. The category $\mathsf{Actions}_\Sigma$ is defined by the following components:

Objects ($\mathbf{obj}_{\mathsf{Actions}_\Sigma}$): Actions inductively built from the set of action labels Σ by application of CSP operators. Thus, an *action* is an action label or it is obtained from the application of a CSP operator over a set of other *actions*. The action labels in an action a is denoted by $\sigma(a)$.

Morphisms ($\mathbf{mor}_{\mathsf{Actions}_\Sigma}$): Let a_1 and a_2 be actions. A morphism $a_1 \rightarrow a_2$ exists if, and only if, $a_2 \sqsubseteq a_1$, where $\sqsubseteq$ is a refinement relation in CSP [34].

Program	::= HHeader HJoin* CircusPar*	1 – *#-component declaration*
·HHeader	::= Kind HashId **where**	2 – *kind of the #-component*
·Kind	::= **computation** | **synchronizer** | **data**	
·HJoin	::= **inner component** $\overline{N}$ HRangeSet? : HashId	3 – *declaring an inner component*
· · HRangeSet	::= [HRange+]	4 – *indexed notation*
· · HRange	::= HExpr . . . HExpr	
·HashId	::= $\overline{N}$ HParams? HPublic?	5 – *Reference of a #-component*
· · HParams	::= ⟨HExpr+⟩	6 – *parameters of the #-component*
· · HPublic	::= $\left(\overline{N^+}\right)$	7 – *public inner components*
CircusPar	::= . . . | **process** $N \triangleq$ HSlice* **Proc**	8 – *declaring a unit (process)*
·HSlice	::= **slice** $\underline{N}$ **from** $\overline{N}$ HIndex?.$\underline{N}$	9 – *declaring a slice of the unit*
· · HIndex	::= [N+]	10 – *indexed notation*
Proc	::= . . . | $\underline{N}$	11 – *reference to a slice (process)*
Action	::= . . . | $\underline{N}$!	12 – *reference to the slice action*
Schema-Ref	::= . . . | $\underline{N}$::	13 – *reference to the slice state*

Fig. 10. *Circus* Syntax with # Extensions

Composition ($\circ_{\mathsf{Actions}_\Sigma}$): It can be derived from the fact that refinement relations are *transitive* and *associative*.

Identities ($\mathbf{id}_{\mathsf{Actions}_\Sigma}$): It can be derived from the fact that refinement relation is *reflexive*. Thus, if $a \in \mathbf{obj}_{\mathsf{Actions}_\Sigma}$, $id_a : a \to a \in \mathbf{mor}_{\mathsf{Actions}_\Sigma}$

Definition 3.2 The category U is defined by a *full sub-category* of the category $\mathsf{Set} \times \mathsf{Set} \times \mathsf{Actions}_\Sigma$, including those objects of the form $\langle State, Action, a\rangle$ that are valid *Circus* processes satisfying the following restrictions:

- *State* is a set of paragraphs defining its internal state.
- *Action* is a set of paragraphs defining labeled actions that specify state modifications that may occur in the process.
- The action combinator a defines the behavior of the process by an interleaved execution of actions in *Actions*. For that, let L be the set of labels of actions in *Action*, such that $L \subset \Sigma$. It is required that $\sigma(a) = L$, which means that the action a makes reference only to the actions defined in the process.

The category U, as defined above, defines a weak refinement relation between *Circus* processes, where for a process u_1 to be a refinement of a process u_2 it is only necessary that u_1 includes all state and action paragraphs of u_2 and that the action combinator a_1 be a refinement of the action combinator a_2. Stronger refinement relations between *Circus* processes exist [35], but the definition provided here is sufficient for the purpose of combining *Circus* specifications using overlapping composition.

Recall that Section 2.3 has defined the meaning of folding operators $\oplus$ as colimits in the category U. In terms of the definition of U proposed in this section, the unit u, in $u = u_1 \oplus u_2$, includes only the state and action paragraphs of u_1 and u_2. Moreover, the action of u is a refinement of the actions of both u_1 and u_2 and any other unit u' that is a refinement of both u_1 and u_2 is a refinement of u.

3.2 Circus Specifications of #-Components

According to the formal definition introduced in the previous section, in a *Circus* specification of a #-component, *processes* denote *units* and *channels* are taken as special #-components, denoting primitive *primitive synchronizers*. Since *channels* in *Circus* have *mailbox* semantics, a #-component *ch*, denoting a channel, has an enumerated *unit* receive, with one unit denoting the references to '*ch*?' for each receiver *process*, and an enumerated *unit* send, with one unit denoting the references to '*ch*!' for each transmitter *process*.

Figure 9(a) presents a schema of a *Circus* specification of a #-component, used throughout this paper, comprising a set of channels, denoted by $U.Chans$ and a set of processes, denoting units. Each process U_i $^{i\in\{1...n\}}$ has a set of paragraphs, denoted by $U_i.State$, whose conjunction denotes the local state of the process, and a set of actions, denoted by $U_i.Action$, which forms the main action of the process when applied to an action combinator $\bigoplus$. In fact, in a unit of a composite #-component C, the set of state and action paragraphs are inherited from the slices of the unit, which, by overlapping composition, are units of inner components of C.

