The # Model: Separation of Concerns for Reconciling Modularity, Abstraction and Efficiency in Distributed Parallel Programming

Francisco Heron de Carvalho Junior *
Instituto Titan
Av. Dom Luiz, 880, sala 406
Fortaleza, Brazil
heron@institutotitan.org.br

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

ABSTRACT
The computer science community has been looking for parallel languages and models with a higher level of abstraction and modularity, without performance penalties, that could be used in conjunction with advanced software engineering techniques, and that are suitable to work with large-scale programs. This paper discusses how the # parallel programming model addresses the issues of modularity and abstraction of parallel programs using the techniques of separation of concerns.

Categories and Subject Descriptors
D.1.1 [Programming Techniques]: Concurrent Programming—Distributed programming, Parallel Programming; D.3.2 [Programming Languages]: Languages Classifications—Concurrent, distributed and parallel languages

General Terms
Languages, Software Engineering

Keywords
Parallel Programming, Separation of Concerns, High Performance Computing Software

1. INTRODUCTION
Several research initiatives developed through the last two decades made possible the use of distributed architectures for high performance computing (HPC), with larger scalability and lower prices than vector systems, causing the consolidation of massively parallel processors (MPP’s), clusters [6], and grids [15] as the main HPC architectures of today [21]. Distributed parallel architectures have brought new challenges for HPC users, regarding their programmability. In the old vector systems, transparent loop parallelization techniques allowed data parallel applications to achieve peak performance with minor additional programming effort. But in distributed programming, configuration of parallelism should be explicit in order to reach peak performance.

In general, the evolution of a programming technology may be divided into three phases. In the first phase, efficiency is the main concern in programming, due to hardware constraints. The second phase looks for portability, due to dissemination of a variety of computer architectures, each one with its own low level programming features. Portable languages and programming models emerge from this phase. The third phase searches for higher levels of abstraction that allow large scale applications for which low abstract and unstructured programming methods are inappropriate.

The first phase of evolution of sequential programming techniques goes from the first years of computer technology to the years before the advent of FORTRAN, in 1954, which marks the beginning of the second phase. The third phase begins with software crisis from the end of 1960’s [12].

There is a gap of about forty years between the evolution of parallel and sequential programming technologies. The first phase of distributed parallel programming evolution goes from the beginning of 1980’s to the creation of CRPC (Center for Research on Parallel Computation), in 1989. It was marked by the use of low level architecture-dependent message passing interfaces for parallel programming. In the second phase, research efforts have been coordinated in order to standardize parallel programming languages and libraries, yielding the development of PVM (Parallel Virtual Machine) [16], MPI (Message Passing Interface) [20], HPF (High Performance Fortran) [14], PETSc [2], ScaLAPACK, etc. The expansion in scale of applications of parallel computing, due to cluster and grid computing, has made the HPC community to search for parallel programming models...
and languages that reconcile generality (G), high level of abstraction (A), portability (P) and efficiency (E), allowing to apply advanced software engineering concepts into the program development process [26]. The GAPE requirements have not been reconciled by current programming techniques despite the efforts that began fifteen years ago with CRPC [13, 4]. Thus, we believe that evolution of parallel programming is entering its third phase.

This paper addresses the issues of modularity and abstraction in a new parallel programming model recently proposed by its authors for fitting GAPE requirements: the # parallel programming model. The third phase of evolution of conventional programming, since 1970’s, has shown the maturity of separation of concerns techniques and notions is a natural way to achieve high abstraction and modularization for programming large scale applications [24, 19]. The # model views coordination medium of parallel programs from two dimensions: the dimensions of processes and dimension of components. Components in the # sense are entities that addresses functional or non-functional, possibly cross-cutting, concerns. Current parallel programming techniques approaches process-based programming, with specification of components being a side-effect of this task. The # model, on the other hand, focuses on component-based programming, with specification of processes being a side-effect of this task. Efficiency is ensured due to the direct correspondence of programming constructors of # model have with low-level MPI primitives. Formal “debugging” is now a reality, since the coordination medium may be modeled using Petri nets, allowing the analysis of formal properties, performance evaluation, simulation, and animation of parallel programs using several existing automatic tools [5].

