[1]GIG-Pattern

Maria Lencastre (mlpm@cin.ufpe.br)[2]
Felix C. G. Santos (fcgs@demec.ufpe.br)
Mardoqueu Souza Vieira (msv@cin.ufpe.br)
Mechanical Engineering Department, Federal University of Pernambuco
Rua Acadêmico Hélio Ramos, S/N, Recife, PE 50740-530 – Brazil

Abstract

This paper presents a pattern called GIG, a generic interface graph which deals with definition and control of processes taking into account some specific requirements of simplicity, easiness of definition from algorithmic language and flexibility in the granularity of defined processes. The pattern is intended to help the design and reuse of programs.

1. Introduction

The use of workflow technology helps the development of more flexible and versatile computation strategies. So, workflow management systems are a relevant support for large class of business applications, and many workflow models as well as commercial products are currently available [8]. While the large availability of tools facilitates the development and the fulfilment of customer requirements, workflow applications still require simple, generic and adaptive solutions for the complex task of rapidly producing effective applications, especially when complex domains are involved.

The GIG pattern was developed after we noticed that many numerical algorithms showed the very same organizing structure when trying to achieve process reuse and flexibility for the adaptation to new strategies. Such an organizing structure in turn allowed for an abstraction, which resulted in the GIG, a generic interface graph. As it will be seen, it is possible to devise frameworks to use the GIG pattern in order to implement different processes in a very flexible and automatic way.

The GIG-pattern describes an abstract workflow solution, whose purpose is to provide expressiveness and adaptability through simplified workflow programming, control and use [8]. Other GIG motivation is to maintain predefined algorithmic structure, which means that the translation from algorithmic language representation of the processes into a computer representation must be as direct as possible. This is important because, the achievement of similarity between the way the programmer has its algorithmic code organized and the implementation of it, can bring simplification in further required changes. Also, sometimes, developers need solutions that does not make restrictions on the scale of the process, that is, which need a mixture of small-scale processes (that execute within applications) and large-scale processes (that execute on top of applications), usually this happens when designers are also the programmers.

As a workflow pattern, GIG provides for the separation of process logic from task logic, which is embedded in user applications, allowing the two to be independently modified and the same logic to be reused in different cases. The GIG-pattern considers features related to run-time control functions [7], which manage the workflow processes and sequence the various activities.

This work was devised from the experience obtained during the implementation of several simulators in the FEM context [5]. FEM is a way of implementing an approximate mathematical theory for a physical behaviour. Researchers of the Mechanical Engineering Department – UFPE – Brazil found the need to organize their code in a way that was easier to adapt to new strategies and also to allow process reuse. So they designed and implemented the GIG, a generic interface graph,

---

which provides an interface for process control dealing with the specific requirements mentioned. In this paper the GIG is presented as a pattern.

The pattern's description is organized in the following way. In section 2 the pattern name is identified. Section 3, details the context in which the pattern solution applies. Section 4 presents a motivation example for the GIG-pattern use. Section 5 presents the design challenge through a question. Section 6 shows pattern forces, that is, the patterns design trade-offs, what pulls the problem in different directions, towards different solutions. Section 7 explains how to solve the problem. Section 8 describes the pattern implementation. Section 9 presents some variants that can extend the pattern. Section 10 presents a simple example of use, in order to clarify the pattern use and section 11 presents a more complex one in the FEM simulators context. Section 12 details the resulting context, telling which forces the pattern resolves and which forces remains unresolved by the pattern. Section 13 presents related patterns. Finally, section 14 talks about known uses.

2. Name: GIG-Pattern, Generic Interface Graph for ProcessC ontrol.

3. Context

Domain specific users, like scientists and engineers, usually program in a procedural style. The explanation for that, in spite of the force of tradition, may be the following. Complex numerical systems usually make use of many different pre-built auxiliary packages (like numerical integrators, solvers for non-linear and linear systems of algebraic equations, and so on) and have their procedures described in algorithmic language. So, the majority of the work is related to making the modules compatible in a monolithic architecture, which resembles the structure of the algorithm. This is a strong force that drives those users towards the procedural style.

During the development of a software system, those developers need functions that help them to organize their logical processes and their involved tasks, in a way that makes easy its future alteration for adapting to new solutions and for the reuse of software routines, avoiding heavy reprogramming. We have repeatedly noticed that many numerical algorithms showed the very same organizing structure. Such an organizing structure comes from the procedural style of the algorithm representation and can be identified to be a Directed Acyclic Graph (DAG) [5].

4. Motivation Example

Consider, for example the case of mesh generation algorithm. A mesh can be described as a partition of a geometric domain into simple geometric entities (triangles, tetrahedra, hexahedra, etc) called geometric finite elements (or simply elements). In Figure 1 we present the algorithm for a particular mesh generation, which, given a plane straight-line graph (PSLG), generates a mesh of triangles.

```
I.  Data input (PSLG)
II. Generate the bounding box for the PSLG
III.Build the initial mesh of the bounding box
IV. For each point in the PSLG do
    IV.I.  Insert point
        IV.I.I.  Find elements affected by the new point
        IV.I.II. Eliminate those elements obtaining the affected region (AF)
        IV.I.III Build new elements from the new point and boundary of AF
V.  Find a line of the PSLG such that it is not an edge of any triangle
  (negative line)
VI. While there still is a negative line do
        VI.I Compute the middle point of the line
        VI.II insert middle point (see IV.I)
VII. Eliminate those triangles, which have any point of the bounding box as
     one of their vertices.
VIII.Data output
```

Figure 1 Mesh Generation Algorithm

This algorithm can be represented using the graph structure presented in Figure 2. Observe that there are fifteen sub-routines, including the driver (which executes the procedures I- VIII). This graph structure can be represented in GIG-pattern (see Figure 3). Each one of those processes can be encapsulated in an object of a class, representing a node of the graph. The proposed pattern describes it as a derivation of a base class called *AlgthmNode*.