3.2.1 Enumerated Units

In practical # programming systems, it is convenient to define a notation for specifying an enumeration of N units, for an arbitrary N. Such notation is not included in the syntax of the overlapping composition calculus, since it only intends to formalize the semantics of composition, but it is supported in HPE using *indexed notation* [4]. For the purposes of this paper, it is relevant to make a explicit discussion about specification of enumerated units because they represent an important class of parallel program in SPMD style (Single Program Multiple Data). CCA, for example, define the notion of SCMD components (Single Component Multiple Data), only to represent such class of parallel programs.

Enumerated units are represented in *Circus* using *replicated processes*. For instance, an enumeration of N units U, represented by $U[i]$ $^{i=1...N}$ in HPE, is defined by the schema

$$\textbf{process}\ U \mathrel{\widehat=} |||\ i : \mathbb{I} \bullet \langle \textit{definition of the } i^{th} \textit{ process} \rangle$$

where $\mathbb{I} = \{1, \ldots, N\}$.

The use of $|||$ is semantically correct, since it makes sense to think about units of #-components as independent units of execution. In *Circus*, the set $\mathbb{I}$ is called *index set* and i is called the *index variable*.

3.3 Overlapping Composition of Circus Specifications of #-Components

Now that the structure of a *Circus* specification of a #-component was discussed, it is necessary to discuss how a *Circus* specification of a #-component can be derived

[4] In fact, the semantics of *indexed notation* in # programming systems may be defined in terms of recursive configurations, but, since *replicated processes* are very close to *indexed notation*, we decided to avoid to include recursion terms in the # overlapping composition calculus to make our formalization simpler.

from the overlapping composition combinators *join* and *fold* applied over *Circus* specifications of other #-components. In fact, it is worth to note that the following two sections presents intuitively which is already implicit in the semantics of terms of HOCC and the definition of the category U presented in Section 3.1.

3.3.1 Joining

To *join* *Circus* specifications denoting #-components means to build a new *Circus* specification, denoting a new #-component, that includes all *processes* and *channels* of the joined specifications. Figure 9(a/b) presents general schemas of specifications of #-components which are joined to form the new schema of Figure 9(c).

3.3.2 Folding

Given a *Circus* specification of a #-component, probably obtained by joining two or more other specifications, two of its *processes*, denoting some of its *units*, can be *folded* in a new *process*, denoting a new *unit*, by applying a *folding operator* $\oplus$.

Figure 2 illustrates two folding scenarios, where a pair of *units* are folded without sharing (b) and with sharing (c). Figures 11(a) and 11(b) present schemas for *processes* denoting *units* U and S, which are *slices* of *units* Q and R, respectively. Figures 11(c) and 11(d) show schemas of *processes* denoting *units* Q and R, pointing out that they include local states and *actions* of U and S, respectively. Finally, Figures 11(e) and 11(f) exhibit the *process* schema denoting unit P, obtaining by folding *units* Q and R. In the first case, Q and R do not share a *slice*. *States* and *actions* of the folded *processes* are only joined. In the second case, Q and R share a *slice* U. The *states* and *actions* corresponding to U are shared. One should notice that the public component that owns U is a refinement of the public component that owns S. At present, for the sake of simplicity, it has been adopted a *weak* (syntactical) form of refinement relation, defined in such a way that $U.State \supseteq S.State$, $U.Action \supseteq S.Action$, making safe to supersede S with U in R.

Enumerated units can be folded if their *index sets* and *index variables* are the same. If the *index variables* are different, they can be renamed. For instance, let

$$\textbf{process}\ U_1 \mathrel{\widehat{=}} |||\ i : \mathbb{I} \bullet\ \mathsf{Proc}_1 \text{ and } \textbf{process}\ U_2 \mathrel{\widehat{=}} |||\ j : \mathbb{I} \bullet\ \mathsf{Proc}_2$$

be enumerated units. A schema for folding of U_1 and U_2 is like

$$\textbf{process}\ V \mathrel{\widehat{=}} |||\ k : \mathbb{I} \bullet\ \text{folding of } \mathsf{Proc}_1\,[i/k] \text{ and } \mathsf{Proc}_2\,[j/k]$$

, where *processes* $\mathsf{Proc}_1\,[i/k]$ and $\mathsf{Proc}_2\,[j/k]$ are folded like in Figure 11. The notation $\mathsf{Proc}\,[i/k]$ means that occurrences of i are replaced by k in process Proc.

For the sake of simplicity, whenever the two folded units have no public slices (their inner components have no public inner components), it is possible to combine them by using a *process* combinator, since no shared state may occur.

