Functionality and Partitioning Configuration: Design Patterns and Framework

Francisco Assis Rosa and António Rito Silva
INESC/IST Technical University of Lisbon Rua Alves Redol n°9, 1000 Lisboa, PORTUGAL
Tel: +351-1-3100287, Fax: +351-1-3145843
Email: fjar@scallabis.inesc.pt, Rito.Silva@acm.org

Abstract

Configuration is often strongly coupled with distribution. In most of the approaches the configurable components are intrinsically deployed on distributed nodes. However, this does not need to be so, functionality configuration can be decoupled from partitioning configuration. In this paper two design patterns which support such decoupling are presented as well as their composition. An object-oriented framework which implements the patterns is also described. The proposed solution simplifies a software development process where traceability, incremental development and application maintenance are key issues. In particular the design patterns provide static and dynamic configuration of component, structure and geometry.

1 Introduction

Usually, configuration is strongly coupled with distribution. Configuration considers two aspects: functionality configuration, definition of components and their, possible dynamic, connections; and partitioning configuration, definition of distributed nodes and the components deployment, possible dynamic, on top of them. In most of the approaches the configurable components are intrinsically deployed on distributed nodes. For instance Darwin [1] couples component definition with component deployment location definition.

However, this does not need to be so. From a developer perspective, the definition of components and their connections can be done independently of the components deployment on distributed nodes. Moreover, it can be independent of the distributed technology to use. Functionality configuration fulfills the application functional requirements by defining the services provided and required by components and their possible dynamic connections. Partitioning configuration fulfills the application non-functional requirements by defining application partitioning and components deployment and redeployment on the distributed nodes. Functionality configuration is orthogonal to partitioning configuration since the developer can verify the satisfaction of functional requirements without considering partitioning issues. So, a separate fulfillment of these two kind of requirements is desirable.

There are three main advantages of separating functionality configuration from partitioning configuration:

- **Traceability**. The separated fulfillment of functional from non-functional requirements allows a traceable development process where it is easier to map design decisions to application requirements.
- **Incremental development**. An incremental development is achieved where the application is firstly enriched with functionality configuration and with partitioning configuration afterwards. This approach allows incremental application requirements fulfillment, application testing and application debugging.
- **Application maintenance**. Since the application design will keep the separation, the maintenance tasks are simplified because the introduction of new functionality configuration requirements has a controlled impact on partitioning and the introduction of new partitioning requirements has no impact on functionality configuration.

Moreover, the advantages usually pointed to configuration are reinforced by this separation:

- *Software prototypes development* can benefit from the separation because different functional and distributed architectures can be tested separately.
- *Flexible systems development* can benefit from the separation because the functional requirements for flexibility can be fulfilled independently of partitioning.
- *Fault tolerance support* can benefit from the separation because the algorithms of reconfiguration after a node fault can be designed independently of the node components.

The rest of this paper is structured as follows. Next section presents two design patterns for functionality and partitioning configuration as well as their composition. Section 3 describes a three-layered framework which implements the design patterns and their composition. Related work is presented and discussed in section 4 and section 5 presents conclusions.

2 Design Patterns

Design solutions for functionality and partitioning configuration are presented as design patterns [2]. Design patterns describe the structure and behavior of a set of collaborating objects. They have become a popular format for describing design at a higher level of abstraction than code.

Contrarily to the common description of design patterns which emphasizes the pattern known uses, the description here presented emphasizes the quality aspects associated with the solution. To do so, a slightly different format is used: intent; problem; problem analysis; forces; structure; participants; collaborations; and forces resolution. Problem analysis describes concepts and solution approaches related to the problem. Forces define the qualities that should drive the design solution while forces resolution describes how the final solution fulfills the desired qualities.

2.1 Functionality Configurer Pattern

2.1.1 Intent

The Functionality Configurer Pattern separates the configuration of a component-based application functionality from the functionality implementation of its components.

2.1.2 Problem

Component-based applications should be able to use both primitive components and composite components. Primitive components should be able to configure its interfaces while composite components should also be additionally able to configure its inner structure of components.

2.1.3 Problem Analysis

Configuration in distributed systems usually involves the promotion of separation between small and large grained programming. While small grained programming deals with the details of component functionality programming, large grained programming deals with the large view of applications as sets of elements, usually components, interacting with each other in order to fulfill application functionality.

Some concepts are usually present when speaking of configuration. *Component* is the basic element of configuration. These elements can have a mapping to a piece of software developed in a programming language. In [3] components are defined as entities with persistent identity and interfaces whose observable behavior is governed by a state. Components are almost never self-contained and usually interacting with other components. *Component Connection* supports cooperation between components. Only components having connections between them should be able to cooperate. *Component Structure* is a description of components and its cooperation structure using connections. The structure description will make the large grained programming.

Solutions to this problem usually involve several variations:

- **Components Structuring** - components can be structured in either a flat structure or a hierarchical structure. A hierarchical structure simplifies configuration.
- **Component Encapsulation** - hierarchical components may encapsulate their sub-components thus hiding them from exterior. Encapsulation introduces a new level of abstraction at the price of performance.
- **Dynamic Configuration** - a basic component can change its implementation and connections between components can also be changed. This allows the possibility of dynamic configuration at run-time as opposed to static (creation-time) configuration.
- **Typed Connection** - connection between components can have restrictions on typing. Type checking enforces applications correctness.