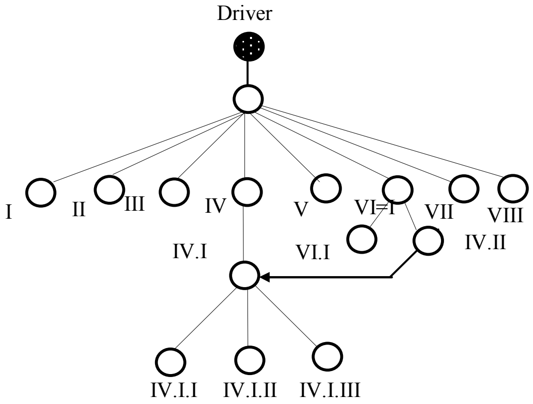

Figure 2 Mesh Generation Graph

Observe that there are many different ways of performing each one of the activities (tasks) described in the above algorithm. For instance, *IV.I.I find elements affected by the new point* concerns a search method in a geometric database of triangles, looking for a triangle whose circumscribed circle contains a given point. There are a lot of search methods available in the specialized literature, each one with its advantages, drawbacks and dependence on special data structures. Replacing the current method by a new one will not affect any other place in the graph.

Entire branches can also be changed as well. For instance, the process *IV.I. insert point*, can be changed by plugging another method to perform that task. That means that all the subsequent processes (children nodes) will be also changed. Besides the severity of the change in the methods needed by the algorithm, all the substitution work can be automatically performed.

On the other hand, the Data Domain of this problem can be decomposed in such a way that all *AlgthmNode* objects (subroutines) will have access only to the data it needs. For instance, the process *III.Build an initial mesh for the bounding box* will need the bounding box and will build the initial mesh, which will be stored in a place in order to be accessed by other nodes. That decomposition will give rise to the classes derived from AlgthmData. The whole set of data pieces depend on the geometric data structure used by the developer. For instance, it can be seen that some structures have to be present: (a) PSLG (accessed by I, II, IV and V); (b) bounding box (accessed by II, III and VII), (c) mesh (accessed by III, IV.I.I, IV.I.II, IV.I.III, V, VII and VIII), (d) auxiliary data (many, it depends on the designer). All those pieces of data will be encapsulated in objects of classes derived from *AlgthmData*.

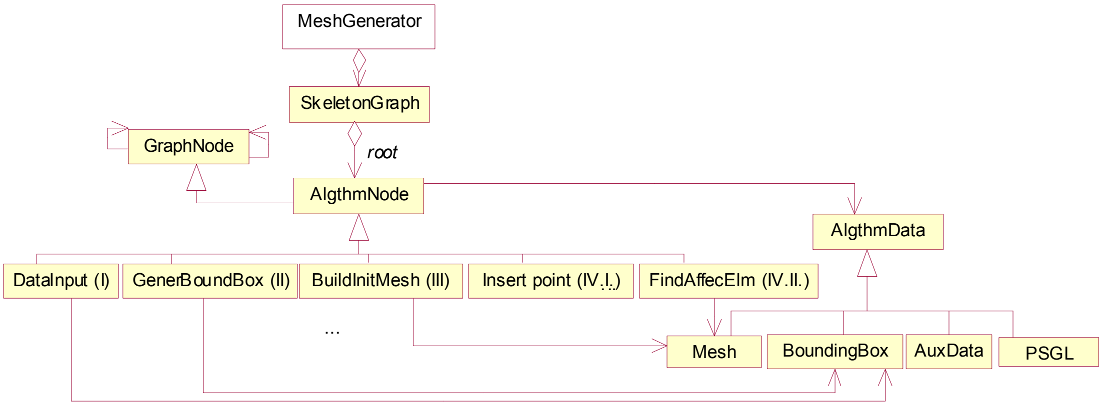

Figure 3 Application of the GIG structure in Mesh generation algorithm

In this example the mesh generation process is the controlled workflow. This process includes information about constituent tasks (represented as the processes (I to VIII). The mesh generation process has requirements related to modularity and exchange of sub-routines, since it has specific parts that have several kinds of implementations, which can be exchangeable.

5. Problem

How to guarantee simplicity in the separation of process logic from task logic, during the development of complex systems, while maintaining solution independence, reuse of processes and the predefined algorithmic structure?

6. Forces

With respect to the defined context, there are different forces, which lead to different solutions. Some of these forces are:

- Maintaining predefined algorithmic structure;
- Simplicity in the process definition;
- Support for different levels of granularity on the defined processes;
- Domain independence;
- Dynamic change of workflow processes;
- Reduction on error occurrences in the coupling of processes;
- Reuse of processes;
- Parallelism and processes synchronization;
- Workflow execution performance;
- Explore existing expertise of domains of knowledge.

The following discussion analyses some of these forces, in order to identify how they are pulling against each other. GIG tries to resolve some opposing forces in the workflow definition context.

When trying to maintain the predefined algorithmic structure, the definition of some sub-process could generate pieces of code that are not easily changeable, because are monolithically defined as a block of code. On the other hand, refined levels on process portioning can provide a process definition at statement level, eliminating existing abstractions (like blocks or modules). Domain independence and dynamic change of process requires abstractions like polymorphism and encapsulation, which are not present in a procedural style (the predefined algorithmic structure).

The guarantee of simplicity in process definition can be one way to avoid errors and stimulate the pattern use. The reuse of already developed and tested processes helps in the simplification of process definition, like the possibility of reusing entire solutions. However, the reuse of processes can also reduces the simplicity due to the need for extensions of classes or configuration. Some other opposing forces to simple process definition are: the guarantee of domain independence, which makes more complex the process definition; also, to allow the definition of processes parallelism and synchronization the programmer has to deal with extra levels of complexity. The simplification can be compromised when parallelism is required for increasing performance.