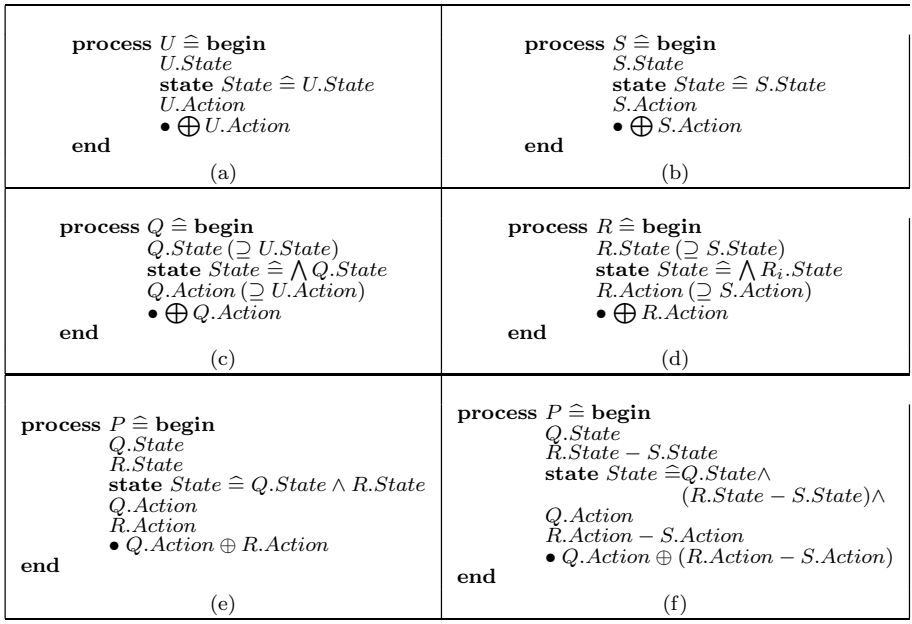

Fig. 11. Folding Schema

3.4 Splitting Units

In # programming, it is often necessary to split an *enumerated unit* in two or more *enumerated units* that may be folded to *enumerated units* of distinct #-components. For that, by taking an enumerated unit in the form

$$\textbf{process}\ U \mathrel{\widehat{=}} |||\ i : \mathbb{I} \bullet\ \mathsf{Proc},$$

one may derive two enumerated units

$$\textbf{process}\ U \mathrel{\widehat{=}} |||\ i : \mathbb{I}_1 \bullet\ \mathsf{Proc}\ \textit{and}\ \textbf{process}\ U \mathrel{\widehat{=}} |||\ i : \mathbb{I}_2 \bullet\ \mathsf{Proc},$$

provided that $\mathbb{I} = \mathbb{I}_1 \cup \mathbb{I}_2$.

4 Case Study

Figures 12 and 13 present simplified specifications for the #-components of the case study introduced in Section 2.6. Readers familiar with the syntax of *Circus* may notice the syntactic extensions for overlapping composition, which are presented in Figure 10.

The first two specifications, in Figures 12(a) and 12(b), are VECTOR and MATRIX, for #-components denoting sequential vectors and matrices, respectively, that reside in the same address space. They are *primitive* #-components, since they do not declare *inner components*. Their respective specification headers declare that they are of kind **data** and their names.

The #-components of kind **data**, representing data structures, are stateful. For the purposes of this paper, other possible kinds are **computations**, denoting #-

components that implement useful parallel computations over parallel data structures, and **synchronizers**, denoting #-components that implement useful interaction patterns among processes. Only in specifications of #-components of kind **synchronizers**, *channels* may be explicitly declared.