This paper is organized as follows. Section 2 introduces the # models for parallel programming, discussing how # programs are specified at coordination and computation levels. Section 3 presents an application used in Section 4 for demonstrating separation of concerns capabilities of the # programming environment. Finally, Section 5 concludes this paper and draws lines for further works.

2. THE # PROGRAMMING MODEL

The # parallel programming model and language may be viewed as a structured way to reason about message passing programming. Its relation with message passing conventional approaches may be compared to the relation of structured and unstructured sequential programming approaches. First, it divides the concerns of specification of computations and parallelism configuration in two orthogonal dimensions: computation and coordination, using distinct languages for these purposes whose constructors does not intersect.

In opposition to conventional message-passing techniques for parallel programming, the focus of # programming is on composing components, instead of composing processes. Components are entities, atoms of modularization, that implement functional or non-functional concerns, possibly cross-cutting ones. The composition of components yields the specification of a process topology as a side-effect. In conventional programming, the composition of processes implicitly specifies the composition of units of concern, presented in a non-modular way. The focus on component programming makes possible to apply advanced engineering techniques in the development of # implementations of large-scale applications.

Components may either be composed or simple. Composed components are programmed using the # configuration language (HCL), from the composition of other components. HCL differs from other compositional languages because it supports overlapping composition and interlacing concerns addressed by components. Simple components, or functional modules, are implemented using a host language, supposed to be any general purpose sequential language bound to the # environment. Functional modules may implement only functional concerns, the atoms of functionality of # programs. The set of composed components forms the coordination medium of a # program, while its set of simple components forms its computation medium.

The # model of parallel programming supports the notion of skeletons [11]. Partial topological skeletons (PTS) expose topological patterns of interaction amongst processes. Skeletons may be used to produce efficient code targeted at specific architectures and execution environments [9]. PTS’s are implemented as composed components parameterized by the concerns addressed.

2.1 Programming Composed Components

HCL describes a collection of units, described as entities that carry a specific task. Units are instantiated from interfaces that describe a collection of input and output ports and a protocol that specifies the order in which these ports must be activated. The activation of a port enables it to complete a communication. Interfaces may be declared separately allowing their reuse. Reusable interfaces are called interface classes. The interface of a unit describes how it interacts with the coordination medium. An interface may be composed from other interfaces. An interface slice is a part of the interface inherited from other interfaces. The task carried by the unit is described by assigning a component to it. The concern addressed by the assigned component defines the role of the unit. The input and output ports of the unit should be (partially) mapped onto the arguments and return points of the component.

A unit may replicate a port. Groups of ports may be of two kinds: any or all. The activation of groups of ports of kind any is inspired by the semantics of the ALT constructor of OCCAM [18]. The activation of groups of ports of kind all enables all the ports belonging to the group. Wire functions may be specified in the boundary between ports, or groups of ports, and arguments/return points for forcing their compatibility.

Ports must be connected through point-to-point, unidirectional and typed communication channels, with three possible modes: synchronous, bounded buffered and ready.

Units may be unified or factorized, using the slices of their interfaces as decomposition units. It is also possible to replicate sub-networks of a unit topology. Behavioral and topological preserving restrictions exist for ensuring the coher-
ence of the network topology of a unit in the presence of these operations.

2.1.1 Virtual Units and Skeletons
Units with no component assigned to are called virtual. They allow to parameterize the concern addressed by a component. A partial topological skeleton (PTS), or abstract component, is a composed component formed by at least one virtual unit. The use of skeletons neatly allows parallel programming to take advantage of specific architectural features tuning code without losing the overall program portability.

2.2 Programming Simple Components
Functional modules are programmed in a host language. To bind a host language to the # programming environment, it is necessary to map the host language constructors and input and output variables of functional modules. No extension to the host language is necessary for this purpose, keeping the orthogonality between coordination and computation medium. Our goal is to implement a really multilingual approach for parallel programming either on top of CCA (Common Component Architecture) [1], a recent standard proposed for integrating components written in different languages in a parallel environment, or the interoperable implementations of MPI [27].