Process reuse improves reduction of errors once pre-tested software is incorporated. Refined levels of granularity, in process definition, provide higher level of tangibility in the number of processes to be controlled, increasing the reuse of processes. The guarantee of domain independence also increases the number of reusable process.

Domain independence, avoiding non-monolithic solutions, makes possible the application of the workflow solution to different applications, improving its reuse. However, in these cases the existing expertise of a knowledge domain cannot be appropriately explored to improve the solution. Also, synchronization and parallelism improve in one-way domain independent application supporting the required functionality to existing applications.

The dynamic change of workflow processes improves the solution power, however, gives the programmer the responsibility and the complex task of making a suitable division of code and data, for further exchanging to be pertinent. The reuse of process is fundamental when the user has to change an existing one by another one, which is already tested and classified. Maintaining the pre-defined algorithmic structure does not help domain independence because it does not provide, for example, encapsulation.

Parallelism and processes synchronization are very relevant to allow system optimization and higher levels of control. The refined levels of granularity, in process definition, can allow a more precise level of parallelism definition. The dynamic change of workflow and the reuse of process increase the synchronization power, in process exchanging. Synchronization and parallelism improve in one-way the power of the dynamic change of process (identifying which process are independent or the order of the dependence). On the other hand this can arises more complexity in the changing of processes.

Maintaining the predefined algorithmic structure can sometimes improve performance due to the direct application of some available optimized code; parallelism also improves performance, since allows simultaneous execution of process. On the other hand, simplicity on process definition can decreases the performance, when it eliminates, for example, the possibility of parallelism definition. The guarantee of domain independences can also decrease performance once the existing expertise cannot be appropriately explored. Other forces which compromise the performance, due to the need of extra verification and controls, are: refinement level of the granularity of process definition; dynamic process exchange requires more controls; control of errors, reuse of process reduces the performance, synchronization.

7. Solution

GIG can be described as a workflow solution [7]. GIG follows the object-oriented style for modelling and programming. For simplicity reasons of use and facility in correctness verification, GIG implements a restricted Direct Acyclic Graph (DAG) [5].

7.1 Participants (Structure)

The GIG structure is presented in the UML diagram below (Figure 4).

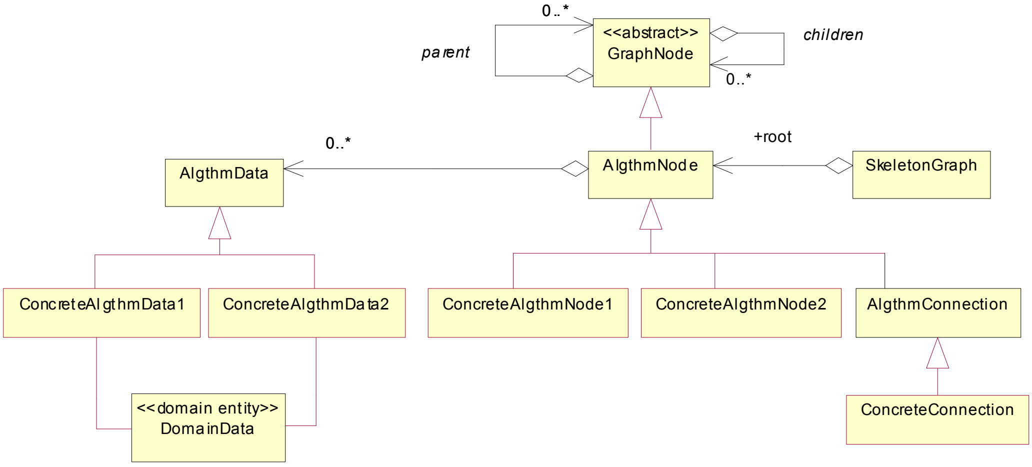

Figure 4 Participants of the GIG-pattern

The GIG pattern is composed of the following participants:

- *GraphNode*: it is an abstract class that implements low level operations related to the interoperability between graph nodes. It controls the relationship between the workflow activities.
- *SkeletonGraph*: it has a reference to the driver of an algorithm graph and encapsulates tools for performing some graph operations. It can be seen as the root of workflow process.
- *AlgthmNode*: represent subroutines that compose the application (workflow tasks) . It is used as a base class for all algorithm classes of the application.
- *ConcreteAlgthmNode* Implements a specific subroutine for a task. It invokes other subroutines which can be tasks (defined as its children) or other defined applications.
- *AlgthmData*: represents a data type to be used by an instance of *AlgthmNode.* It is used as a base class for all algorithm data classes of the application.
- *ConcreteAlgthmData* Represents data from the application domain, that is used in *ConcreteAlgthmNode* classes.
- *DomainData*: represents the whole set of types related to the problem domain data.
- *AlgthmConnection*: it is an *AlgthmNode*, which references an algorithm subrotine that was not connected to the graph. This class responsibility is to fetch, and build (like a proxy [10]) the related algorithm and replaces itself with the fetched algorithm. In this way several software processes represented by *SkeletonGraph*s can be assembled producing a complex software system.

7.2 Collaborations

We can identify the following collaborations between GIG participants:

- *GraphNode* encapsulates the responsibility of providing access to other *GraphNodes*, which are its children.
- *ConccreteAlgthmNode* executes the associated process (subroutine) with the help of other processes represented by its children, through calls inserted in its process code. It relies on *GraphNode* to have access to its children *AlgthmNodes*.
- *ConcreteAlgthmData* provides access to workflow data. *AlgthmNode* communicates with Concrete*AlgthmData* objects to have access to its data.
- *AlgthmConnection* provides the dynamic connection for *AlgthmNodes*. The way objects of this class interact with its *SkeletonGraph* or its parents *AlgthmNodes* depends on the implementation. The important thing is that it represents the point where a driver node of a software process/subroutine will be plugged in. It also contains the necessary information about the new *AlgthmNode*.