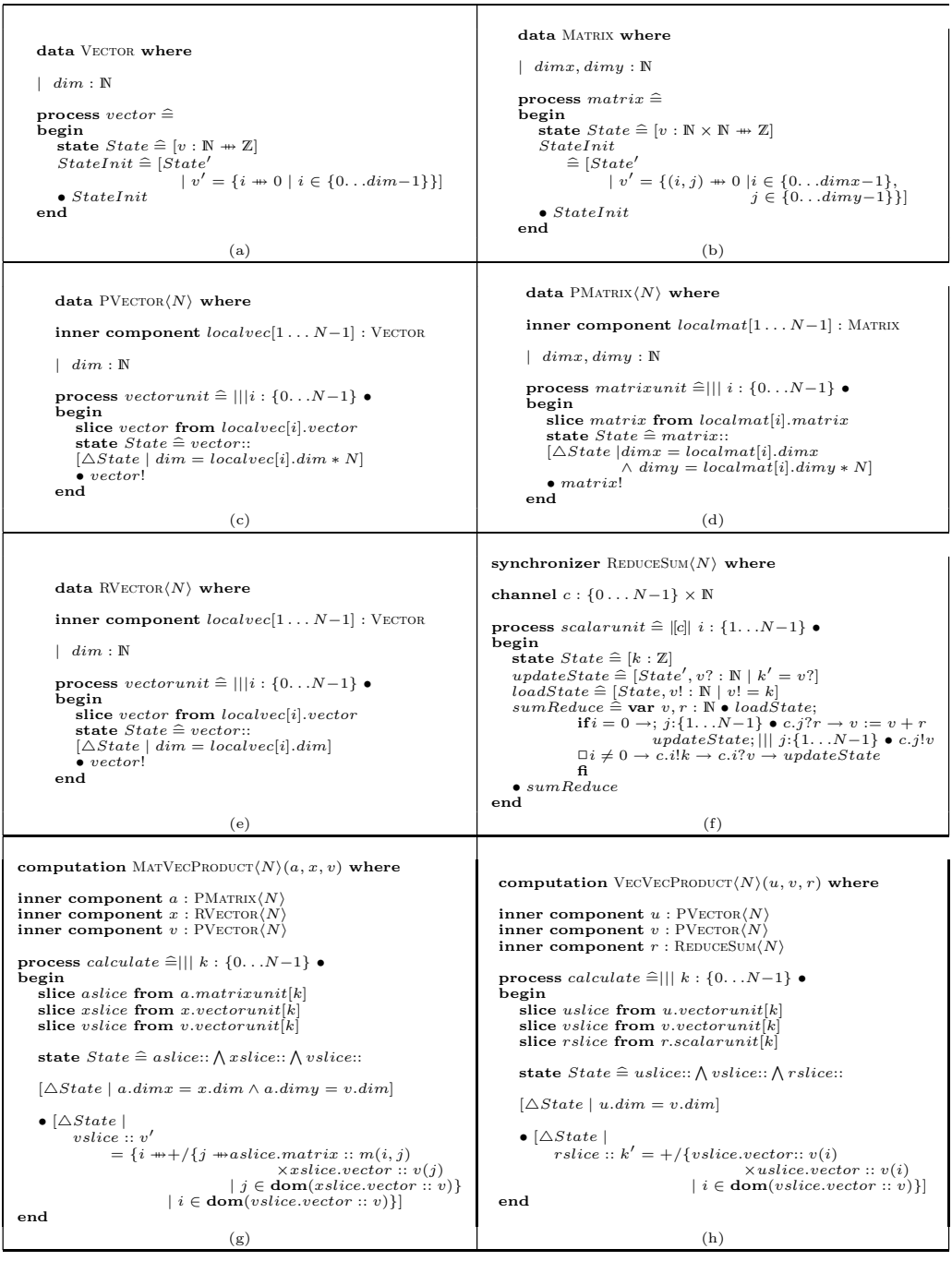

Fig. 12. Specifications for the Example

```
synchronizer ScatterMxN⟨M, N⟩(in, out) where

inner component in : PVector⟨M⟩
inner component out : PVector⟨N⟩

channel c : {0...N−1} × ℕ × ℕ

| dim : ℕ

process a ≙ ‖[c]|i : {0...M−1} •
begin
   slice islice from in.vectorunit[i]
   slice oslice from out.vectorunit[i]

   state State = islice:: ⋀ oslice::

   [△State | dim = in.dim = out.dim]

   updateL ≙ [△State; kj?, v? : ℕ
              | oslice.vector :: v′
                    = oslice.vector :: v ⊕ {kj ↦ v}]

   updateR ≙ (j, ki, kj : ℕ •
              c.j!(kj, islice.vector :: v(ki)) → skip)

   distribute ≙ ||| :ki{0...(dim/M) − 1} •
                  var ii, j, kj, v : ℕ •
                  ii := ki + i ∗ (in.dim/M);
                  j := ii div (dim/N);
                  kj := ii % (dim/N);
                  if j = i →v := islice.vector :: v(kj);
                            updateL
                  □ j ≠ i → updateR (j, ki, kj)
                  fi

   collect ≙; {1...dim/N} • c.i?(kj, v) → updateL;

   • distribute ||| collect
end

process b ≙ ‖[c]|i : {M...N−1} •
begin
   slice oslice from out.vectorunit[i]

   state State = out::

   updateL ≙ [△State; kj?, v? : ℕ
              | outslice.vector :: v′
                = oslice.vector :: v ⊕ {kj ↦ v}]

   collect ≙; {1...dim/N} • c.i?(kj, v) → updateL

   • collect
end
```

(a)

```
application AppExample⟨M, P⟩(a, x, b, y, r) where

inner component axv:MatVecProduct⟨M⟩(a, x, inv)
inner component rV:ScatterMxN⟨M, M + P⟩(inv, outv)
inner component byu:MatVecProduct⟨P⟩(b, y, inu)
inner component rU:ScatterMxN⟨P, M + P⟩(inu, outu)
inner component byu:VecVecProduct⟨P⟩(outv, outu, r)

process p ≙ ‖[rV.c, rU.c]‖ i : {0...M−1} •
begin
   slice doAXV from axv.calculate
   slice redistV from rV.a
   slice redistU from rU.b
   slice doVUr from vur.calculate

   state State ≙doAXV:: ⋀ redistV::
                        ⋀ redistU:: ⋀ doVUr::

   • doAXV!;
     (redistV! ||| redistU!);
     doUVr!
end

process q ≙ ‖[rV.c, rU.c]‖ i : {M...M+P−1} •
begin
   slice doBYU from byu.calculate
   slice redistU from rU.a
   slice redistV from rV.b
   slice doVUr from vur.calculate

   state State ≙doBYU:: ⋀ redistU::
                        ⋀ redistV:: ⋀ doVUr::

   • doBYU!;
     (redistU! ||| redistV!);
     doUVr!
end
```

(b)

Fig. 13. Specifications for the Example (continued)

The parallel counterparts of Vector are RVector and PVector. The latter denotes #-components implementing vectors that are replicated in the address spaces of two or more processes, while the former denotes #-components implementing vectors that are contiguously and equally partitioned across the address spaces of two or more processes. Each partition of a PVector is a Vector, represented by an inner component *localvec*[*i*], for $i \in \{0...N-1\}$. In fact, in HPE, RVector and PVector are concrete implementations of the same *abstract component*, representing parallel vectors, but assuming different *partition strategies*. PMatrix represents a matrix equally partitioned by rows in the address space of two or more processes. Each partition is a Matrix, represented by an inner component *localmat*[*i*], for $i \in \{0...N-1\}$. The headers of PVector, RVector and PMatrix says that they are parameterized by the value N (enclosed by angle brackets), denoting the number of processes where the parallel data structures are distributed.