The approaches for programming functional modules depend on the kind of host language employed. In lazy functional languages, such as Haskell, arguments and return points correspond to the arguments and elements of a tuple returned by a function main in a module. Lazy lists may be used to model stream communication, allowing the overlapping communication and communication [7].

In procedural languages, such as C and Fortran, functional modules are treated as abstract data types. The encapsulated data structure models the computational state of the process. The functions that access the data structure correspond to the return points of the component. The value produced by a function is transmitted through the port associated with the return point. There is a correspondence between the parameters of the return functions with the arguments of the component. The same argument may be referred in more than one parameter of return function. Special functions, called guard checkers, are needed to implement choices in the alt constructor of the unit protocol. The use of the par combinator allows to execute more than one return function simultaneously. We intend to use this approach to implement multiprocessor parallelism by using openMP. It has been suggested to enrich the set of protocol combinators in order to support loop parallelization and other techniques implemented by openMP. The # programming environment may be able to exploit multiple levels of hierarchy of parallelism [3].

In object oriented languages, such as JAVA and C++, functional modules are objects. Return functions are implemented as methods and arguments as method parameters. Procedural and object oriented functional modules may be overlapped. In the first case, using C language as an example, it is possible to use external clauses for accessing the encapsulated data structure of other modules. In the second case, multiple inheritance, aspect programming [19], or hyperspace [23] programming may be used.

3. APPLICATION CASE STUDY: CSM
Figure 1 presents the topology of the CSM (Climate System Model) component. CSM has been developed at NCAR (National Center for Atmosphere Research), in the USA, in Fortran/MPI. CSM involves parallel execution of five processes. Four of them implement climate models for atmosphere, biosphere, hydrosphere, and cryosphere. They interact by exchanging messages with an intermediate unit, the coupler.

In # programming, the implementation of the CSM models and the coupler may be encapsulated in components that are assigned to units atm, ind, ocn, ice and cpl, whose interfaces and interconnectivity describe the pattern of parallel synchronization between the models and the coupler. Climate components may be implemented as composed or simple components. This is transparent from the perspective of CSM configuration. Indeed, it is possible to use distinct versions of the models without modifying the configuration of CSM. If a model is implemented as a composed component, then it is also a distributed parallel program. This approach, where all models are parallel programs allows CSM to be implemented in a grid environment and each model to be placed in a cluster within the grid. The configuration of each model specifies the distribution of processes onto the cluster nodes. On the other hand, if CSM is executing in a cluster, models may be implemented as (sequential) simple components, where each model is placed in a cluster node for sequential execution.

4. EXAMPLES
The next three sections examine examples of applications where # programming may significantly improve the ability to modularize cross-cutting concerns. They are: allocation of processes onto processors, parallel I/O operations and parallel “debugging”.

4.1 Allocation Policies and Load Balancing
The placement of processes onto processors may be considered the most important non-functional concern in parallel programming. A good allocation policy ensures good load balancing of processors. In static allocation policies,
configuration Fast_Ethernet<N> where

unit processing_node where ports : in → out
    unit hub where ports : in → out
connect processing_node → out to hub in, in, buffered
connect hub → out to processing_unit in, buffered
replicate N processes node grouping: hub.out any, hub.in any

Figure 2: An Architecture Description Component

configuration PadmeC_Cluster where
use Architecture (Fast_Ethernet, Processor_INFO)
define NPROCS 16

iterators r range [1,NPROCS]
iterators w/n range [NPROCS/2]
iterators o range [NPROCS/2,NPROCS]

unit padmeC_cluster
    assign Fast_Ethernet<N> to padmeC_cluster
    \{ alias w[i] for processing_unit[i] \}
    \{ assign Processor_INFO<"Pentium IV", 1.6, 100, 1024> to w[i] \}
    \{ assign Processor_INFO<"Pentium IV", 2.4, 128, 1024> to w[i] \}

Figure 3: A Component for a Cluster

processes remain in a processor until the end of execution, while dynamic allocation allows the migration of processes between processors. Despite its cross-cutting nature, it is simple to modularize the placement specification from the functional concerns of the application. MPI and PVM, for example, define allocation policies in separate configuration files.