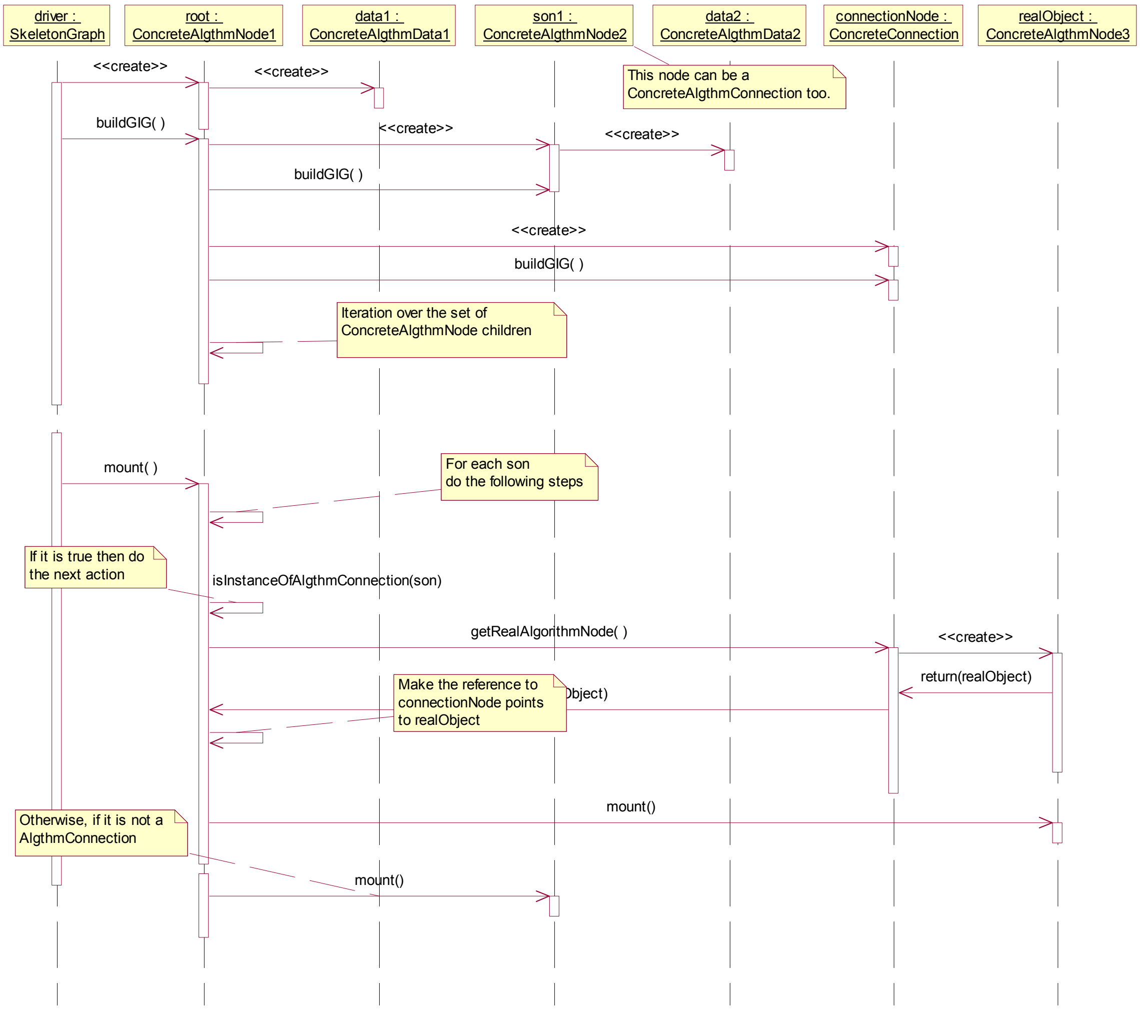

Figure 5 Sequence diagram for GIG building

We can summarize the main part of GIG-pattern interaction, through the UML sequence diagram of Figure 5. The GIG driver, an object of *SkeletonGraph* class, creates the GIG root that is an instance of *AlgthmNode* class. When the *SkeletonGraph* requests the root to build the GIG, it instantiates each child and asks them to build its sub-graphs recursively. After, the graph building the driver waits for requests for GIG mounting, CIG execution, GIG reprogramming, and for other graph manipulation functionalities. The GIG mounting is the act of remove the references to *ConcreteAlgthmConnection* objects and so preparing the GIG for execution. The GIG execution is similar to the GIG building procedure by changing *buildGIG* calls for *execute* and without <<*create*>> calls, during the execution each *ConcreteAlgthmNode* its subroutine accesses the *ConcreteAlgthmData* objects associated with it's node.

8. Implementation

There are some implementation issues associated with the GIG participants, described previously, who need some extra explanation. Other important details about implementation are related to the design steps to be followed by the user for applying the GIG-pattern to a new application. In section 9, and 10 these steps are exemplified.

8.1 Implementation Issues

The *DomainData* is implemented by a set of subclasses of the *AlgthmData*. The subclasses of *AlgthmData* describe the specific domain treated in the problem. An example of the partition of the domain in different levels can be seen in the section 11.

The *AlgthmData* and *AlgthmNode* objects must be materialised for the workflow they are serving. The materialization activities, of *AlgthmData* and *AlgthmNode* objects, can be delegated to object factories that are responsible for accessing the data repository and instantiating the objects, these object factories can have object pools to reuse objects, see section 13 for details about the patterns that can be applied.