The set of **inner component** declarations specifies the collection of #-components to be joined, or, in other terms, the direct inner components of the #-component being specified. In fact, the effect of an **inner component** declaration is to import all global declarations of the specification of the inner component, including channels, constants, and types, to the enclosing specification. Besides that, the specification of each of its processes must be unfolded inside a process of the enclosing configuration by using the **slice** declaration. Thus, for example, PVECTOR declares N inner components, locally named $localvec[i]$, for $i \in \{0...N-1\}$, by using an indexing notation, representing the local vectors whose units *vector* are unfolded in the enumerated unit *vectorunit*, becoming its slices. In the i-th unit *vectorunit*, all identifiers in *vector* are prefixed by $localvec[i]$ (the identifier of the inner component) for avoiding naming clashes.

The synchronizer #-component REDUCESUM denotes the reduction of integer values stored by separate processes, by applying sum. The resulting sum is stored in each process.

A #-component MATVECPRODUCT, with an enumerated unit *calculate*, represents the product of a PMATRIX-matrix a and a RVECTOR-vector x, resulting in a PVECTOR-vector v. These data structures are represented by inner components. One may notice that the distributed vectors x and v have distinct partition strategies, which are the appropriate ones for promoting better data locality, avoiding communication among *calculate* units. The data structures a, x, v are *public*, as defined in the header of MATVECPRODUCT (enclosed by parenthesis) for the purpose of sharing. VECVECPRODUCT is analogous to MATVECPRODUCT. The input vectors u and v are both PVECTORS. The scalar calculated by each process, from their partitions of u and v are processed by REDUCESUM. Thus, the result r is copied into each process.

The #-component SCATTERMXN maps an input PVECTOR-vector *in* that resides in the address space of a set of processes P to an output PVECTOR-vector *out* that resides in the address space of a set of processes Q, where $Q \subseteq P$, represented by public inner components. It may be viewed as a generalized *scatter* operation, in the sense of message passing libraries such as MPI [29], where there are many source processes, where the input data is partitioned, instead of only one. For that, it has two enumerated units, named a and b. The first one ranges from 0 to $M-1$, representing processes in P, while the last one ranges from M to $N-1$, representing processes in Q. For this reason, by unfolding (**slice** declaration), the units of the PVECTOR-vector *in* becomes slices of a, while units of the PVECTOR-vector *out* are split in two groups, respectively ranging from 0 to $M-1$ and M to $N-1$ and mapped as slices to units a and b.

Finally, the #-component APPEXAMPLE implements the configuration of Figure 7, comprising two enumerated units named p and q. As described in Section 2.6, p represents the processes involved in the calculation of $\widehat{v} = A \times \widehat{x}$, while q represents the processes involved in the calculation of $\widehat{u} = B \times \widehat{y}$. All units are involved in the calculation of $r = \widehat{v} \cdot \widehat{u}$. The inner components axv, byu, rV, and rU represent the required operations. In their declarations, the public inner components are renamed

and, by naming match, some of them are configured to be shared. For example, the input of rV is the output of axv, named in_v, meaning that the result of the $A \times \widehat{x}$, a vector placed in M processes is distributed across the N processes involved in the overall calculation. The inputs of byu are the outputs of rV and rU, named out_v and out_u, respectively, since $r = \widehat{v} \cdot \widehat{u}$ is executed in all processes.

The main actions of p and q define that matrix-vector multiplications, redistributions of vectors $\widehat{v}$ and $\widehat{u}$, and vector-vector product are executed sequentially. Moreover, redistributions of $\widehat{v}$ and $\widehat{u}$ are performed in parallel, since there is no data dependency between rV and rU.

5 Final Remarks and Further Works

Previous work mapped the original semantics of parallel programs in Haskell$_\#$ [14] onto OCCAM and CSP, targeting at the analysis of formal properties of parallel programs from their behavior specification [24]. At that time, our research group was working with the translation between Petri nets and some well-known concurrent languages for programming and specification, such as LOTOS and OCCAM [27,26], motivating the definition of translation schemas of Haskell$_\#$ onto Petri nets [25,17]. Besides that, schemas for partial generation of code capturing Petri nets behavior, using *behavior protocols* [14] were proposed. They were defined as synchronized regular expressions, a simple formalism that has equivalence with Petri nets according to tracing semantics [23]. Behavior protocols were adopted in HPE, but they are not able to specify functional meaning of the units of a #-component.