In # programming, allocation policies may be encapsulated in skeletons. Each virtual unit implemented models a processor or class of processors. By unifying a process (unit) of the main component with a virtual unit of the allocation policy skeleton, the compiler may infer that the process must be allocated to the processor, or a processor from a certain class, represented by the virtual unit. In the resulting # topology, the processor unit is a slice of the process defining its role in the placement concern. It is possible to use communication channels to define the connectivity of processors (physical channels), but this information may be only useful for teaching purposes, for instance. In a Fast Ethernet based cluster architecture, for example, it is possible to define a hub as a unit. Each unit that represents a processor has one input and one output port to the hub. Figure 2 illustrates a general skeleton that represent the topological structure of a Fast Ethernet-based cluster.

It is possible to define new components from the Fast_Ethernet skeleton for modeling specific clusters. In Figure 3, the simple component Processor_INFO is assigned to the units of skeleton PadmeC_Cluster, configuring parameters of each processor that characterize the processing units and may be useful for compilers when generating code.

Figure 4 shows how processes of CSM component may be allocated to processors modelled in PadmeC_Cluster skeleton. A process is unified with a unit representing the processor where it will be placed. It is possible to place several processes onto a processor by replicating the unit representing the processor. For supporting portability, programmers may keep decoupled the component that represents the functional processes of the program from the components that represent their non-functional concerns. The porting of a # program between architectures needs only to overlap the network of functional processes with another component representing the new architecture. Indeed, it may be possible to overlap it with several components representing architectures and to inform which component to adopt as an allocation skeleton at compile time.

In large-scale grids, the topology of the network of processors is dynamic. It is possible to add or remove processors, or a processor to become unavailable. In this case, it would be desirable to keep concerns about identification of individual processors transparent to the programmer. For that, a semi-implicit approach could be adopted, where units in the allocation policy skeleton represent a class of processors. Whenever a process is unified with one of these units, the compiler may choose one of the processors belonging to that class for allocation.

4.1.1 Dynamic Allocation Policies

In certain loosely-coupled parallel architectures, such as grids, the ability to move processes between processors may be supported for improving the efficiency and fault-tolerance. Despite the static nature of # programming, it is possible to provide support for dynamic allocation of processes onto processes. However, this is not an issue concerning # programming. From this perspective, allocation is static. It may be possible to define an initial configuration using the explicit approach described in the previous section for defining the initial placement of processes, and let the run-time system to control migration of processes whenever necessary. It may also be possible to use the implicit approach in order to give information for the run-time system about demands of processes and processor characteristics. Indeed, it is possible to let the run-time system to control allocation, in a transparent way to programmers.

4.2 Parallel I/O

Collective I/O operations on disks constitute an important parallel programming concern. In scientific applications, it is common to perform computations on immense collections of raw data. Only disks have enough capacity and reliability for this data. Indeed, it is recognized that performance of I/O operations is a “bottleneck” for performance of certain parallel programs. CSM is an example of application for which performance of I/O operations on disk is crucial. The processes ocn and ice implement static models that make use of data obtained from real climate measures stored in
files from MSS (Mass Storage System). Special libraries for improving I/O performance in high-performance parallel programs have been proposed, such as MPI-IO [25].

In conventional parallel programming approaches, code for performing collective I/O operations on disks is scattered along the code of processes. It is necessary to inspect the code of all processes for discovering which ones are involved in collective operations, turning problematic maintenance procedures of large scale parallel programs.

In # programming, concerns relating to I/O operations may be encapsulated in components. Figure 5 presents one basic topological abstraction for a # component addressing I/O concerns. The unit user represents an agent that performs I/O operations on the device represented by the unit device. Here, a device may be a disk, a screen, or any other I/O device that can appear in a parallel program. The communication channels represent the flow of data between the user process and the device in I/O operations. Ports open and close in the user process are used for opening and closing the device. The other ports, read and write, represent basic input and output operations on the device, respectively. This is a simplistic specification for illustrative purposes. Several other operations, or variations of the existing ones, supported in certain devices could be represented as ports.