The *AlgthmNode* subclasses need to cast the *AlgthmData* objects, associated with each node, to the primitive type.

As was shown each *AlgthmNode* object must have a reference for all its children and data. This reference can be hard coded in *AlgthmNode* subclass, or in a file, or can be handled by another class, wihich has the responsibility to relate each *AlgthmNode* with its children. An example of such a class is *DataAlgthmServer* use in the example in section 10. In this case each *AlgthmNode* can ask to the *DataAlgthmServer* for its children and data or the *DataAlgthmServer* can be active and responsible to build the GIG.

8.2 The Design Steps

The following design steps describe which actions the user needs to perform to apply the GIG-pattern to a problem:

1) Starting from an algorithm in natural language the procedure is first divided into different algorithm sub-routines (algorithm nodes) and then it is organized in the form of a graph.
2) The division of the algorithm into several routines induces a decomposition of the domain data in order to provide them with an appropriate distribution of access to the data. The result of this process gives the *AlgthmData* set.
3) Each *AlgthmNode*, that is an algorithm sub-routine, places calls to its children nodes, which implement processes inside the whole process. The logic is defined inside each *AlgthmNode* subclass and it references the execution of a child algorithm, independently of the routine that is in that child.
4) Each *AlgthmNode* is related to a set of *AlgthmData*, which may be shared with other nodes.

5) The driver of the whole process is identified.

9. Variants

(i) Use of the *TypeObject* pattern [9] to enhance adaptability, producing independence between the software routines and its data components. This is important in situations where the same software component is to be used in different situations and with different pieces of data. The class diagram is like the one in Figure 6. With this extension, AlgthmType provides the AlgthmNode the needed functionality independently of AlgthmData. The relationship between the *AlgthmData* and *DataType* can be done at run time. This extension does not affect the already described interactions of *AlgthmNode* and *AlgthmData* with the other participants.

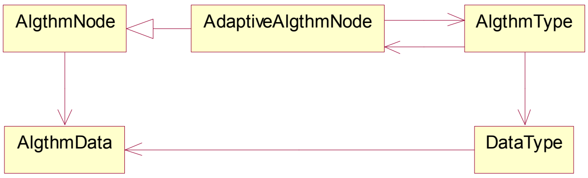

Figure 6 Class diagram for a variant of AlgthmNode

(ii) Hierarchical levels of procedures can be defined, helping in the software management. An application of this extension can be seen in section 11, where three levels of *SkeletonGraphs* were defined. For each one of those levels one may define specific functionalities for all their respective *AlgthmNodes* and *AlgthmData*. Also, at the level of the functionalities of the *SkeletonGraphs* objects specific tools can be defined. These extensions can be oriented for the applications being considered.

(iii) It can be extended to deals with the definition/execution of processes running in a distributed environment. We will not go into further details because this is still under development.

10. First Example of Usage

To exemplify and make clear the use of the GIG-pattern, a very simple application was designed. This application involves the generation of random sequence of items, which are further sorted. A better understanding of the GIG applicability and power, however, can be seen in sections 4 and 11. The proposed application can be subdivided into the following sub-processes (algorithm sub-routines): generation of a random sequence of items; sort of these items; and display of the sorting items. The sort sub-process is implemented here using the Heapsort algorithm derived from [11]. Any one of the sub-processes can be modified afterwards to another one, generating a different solution algorithm.

The Heapsort algorithm written in natural algorithmic language is described in Figure 7. This algorithm is an example solution to a very well known problem, which has many solutions.

In this example the decomposition of the domain data is very simple. It generates only the *HeapSortData* data type, which is used by all Heapsort sub-routines. The created *AlgthNode* classes are derived from the process organization in Figure 8.

Each *AlgthmNode* that is an algorithm subroutine places calls to its children nodes, which implement processes inside the whole process. The logic is defined inside each *AlgthmNode* subclass and it references the execution of a child algorithm, independently of the routine that is in that child.

```
HeapSort (A)

Buil_Max_Heap (A)
for i ← length[A] down to 2 do
Exchange A[1] ←→ A[i]
heap-size[A] ← heap-size[A] –1
Max_Heapify

Build-Max-Heap(A)
Heap-size[A] ← length[A]
For I ← length[A]/2 downto 1 do
Max-Heapy(A,i)

Max-Heapify(A,i)
l ← LEFT(i)
r ← RIGHT(i)
if l <= heap-size[A] and A[l] > A[I]
   then largest ← l
   else largest ← i
if r <= heap-size[A] and A[r] > A[largest]
   then largest ← r
if largest != I
   then exchange A[I] ←→ A[largest]
        Max-Heapify(A,largest)
```

Figure 7 Heapsort Algorithm

The algorithms organization in the form of a direct acyclic graph can be seen in Figure 8.

```
Main Program

I. Generation of aleatory data
II. HeapSort
 II.I Build-Max-Heapify
 II.I.I Max-Heapify
III.Print
```

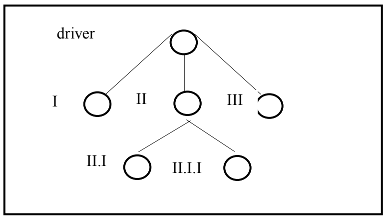

Figure 8 . Main program and correspondent GIG direct acyclic graph

Figure 6 shows the UML class diagram, which was created to implement the application classes in C++ [12], which applies the GIG-pattern. Some classes, described below, were created to implement the GIG pattern and solve the sorting example:

- The *Factory* class follows the GIG implementations suggestions, described in section 9. They define a common interface to materialise *AlgthmData* and *AlgthmNode* objects from a data source. The *FactoryHeapSort* class was created to materialise objects from the classes created to represent the Heapsort algorithm in GIG.
- The classes created to represent the *AlgthmNodes* are: *GenerateRandomAlgthmNode*, *HeapSortAlgthmNode*, *BuildMaxHeapAlgthmNode*, *MaxHeapifyAlgthmNode*, and *PrintAlgthmNode*, each one being related to a procedure described in the HeapSort algorithm, see Figure 8.
- The *HeapSortData* is an *AlgthmData* subclass created to store the sequence of items to be sorted and some control variables.
- The root (driver) of the whole process is here the *GenerateRandomData*..