Thus, *Circus* comes to fill a gap in our previous work on the specification of parallel programs targeting at # programming systems. Besides to include the subset of CSP that is necessary to the model behavior of #-components, expressed by the order of activation of units that denote actions, it adds expressiveness for specifying the functional meaning of units of #-components of some subset of kinds of # programming systems that describe computations in some host programming language, making possible to generate code by applying semi-automatic refinement steps. The incorporation of *Circus* specifications into the Front-End of HPE, for the specification of units in configurations, replacing behavior protocols is also in scope. It is also planned to apply the earlier works on the translation between CSP and Petri nets to translate the CSP subset of *Circus* specifications of #-components onto Petri nets.

Besides the contributions of *Circus* to # programming systems, the approach presented in Section 3 also enriches *Circus*, by introducing a compositional approach, based on the # overlapping composition, to build *Circus* specifications.

Further papers will attempt to show better examples of specifications of #-components using *Circus* and how they can be used to improve the practice of parallel programming with #-components, by proving properties about #-components, their composition, and their coordination. In fact, there is still a lot of work to do to define how to integrate *Circus* tools to a # programming system like HPE. For instance, a important question that arises is how specifications could be used to

certify deployed #-components. This is an important concern in scientific and engineering application domain, since it is very important for scientists and engineers to know if #-components implement correctly some mathematical model of their interest or synchronization patterns that are used to implement them appropriately on the target execution platform. The today's practice of scientists and engineers is to certify reusable pieces of software by reputation, where a "component" is considered "certified" after many years of successful use in practice.

References

[1] *Wiki of MCMD-Workgroup at CCA Forum*, https://www.cca-forum.org/wiki/tiki-index.php?page=MCMD-WG.

[2] *The semantics of circus*, in: D. Bert, J. P. Bowen, M. C. Henson and K. Robinson, editors, *ZB 2002: Formal Specification and Development in Z and B*, Lecture Notes in Computer Science **2272** (2002), pp. 184—203.

[3] *Circus Website*, http://www.cs.york.ac.uk/circus/ (2008).

[4] Arbab, F., *Reo: A Channel-Based Coordination Model for Component Composition*, Mathematical Structures in Computer Science **14** (2004), pp. 329–366.

[5] Armstrong, R., G. Kumfert, L. C. McInnes, S. Parker, B. Allan, M. Sottile, T. Epperly and Dahlgreen Tamara, *The CCA Component Model For High-Performance Scientific Computing*, Concurrency and Computation: Practice and Experience **18** (2002), pp. 215–229.

[6] Baduel, L., F. Baude and D. Caromel, *Asynchronous Typed Object Groups for Grid Programming*, Journal of Parallel Programming **35** (2007), pp. 573–613.

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

[8] Baude, F., D. Caromel, L. Henrio and M. Morel, *Collective Interfaces for Distributed Components*, in: *7th International Symposium on Cluster Computing and the Grid (CCGrid 07)* (2007).

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

[10] 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).

[11] Carvalho Junior, F. H., R. M. F. Lima and R. D. Lins, *Coordinating Functional Processes with Haskell$_{\#}$*, in: ACM Press, editor, *ACM Symposium on Applied Computing, Track on Coordination Languages, Models and Applications*, 2002, pp. 393–400.

[12] Carvalho Junior, F. H. and R. Lins, *A Categorical Characterization for the Compositional Features of the # Component Model*, ACM Software Engineering Notes **31** (2006).

[13] Carvalho Junior, F. H., R. Lins, R. C. Correa and G. A. Araújo, *Towards an Architecture for Component-Oriented Parallel Programming*, Concurrency and Computation: Practice and Experience **19** (2007), pp. 697–719, special Issue: Component and Framework Technology in High-Performance and Scientific Computing. Edited by David E. Bernholdt.

[14] 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.

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

[16] Carvalho Junior, F. H. and R. D. Lins, *An Institutional Theory for #-Components*, Electronic Notes in Theoretical Computer Science **195** (2008), pp. 113–132.

[17] Carvalho Junior, F. H., R. D. Lins and R. M. F. Lima, *Translating Haskell$_{\#}$ Programs into Petri Nets*, Lecture Notes in Computer Science (VECPAR'2002) **2565** (2002), pp. 635–649.

[18] Dijkstra, E. W., *Guarded commands, nondeterminacy and the formal derivation of programs*, Communication of the ACM **18** (1975), pp. 453–457.

[19] Eilenberg, S. and S. Mac Lane, *General Theory of Natural Equivalences*, Annals of Mathematics **43** (1942), pp. 757–831.

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

[21] Goguen, J., *Categorical Foundations for General Systems Theory*, Advances in Cybernetics and Systems Research (1973), pp. 121–130.

[22] Hoare, C. A. R., "Communicating Sequential Processes," Prentice-Hall International Series in Computer Science, 1985.

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

[24] Lima, R. M. F. and R. D. Lins, *Haskell$_{\#}$: A Functional Language with Explicit Parallelism*, Lecture Notes in Computing Science (VECPAR - International Meeting on Vector and Parallel Processing) (1998), pp. 1–11.