User units are virtual, while device units are not. This is justified by the fact that the basic functionality of device units is defined a priori. It is a known that disks implement I/O operations on the device represented by the unit device.

Using replication, it is possible to represent complex arrangements of devices and processes for representing collective I/O operations. By replicating the device, it is possible to model synchronous I/O operations performed by a user agent on a collection of devices. The nature of operations is influenced by the kind of group of ports derived from replication in the user process. For instance, in a operation represented by a group of ports of kind all, the operation occurs simultaneously on all devices. In a operation of kind any, a collection of processes performs a collective I/O operation on the device. In a operation of

---

1MSS is a central, large-scale data archive that stores data used and generated by climate models and other programs executed on compute servers at the National Center for Atmospheric Research of USA (NCAR).

---

**Figure 5: I/O Component Topology**

**Figure 6: CSM with I/O**
kind any, only one from a collection of processes that are ready to complete the operation completes the operation. By replicating both individually, it is possible to model a collection of user’s performing collective operations on a collection of device’s. By replicating both simultaneously, it is possible to model a collection of users performing independent operations on their respective devices.

An experienced parallel programmer may imagine several possible arrangements that may be modeled in a # parallel program using the approaches described above. By applying assignment operations on user units or by unifying existing units in a configuration with user units, it is possible to specify which task is performed by user agents. The configuration of a device unit is performed by unifying it to a unit for which a specific simple component was assigned, whose static parameters are used for configuring the device. For example, in this paper, it is introduced the simple composition "debugging" parallel programs that will be motivated and discussed in the following paragraphs.

The “debugging” of # programs is performed in separately at the coordination and computation levels. At the coordination level, the static and dynamic properties and correctness of synchronization protocols among processes are analyzed by using automatic tools for Petri nets [22]. All information regarding coordination medium semantics in a # program may be modeled using Petri nets [10]. Several automatic tools exist for simulation, analysis and verification of formal properties of models specified using this formalism [17]. At the computation level, once each simple component may be viewed in isolation, it is possible to use “debugging” techniques appropriate for the language paradigm used for programming each process. For instance, processes implemented in Haskell, a purely functional programming language, may be formally analyzed by using algebraic reasoning. On the other hand, in imperative and object oriented programming languages, where side-effects may exist, “debugging” is normally performed by inspecting the current state of the program, represented by the value of global variables, placing assertions in certain verification points in the code. In this sense, a “debugger” may be considered a non-functional cross-cutting concern, scattered along the code of processes comprising the parallel program. In conventional parallel programming, it is impossible to modularize “debugging code”, but # programming allows to apply the notion of overlap of components for that purpose.

For demonstrating how to implement a modular “debugger” for a # program, consider again the CSM application. Figure 8(a) exhibits the configuration code of the component named CSM_DEBUGGER. It implements a (parallel) “debugger” for CSM. The simple components (modules) assigned to the “debugger” processes implement verifications of their states in execution. The “debugger” component is implemented by overlapping instances of components IOWRITEONLY, once the “debugger” presents execution traces to users through a screen terminal, and BCAST, once the coupler, at certain steps, broadcasts to climate models information about its state. This is justified by the fact that, in parallel “debugging”, it may be necessary to test some global assertions about the state of the parallel program that refers to variables from many processes.

Figure 8(b) presents the configuration code that overlaps components CSM and CSM_DEBUGGER for defining the component CSM_WITH_DEBUGGER. As discussed in Section 2.2, overlapping merges the modules assigned to the unified processes. Modules of the CSM “debugger” access the global variables of CSM functional modules by employing the host language support for referencing global variables declared in other modules, such as external declarations in C language. The diagram in Figure 7 illustrates composition steps of CSM_WITH_DEBUGGER. Notice that each one of the final processes (cpl, atm, ice, ocn, and ind) has slices that define their roles in the concerns addressed by components BCAST, IOWRITEONLY and CSM, that are overlapped for composing the final topology. In a process, a slice may be replaced, modified or removed without affecting other slices. Besides that, the insertion of a new slice, by overlapping a new component implementing a new concern, should not affect existing slices.