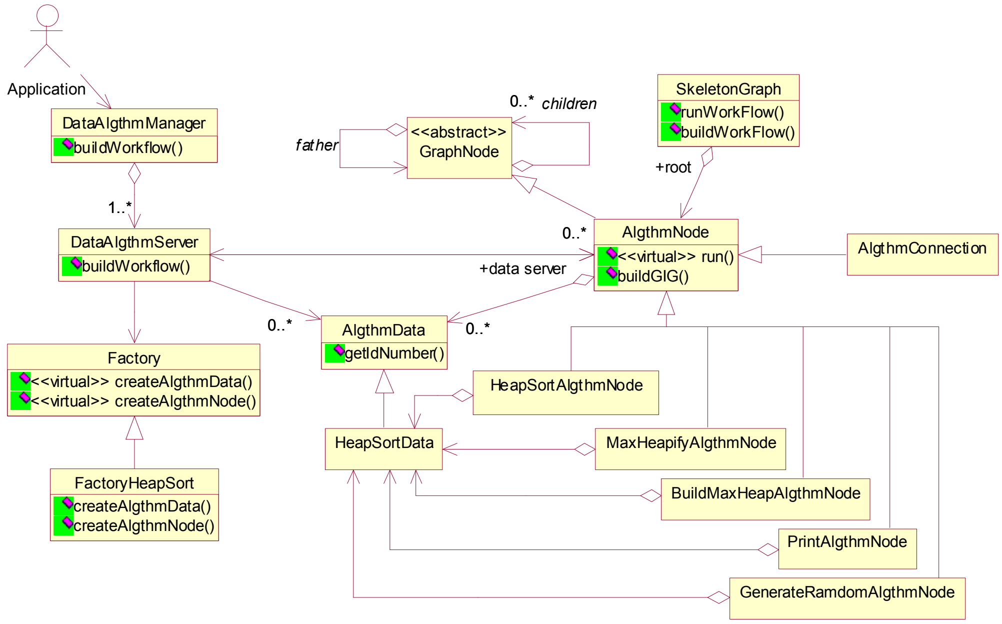

Figure 9 The Class diagram of the GIG implementation and example classes

The sample code presented in Figure 11 is an extract from the implementation of the class *HeapSortAlgthmNode*. With this example, we can understand how the implementation of the *run* method of an *AlgthmNode* class calls the children nodes in its code. In this example the *HeapSortAlgthmNode* class order its first child to run, before some other tasks are performed.

```
void HeapSortAlgthmNode::run()
{
    this->runChild(0); //run Build-Max-Heap
    for( int i = heap->size(); i >= 2; i--)
    {
        heap->swap(1,i);
        heap->decrementHeapSize();
        this->runChild(1);//run Max-Heapify
    }
}
```

Figure 10 Implementation of the run method for the class HeapSortAlgthmNode

11. Second Example

In [6], we present an application of GIG in FEM simulators. As usual it is observed that an algorithm defined for the solution of a problem by the FEM has repeated (similar) hierarchical structures. Thus in the pursuing of a high degree of reusability, therefore a framework considering hierarchical levels of processes were used, where each level may have several possibilities of algorithms, and can be easily described by a GIG graph. The whole hierarchy is represented making the connections between the different levels and generating a complete graph. Global Skeleton, the Block Skeletons, the Group Skeletons, and the Phenomena procedures define those levels [2]. These levels satisfy a number of requirements, such as: (i) to separate less reusable modules from reusable ones; (ii) to make it more comprehensible the decomposition of the simulation data among the several processes; (iii) to make it possible the dynamic re-configuration of the simulator through the replacement of reusable modules.

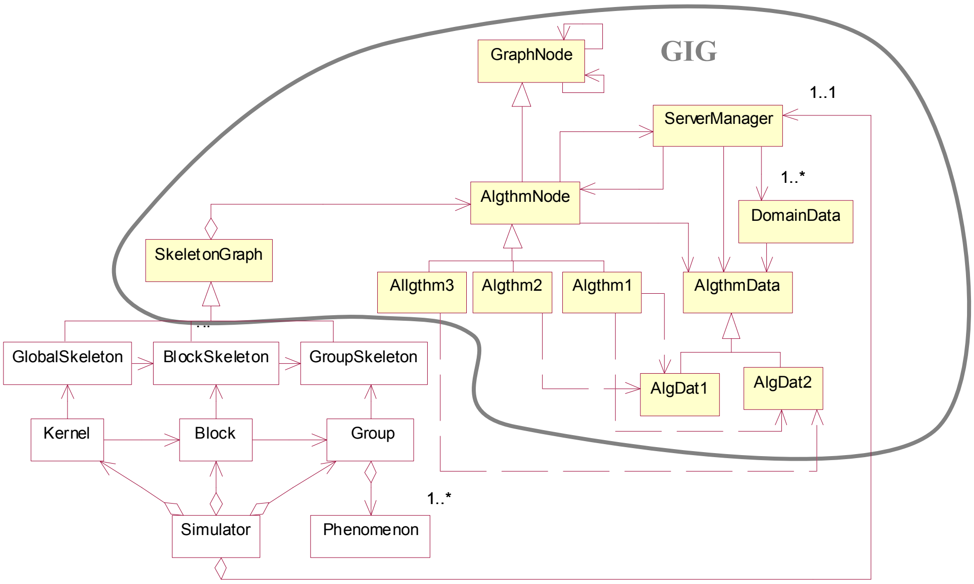

Figure 11 FEM Simulator and GIG classes

For instance, the global Skeleton articulates time loop (if present), adaptation iterations and defines processes involving the call of Block Skeletons. Block Skeletons may define different solution strategies for different Groups, thus, articulating Group processes. Group Skeletons articulate their phenomena procedures in very specific less reusable ways. It is in this level that solvers for algebraic systems are applied. Phenomena are the abstraction of the entities being simulated. All those skeletons can be implemented as objects from classes following the GIG pattern (see Figure 11). Therefore, the GIG would allow for the realization of the interoperability of the different levels of computation (by automatically plugging the lower level skeletons in the higher ones).