[25] Lima, R. M. F. and R. D. Lins, *Translating HCL Programs into Petri Nets*, in: *Proceedings of the 14th Brazilian Symposium on Software Engineering*, 2000.

[26] Lima, R. M. F., R. D. Lins and J. A. M. Queiroz, *Translating Basic LOTOS into Occam 2*, in: *EUROMICRO'96 (Beyond 2000: Hardware/Software Design Strategies)*, 1996, pp. 229–234.

[27] Maciel, P. R. M., R. D. Lins and P. R. F. Cunha, *Introdução às Redes de Petri e Aplicações*, Décima Escola de Computação (1996).

[28] 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.

[29] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, International Journal of Supercomputer Applications and High Performance Computing **8** (1994), pp. 169–416.

[30] 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.

[31] Morgan, C. C., "Programming from Specifications," Prentice-Hall, 1994.

[32] Pierce, B., "Basic Category Theory for Computer Scientists," The MIT Press, 1991.

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

[34] Roscoe, R. W., "The Theory and Practice of Concurrency," Prentice-Hall, 2005.

[35] Sampaio, J. C. P., A. C. A. Woodcock and A. L. C. Cavalcanti, *Refinement in Circus*, Lecture Notes in Computer Science **2391** (2002), pp. 451–470, Formal Methods Europe (FME) 2002.

[36] Smith, D. R., "Composition by Colimit and Formal Software Development," Springer, 2006 pp. 317–332.

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

[38] Woodcock, J. C. P. and J. Davies, "Using Z Specification, Refinement, and Proof," Prentice-Hall, 1996.

A Category Theory Background

Category theory [32,7,20] is a relatively young branch of mathematics, firstly introduced by Mac Lane and Eilenberg when interested to develop the notions of of *functor* and *natural transformation* [19]. It is a basic conceptual and notational framework in the same sense of set theory and graph theory, for example, but placed at a higher level of abstraction. In fact, the notion of cateogory is often introduced as a generalization of sets or graphs, but reaching sufficient abstraction power to

express more complex algebraic mathematical structures in a general setting that allows to study the commonalities of concepts in and between these structures. Due to its ability to deal with abstraction, category theory has been adopted as a standard mathematical framework in several computer science domains, replacing the role of set theory. In fact, it has been extensively used in developments in theories of programming languages (types, semantics, and implementation), concurrency, automata, formal methods, constructive logic, algorithms, and so on.

This appendix provides a brief introduction to basic categoric constructions that are used in this paper. The readers that are interested in additional background in category theory may use the proposed literature.

A.1 The Definition of Category and Their Basic Constructions

A category $\mathcal{C}$ is defined by a tuple $\langle \mathbf{obj}_{\mathcal{C}}, \mathbf{mor}_{\mathcal{C}}, \circ_{\mathcal{C}}, \mathbf{id}_{\mathcal{C}} \rangle$, whose elements are defined as following:

- $\mathbf{obj}_{\mathcal{C}}$ is a collection of *objects*, known as $\mathcal{C}$-objects;
- $\mathbf{mor}_{\mathcal{C}}$ is a collections of *arrows*, also known as *morphisms*. A $\mathcal{C}$-arrow f is written $f : A \to B$, where $A, B \in \mathbf{obj}_{\mathcal{C}}$. A and B are, respectively, the domain and codomain of f;
- $\circ_{\mathcal{C}}$ is an *associative composition operation*. Let $A, B, C \in \mathbf{obj}_{\mathcal{C}}$ and $f : A \to B, g : B \to C \in \mathbf{mor}_{\mathcal{C}}$. Then, $f \circ_{\mathcal{C}} g : A \to C \in \mathbf{mor}_{\mathcal{C}}$. Since $\circ_{\mathcal{C}}$ is associative, $(f \circ_{\mathcal{C}} g) \circ_{\mathcal{C}} h = f \circ_{\mathcal{C}} (g \circ_{\mathcal{C}} h)$ for any arrows $f, g, h \in \mathbf{mor}_{\mathcal{C}}$ that may be composed. The operator $\circ_{\mathcal{C}}$ will be written $\circ$ where it is not ambiguous;
- $\mathbf{id}_{\mathcal{C}}$ is a collection of identity arrows ($\mathbf{id}_{\mathcal{C}} \in \mathbf{mor}_{\mathcal{C}}$), one for each object in $\mathbf{obj}_{\mathcal{C}}$. Thus, $\iota_A : A \to A \in \mathbf{id}_{\mathcal{C}}$ if, and only if, $A \in \mathbf{obj}_{\mathcal{C}}$ and for any other morphism $h : A \to A \in \mathbf{mor}_{\mathcal{C}}$, $\iota_A \circ_{\mathcal{C}} h = h \circ_{\mathcal{C}} \iota_A = h$.

By appropriately defining objects and arrows in such a way that composition and identities hold, one may define categories for representing known algebraic structures. For example, the category **Set** defines objects as sets and arrows as the total functions between them, the category **Gr** defines objects as graphs and arrows as the graph homomorphisms.