The use of I/O components for modular description of I/O concerns may be translated directly in operations using high-performance I/O libraries. For example, collective I/O operations may be translated in calls to collective I/O operations of MPI-IO. Optimizations may be applied whenever possible. It may be expected that the same, or best, hand tuned code can be generated by # compiler.

The basic component abstraction presented in Section 5 may be enriched for supporting variations of I/O operations supported in specific libraries. Also, it is possible to ignore some operations by simply not activating their corresponding ports in the protocol of processes.

Figure 6 presents how I/O concerns in CSM application may be modularized using # components overlapping components. There are two main concerns to be addressed. The first one specifies I/O operations performed by ocn and ice units on MSS, while the second one corresponds to operations periodically performed for output computation status during execution of each model in a terminal device. The component MSS illustrates the use of a specific I/O component for performing I/O operations in MSS, derived from generic I/O components.

4.3 Parallel “Debugger”

In large-scale parallel programming, the task of detecting and correct programming errors is very hard even for experienced programmers. While in sequential programming one concentrate only on the analysis of a single control flow, in parallel programs several independent control flows exist. The number of execution paths to analyze increases exponentially with the number of processes. Additionally, it is necessary to “prove” the correctness of the communication/synchronization interaction patterns, avoiding deadlocks and buffer overflows, maximizing parallelism, etc. One technique widely adopted for reducing the complexity of parallel programming “debugging” task is to write parallel programs in SPMD (Single Program, Multiple Data) style. The # programming also supports programming in SPMD-style programming [8], but it provides powerful artifacts for “debugging” parallel programs that will be motivated and

The basic component abstraction presented in Section 5 may be enriched for supporting variations of I/O operations supported in specific libraries. Also, it is possible to ignore some operations by simply not activating their corresponding ports in the protocol of processes.

Figure 6 presents how I/O concerns in CSM application may be modularized using # components overlapping components. There are two main concerns to be addressed. The first one specifies I/O operations performed by ocn and ice units on MSS, while the second one corresponds to operations periodically performed for output computation status during execution of each model in a terminal device. The component MSS illustrates the use of a specific I/O component for performing I/O operations in MSS, derived from generic I/O components.

4.3 Parallel “Debugger”

In large-scale parallel programming, the task of detecting and correct programming errors is very hard even for experienced programmers. While in sequential programming one concentrate only on the analysis of a single control flow, in parallel programs several independent control flows exist. The number of execution paths to analyze increases exponentially with the number of processes. Additionally, it is necessary to “prove” the correctness of the communication/synchronization interaction patterns, avoiding deadlocks and buffer overflows, maximizing parallelism, etc. One technique widely adopted for reducing the complexity of parallel programming “debugging” task is to write parallel programs in SPMD (Single Program, Multiple Data) style. The # programming also supports programming in SPMD-style programming [8], but it provides powerful artifacts for “debugging” parallel programs that will be motivated and discussed in the following paragraphs.

The “debugging” of # programs is performed in separately at the coordination and computation levels. At the coordination level, the static and dynamic properties and correctness of synchronization protocols among processes are analyzed by using automatic tools for Petri nets [22]. All information regarding coordination medium semantics in a # program may be modeled using Petri nets [10]. Several automatic tools exist for simulation, analysis and verification of formal properties of models specified using this formalism [17]. At the computation level, once each simple component may be viewed in isolation, it is possible to use “debugging” techniques appropriate for the language paradigm used for programming each process. For instance, processes implemented in Haskell, a purely functional programming language, may be formally analyzed by using algebraic reasoning. On the other hand, in imperative and object oriented programming languages, where side-effects may exist, “debugging” is normally performed by inspecting the current state of the program, represented by the value of global variables, placing assertions in certain verification points in the code. In this sense, a “debugger” may be considered a non-functional cross-cutting concern, scattered along the code of processes comprising the parallel program. In conventional parallel programming, it is impossible to modularize “debugging code”, but # programming allows to apply the notion of overlap of components for that purpose.