In the example described below, we consider a FEM simulator specification. This kind of simulator is capable of solving, for example, problems involving transient phenomena, where the phenomena context includes linear temperature-dependent elasticity, rigid body motion and linear heat transfer [2,6]. Only two blocks are needed in the present case. The number of Groups depends on the phenomena types present in a specific simulation. The number and type of phenomena depends as well on the simulation being carried out. In the ith-Block Skeleton Nig is its number of groups.

```
I.From Blocks i = 1 until 2
  I.0)Retrieve initial state for Block i
  I.I)Compute initial time step $\Delta t_i$ for Block i
  I.II)Compute initial auxiliary data for Block i
II.Compute initial $\Delta t = \min_{1 \le i \le 2}\{\Delta t_i\}$,set time instant $t_1$=0
III.While $t_1 \le T_{max}$ do:
  III.0)Set $t_0 = t_1$ and $t_1 = t_0 + \Delta t$
  III.I)For Block i = 1 until 2
    III.I.0)Solve for Block i
    III.I.I)Compute next time $\Delta t_i$ for Block i
  III.II)Compute next time step $\Delta t = \min_{1 \le i \le n} \{\Delta t_i\}$
  III.III)Continue with time iteration
IV.End of the simulation
```

Figure 12 Global Algorim Skeleton

Figure 12 shows the Global Skeleton, while Figure 13 shows two Block Skeletons. Figure 14 and Figure 15 present the GIG direct acyclic graph to implement Global and Block Algorithm skeletons.

```
Is-B$^i$)Retrieve Initial State for Block **i** (see(I.0)):
    Is-B$^i$.0)For **r = 1** until Nig
      Is-B$^i$.0.0)Group **r**, compute phenomena initial states
It-B$^i$)Compute initial time step for Block **i** (see(I.I)):
    It-B$^i$.0)For **r = 1** until Nig
      It-B$^i$.0.0)Group **r**, compute Initial time step $\Delta r$
    It-B$^i$.I)Set $\Delta t^i$ = min $_{1 \le r \le Nig}$ $\{\Delta_r\}$
```

Figure 13 Block Algorithm Skeletons

As it was already said, there should be *AlgthmData* objects, which will contain the needed problem and process data needed by each *AlgthmNode* object. A specialization of *AlgthmNode* is *AlgthmConnection*, which is defined whenever a lower level process is to be called up. Its *AlgthmData* object includes pieces of information needed in the identification of the lower level skeleton that will be plugged in the Algorithm Skeleton Graph. This identification concerns a driver *AlgthmNode* object (from another graph, integrating in this way the graphs presented in Figure 14 and Figure 15), which will substitute the related *AlgthmConnection* object.

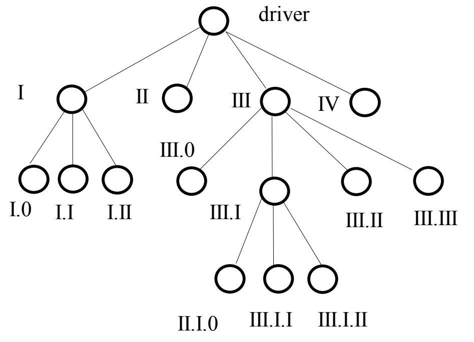

Figure 14. Global Alg.Skeleton graph

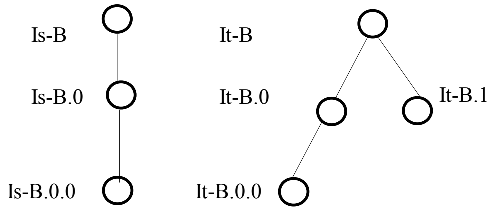

Figure.15 Block Alg.Skeletons graphs

12. Consequences

In what follows we make some considerations about the forces treated by the proposed pattern.

We can observe **positive forces** for the use of the GIG-pattern:

- Easiness of translating from algorithmic language into computer processes and simplicity in the process definition. It supports an organisation in a graph level, providing the distribution of code in a very flexible way, not compelling a rigid division of code. To improve simplicity in process definition we try to avoid unnecessary levels of details and to maintain similarity to the predefined algorithmic structure.
- Different users have evaluated this pattern with success, in applications with different levels of complexity. A simple example can be seen in section 10, and a more complex one in [6].

- Support for different levels of granularity of the defined processes. It allows a flexible representation for a mixture of scales, since it does not restrict the levels of programming into which the code is defined. Differently, in [1] the workflow must be defined in terms of a set of node types that are already been coded in the programming language level.
- It can be applied to any domain solution, through the definition of specific domain data classes and algorithms, as it can be seen from the pattern participants, in section 6.1,
- It allows the test of individual parts of the process independently, reducing the error occurrences in the coupling of processes.
- It allows the reuse of entire solutions, making changes in specific points. In GIG, it is easy to change parts of the graph, maintaining the other ones intact.
- It allows the graph change (that is, the process change) at run-time. This is achieved through the GIG intrinsic dynamic structure, as was shown in section 7.2. Data and process can be defined at run-time, depending on GIG implementation, once the pattern can be easily extended to incorporate design patterns like [9], as presented in section 9.