If $\mathbf{obj}_{\mathcal{C}}$ and $\mathbf{mor}_{\mathcal{C}}$ are sets, $\mathcal{C}$ is a *small category*. The dual category of $\mathcal{C}$, called $\mathcal{C}^{op}$, is obtained by inverting direction of all arrows of $\mathcal{C}$. Duality is an important concept in category theory Some property holds for $\mathcal{C}^{op}$ if, and only if, the property holds for $\mathcal{C}^{op}$ from a dual perspective. Duality also implies that category theory concepts are always presented in pairs: the concept and its dual concept. For example, in **Set**, cartesian product is the dual concept of disjoint union, which means that disjoint union in **Set** correspond to cartesian product in $\mathbf{Set}^{op}$ and vice-versa.

Cartesian product and disjoint union are respectively the dual concepts of product and coproduct in category theory. There is a number of other important constructions involving objects and arrows in a category, whose interpretation may be taken in particular for each category, allowing to compare concepts from distinct

algebraic structures at a higher abstraction level. A morphism $f : Y \to Z$ is a ***monomorphism*** if for any two morphisms $g, h : X \to Y$, $f \circ g = f \circ h$ implies that $g = h$. Its dual concept is ***epimorphism***. f is an ***isomorphism*** if its inverse morphism $f^{-1} : Z \to Y$ exists, such that $f \circ f^{-1} = id_Z$ and $f^{-1} \circ f = id_Y$. It may be proved that every ***isomorphism*** is both a ***monomorphism*** and an ***epimorphism***. A $\mathcal{C}$-object I is an ***initial object*** of $\mathcal{C}$ if for any $\mathcal{C}$-object J there exist an unique $\mathcal{C}$-morphism $i : I \to J$. Its dual concept is called ***terminal object***. Let $f, g : X \to Y$ be $\mathcal{C}$-morphisms.

Let **Cat** be the category where objects are small categories. **Cat**-morphisms are reflexive graph homomorphisms, called ***functors***. Let $F : \mathcal{C}_1 \to \mathcal{C}_2$ be a functor. It is defined by a pair of functions $\langle F_O, F_M \rangle$, where $F_O : \mathbf{obj}_{\mathcal{C}_1} \to \mathbf{obj}_{\mathcal{C}_2}$ and $F_M : \mathbf{mor}_{\mathcal{C}_1} \to \mathbf{mor}_{\mathcal{C}_2}$. Let $F, G : \mathcal{C} \to \mathcal{C}'$ be functors. A ***natural transformation*** η from F to G ($\eta : F \to G$) associates to every $\mathcal{C}$-object X a $\mathcal{C}'$-morphism $\eta_X : F(X) \to G(X)$, such that $\eta_Y \circ F(f) = G(f) \circ \eta_X$ is satisfied for every $\mathcal{C}$-morphism $f : X \to Y$. A useful category is $\mathcal{D}^{\mathcal{C}}$, whose objects are functors from $\mathcal{C}$ to $\mathcal{D}$ and morphisms are natural transformations between them.

A ***diagram*** $\mathcal{D}$ in a category $\mathcal{C}$ is a graph $(O, M, \partial_0, \partial_1)$, for which it is defined a graph homomorphism from $\mathcal{D}$ to $\mathbf{shape}(\mathcal{C})$, where $\mathbf{shape} : \mathbf{Cat} \to \mathbf{Gr}$ is a forgetful functor that takes a category and maps it to its transitive and reflexive graph shape. A diagram $\mathcal{D}$ is ***commutative*** if all paths in $\mathcal{D}$ with the same origin and target are equivalent. Commutative diagrams are often used to describe properties about categories and to define categorical constructions, as in Figure 5.

Let $\mathcal{D}$ be a commutative diagram in $\mathcal{C}$. A ***cone*** over $\mathcal{D}$ is defined by a pair $\langle \mathcal{A}, X \rangle$, where $\mathcal{A}$ is a collection of $\mathcal{C}$-morphisms and X is a $\mathcal{C}$-object, such that: for all nodes Y_i in $\mathcal{D}$, there is a corresponding morphism $a_i : Y_i \to X \in \mathcal{A}$ and for all arcs $f : Y_n \to Y_m$ in $\mathcal{D}$, $a_n \circ f = a_m$. The concept of ***cocone*** is dual to the concept of cone. A cone $\langle \mathcal{A}, X \rangle$ is a ***limit*** of $\mathcal{D}$ if for any cone $\langle \mathcal{A}', X' \rangle$ in $\mathcal{D}$ there exists a $\mathcal{C}$-morphism $g : X \to X'$, such that for any $a_i : Y_i \to X \in \mathcal{A}$ and $a'_i : Y_i \to X' \in \mathcal{A}'$, $g \circ a_i = a'_i$. The dual concept of limit is ***colimit***. A limit (colimit) is a ***product*** (***coproduct***) whenever $\mathcal{D}$ is a finite and discrete diagram (*FD-diagram*).

In Section 2.3, the concept of cocone have been used to define the semantics of *folding operators* in overlapping composition. The intuition behind the use of this concept in such context was discussed. In fact, research in software architecture have been revealed the important role of cocones and colimits for defining semantics and understand the fundamentals of composition of systems [21,20,36].