For demonstrating how to implement a modular “debugger” for a # program, consider again the CSM application. Figure 8(a) exhibits the configuration code of the component named CSM_DEBUGGER. It implements a (parallel) “debugger” for CSM. The simple components (modules) assigned to the “debugger” processes implement verifications of their states in execution. The “debugger” component is implemented by overlapping instances of components IOWRITEONLY, once the “debugger” presents execution traces to users through a screen terminal, and BCAST, once the coupler, at certain steps, broadcasts to climate models information about its state. This is justified by the fact that, in parallel “debugging”, it may be necessary to test some global assertions about the state of the parallel program that refers to variables from many processes.

Figure 8(b) presents the configuration code that overlaps components CSM and CSM_DEBUGGER for defining the component CSM_WITH_DEBUGGER. As discussed in Section 2.2, overlapping merges the modules assigned to the unified processes. Modules of the CSM “debugger” access the global variables of CSM functional modules by employing the host language support for referencing global variables declared in other modules, such as external declarations in C language. The diagram in Figure 7 illustrates composition steps of CSM_WITH_DEBUGGER. Notice that each one of the final processes (cpl, atm, ice, ocn, and ind) has slices that define their roles in the concerns addressed by components BCAST, IOWRITEONLY and CSM, that are overlapped for composing the final topology. In a process, a slice may be replaced, modified or removed without affecting other slices. Besides that, the insertion of a new slice, by overlapping a new component implementing a new concern, should not affect existing slices.
The CSM_Debugger is a simple example that motivates how the “debugging” concern, a inherently cross-cutting concern in a parallel program, may be modularized. But, in # programming, several other modular debugging strategies may be implemented using the ability to overlap components. Indeed, it is possible to create general “debuggers” for classes of applications or that may be applied to subsets of processes in a parallel program. It is also possible to overlap several debuggers, encapsulating distinct specific “debugging” concerns.

The inheritance notion supported by object oriented languages may be very useful for defining “debuggers” for specific classes of applications. A structure of basic classes may be used for encapsulating common variables that exist in every application in the class. A specific “debugger” may be instantiated from basic classes, concentrating only on its encapsulated variables.

5. CONCLUSIONS
The advent of cluster and grid computing have led to higher complexity and scale demands for high performance applications. Traditional parallel programming methods and tools, based on low-level message passing, offer poor abstraction and modularity support for the development of large-scale parallel programs. In software engineering, modularization and abstraction mechanisms based on encapsulation of concerns in modules have been traditionally employed in the development of large-scale programs. In the last two decades, software engineering practice has demonstrated that module encapsulation does not allow modularization of inherently interlaced concerns [19].

The # model was proposed to meet the demands of the high performance computing community for integrating parallel programming techniques with modern large scale software engineering techniques [26]. In order to demonstrate the # model ability to work with separation of encapsulated and cross-cutting concerns, three common examples from parallel programming practice were used.

The work with the # model appears in the context of the development of a higher level parallel programming environment for the development of large scale parallel programs targeted at cluster and grid architectures, supporting the analysis of formal properties analysis and performance evaluation of programs using the Petri net formalism and its high level variants. A prototype, named VHT (Visual # Tool), that provides visual support for # programming at coordination level, with support for overlapping components, has been implemented in Java.

6. REFERENCES
configuration CSM_Debugger where
#define MAIN_terminal "150.14.12.2"
use Applications.CSM.Simple Debugging (stm, scw, sw, lw, mG, t)
use IO General t0WaitOnly
use Collective Box
unit debug_output: assign t0WaitOnly to debug_output
unit debug_bcast: assign Box &gt; to debug_bcast
replicate 5: debug_output user
assign iMainTerminal(MAIN_TERMINAL) to debug_output device
interface ICSMDebugger where ports: Interface &gt; bc dbg msg #
BCast
IBCast
unify
unify
unify
unify
interface IModelDebug where ports: Interface &gt; bc dbg msg #
BCast
IBCast
Applications.CSM.Debugging.
Applications.CSM.Functional.
use configuration
assign
assign
unify
interface ICSMDebugger
replicate 5: debug
unit debug
Collective.
use
use
Applications.CSM.Simple.Debugging.
Applications.CSM.Functional.
use configuration
assign
assign
unify
Figure 8: A "Debugger" for CSM