Some **negative forces**, or restrictions, can also be identified:

- The pattern makes severe restrictions on the graph structure, requiring it to be an acyclic graph. The designer is not allowed to define neither recursive iterations nor loops out of the node code.
- GIG-pattern makes no explicit reference or imposition for the use of a specific set of process types, differently from [1]. We can consider that this may cause loss of workflow-refined control. It is the programmer responsibility to define and manage this organization, if required by the application.
- Flow control is inside each node code. This can bring difficulties to some part of the process adaptation and control.
- Synchronization is not GIG-pattern responsibility. GIG does not define a specific structure to deal with process parallelism and processes synchronization. To allow the definition of processes parallelism, the programmer has to deal with extra complexity. GIG-pattern requires unnecessary levels of repetition, that is, the replication of whole process graph branches.

We may summarize saying that this pattern it is **not so appropriate** for applications that are simple and do not require exchangeable processes, modularity or articulation of sub-routines. Also it is **inappropriate** for applications where there is a need of a high level of refinement in the programs code, or if they need to process synchronization and parallelism; in these cases, an alternative is the use of the Micro-workflow proposal, described in [1]. However, through the use of the Micro-workflow alternative one of the worthy things you loss is simplicity and the level of granularity; the translation from algorithmic language is not such a direct mapping, losing in this way some levels of abstraction. The application of the GIG pattern to simplify application can be more expensive then a simple solution. On the other hand, it provides extra facilities like reuse, flexibility for new solutions, domain independence, etc.

13. Related patterns

The following patterns, can be used together with the GIG-pattern:

- *Factory Method* [10], which can be used to materialize objects for workflow management;
- *Template Method* [10], used to define skeletons of algorithms in *DataAlgthmServer* class;
- *Composite* [10], used to implement the *AlgthmNode* class functionality in the framework.
- *Proxy* [10], used in *AlgthmConnection* class;
- *Strategy* [10], *used in AlgorithmNode* and *AlgorithmData* classes
- Adaptive object-model patterns, such as *TypeObject* [9], shown in variants section.
- *FEM-SimulatorSkeleton* [2] achieves great benefit from the GIG approach.

14. Known uses

Many variations of numerical algorithms show the very same organizing structure, which was abstracted by the GIG-pattern. Some of these numerical algorithms are: mesh generation procedures, geometric reconstruction from planar slices and integration of geometric reconstruction procedures, and so on. Despite of being a generic solution that can be applied elsewhere, the users of this pattern have been scientists and engineers. The GIG-pattern has been applied with success in the development of different FEM simulator applications, and in a variety of other numerical methods in computational Mechanics. Other known uses, which applies GIG pattern, is a specific environment called Plexus, whose objective is the construction of FEM simulators for treating problems involving coupled multi-physic phenomena [2,6]. This environment applies GIG as a general solution for the numerical methods and articulation strategies for solving groups of phenomena [15]. Section 11 has given some details.

Acknowledgments

Thanks to Joe Yoder who served as the SugarLoaf PLoP shepherd.

References

1. Manolescu D.A., "Micro-Workflow: A Workflow Architecture Supporting Compositional Object Oriented Software Development", Ph.D, Depart. of Computer Science University of Illinois at Urbana-Champaign, 2001.
2. Lencastre M., Santos F., Rodrigues I., "FEM Simulator based on Skeletons for Coupled Phenomena", SugarloafPLoP'2002, Brazil.
3. Lencastre M., Santos F., "FEM Simulation Environment for Coupled Multi-physics Phenomena". Simulation and Planning In High Autonomy Systems, AIS02, Portugal, 2002.
4. Lencastre M., Santos F., Araújo J, "A Process Model for FEM Simulation Support Development" SCSC2002', Summer Computer Simulation Conference, California, 2002
5. Lencastre, M., Santos F., Vieira M. "Workflow for Simulators based on Finite Element Method", Internat.Conference on Computational Science 2003 (ICCS 2003), Saint Petersburg, Russian, 2003.
6. Lencastre M., Santos F., Vieira M., "A Case Study using GIG", submitted 2003.
7. Workflow Management Coalition, "The Workflow Reference Model, Workflow Management Coalition Specification", - Winchester, Hampshire - UK, 95.
8. Casati F., Fugini M.G, Mirbel, I. and Pernici, B., "WIRES: A methodology for developing Workflow Applications", Requirements Engineering (2002), volume 7, number 2, Editors P. Loucopoulos and J. Mylopoulos ISSN:0973602.
9. Yoder J., Johnson R. "The Adaptive Object Model Architectural Style", Proceeding of The Working IEEE/IFIP Conference on Software Architecture (WICSA3'02), World Computer Congress in Montreal, 2002.
10. Gamma E., Helm R., Johnson R., Vlissides J., "Design Patterns: Elements of Reusable Object-Oriented Software" Addison-Wesley, 1995.
11. Cormen T., Leiserson C., Rivest R., Stein C., "Introduction to Algorithms" second edition, MIT Press, 2001.
12. Stroustrup, B. "The C++ Programming Language", third edition, Addison-Wesley, 1997.
13. Fayad M., Douglas S. Johnson R., "Building Application Frameworks: Object-Oriented Foundations of Framework Design", Wiley Computer Publishing, 1999.
14 "Software Architecture System Design, Development and Maintenance" Edited by Jan Bosch, Morven Gentleman, Christine Hofmeister, and Juha Kuusela; Kluwer Academic Publishers 2002.

15. Lencastre M, "PLEXUS - A domain specific approach for FEM simulators development", a PhD thesis being developed in Federal University of Pernambuco, Brazil, 2003.