2.1.4 Forces

An object oriented solution for the functionality configuration problem should address the following forces:

- **Encapsulation** - component configuration issues should be placed near the components instead of spread throughout the application.
- **Extensibility** - the solution should be extendible to the variations described in the problem analysis.
- **Modularity** - functionality configuration should be separated from basic components functionality. Moreover, other topics such as distribution configuration or distributed communication should also be orthogonal.
- **Reusability** - the solution for functionality configuration should be reusable in different situations. In particular, reuse of components implementation should be independent of configuration reuse.

2.1.5 Structure and Participants

The pattern functionality configuration will be presented in two steps: primitive component structure and composite component structure.

Primitive Component Structure The Booch[4] class diagram in Figure 1 illustrates the primitive component structure.

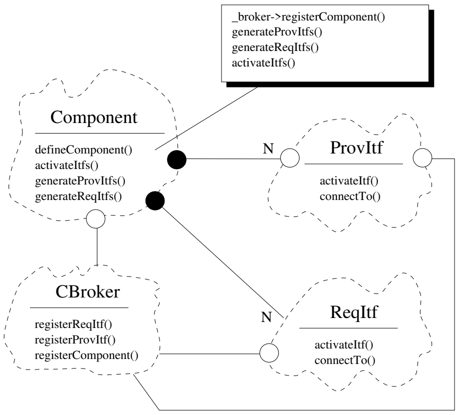

Figure 1: Component Class Structure

The main participants in the primitive component structure are:

- `Component` - contains the implementation of a component. It also contains a set of required interfaces (`ReqItf`) and a set of provided interfaces (`ProItf`) having the responsibility of its manipulation. Defines methods for component definition. Method `defineComponent` deals with the definition of the component, i.e. registration of the component with the `CBroker`, generation and activation of the component's interfaces. Methods `generateProvItfs`, `generateReqItfs` and `activateItfs` are responsible for the component's interfaces generation and registration with `CBroker`.

- `ReqItf` - represents an external request to another component. It has a unique identification. Defines two methods: `activateItf` and `connectTo`. The former is responsible for registering itself in it's `CBroker` while the latter is responsible for the establishment of the connection to other interfaces.

- `ProvItf` - represents one possible entry point to a component. It has a unique identification. Requests to a component will be accomplished through the use of these elements. Each specific `ProvItf` will have a subset of the interface of its component. Like `ReqItf`, it defines two methods `activateItf` and `connectTo` with similar responsibilities as those of `ReqItf`.

- `CBroker` - associates component and interfaces identifications with their locations. It is responsible for component and interface registrations manipulation. A `CBroker` provides visibility to a component in terms of other interfaces and components.

Composite Component Structure The Booch[4] class diagram in Figure 2 illustrates the composite component structure.

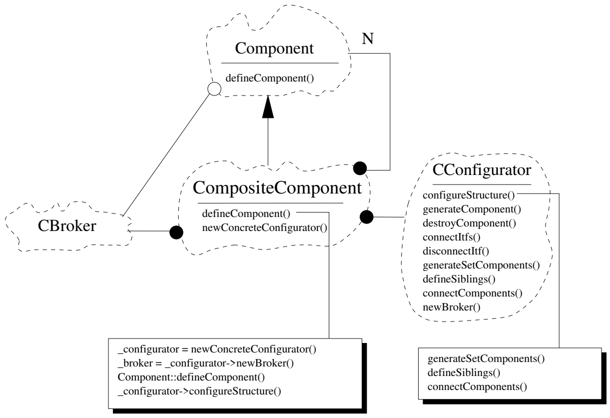

Figure 2: Composite Component Class Structure

Besides `Component` and `CBroker` previously presented, the main participants in the composite component structure are:

- `CompositeComponent` - offers the necessary abstractions for component encapsulation. It manages the list of its sub-components. Inherits from `Component` its component definition behavior. Interacts with `CConfigurator` to perform its configuration.

- `CConfigurator` - it is responsible for composite component configuration. Method `configureStructure` is responsible for the sub-component creation and connection establishment. It uses methods `generateSetComponents` which is responsible for the generation and activation of all the composite component's sub-components and `connectComponents` which establishes all the connections between sub-components using method `connectItfs`.

2.1.6 Collaborations

The main collaboration present in the pattern is the composite component configuration collaboration.

The composite component collaboration involves the cooperation of classes `CompositeComponent`, `CConfigurator` and `CBroker`.

The collaboration is started by a call to the `defineComponent` method in

CompositeComponent. As first step in this collaboration, CompositeComponent uses its newConcreteConfigurator method thus acquiring a new CConfigurator. Using this new CConfigurator it calls method newBroker to obtain its new CBroker. Then it defines its interfaces by calling superclass method Component::defineComponent and defines its inner structure by calling method configureStructure in CConfigurator. Method configureStructure calls methods generateSetComponents which will allocate new Component objects and associate them with CompositeComponent, and method connectComponents which will in turn establish connections between component's interfaces using method connectItfs.

This collaboration corresponds to static CompositeComponent configuration. Dynamic configuration can be achieved by using the services provided by connectItfs, disconnectItfs, generateComponent and destroyComponent.

2.1.7 Forces Resolution

The presented solution addresses the following forces:

- **Encapsulation** - encapsulation is achieved by placing the configuration code near components. Components offer a facade-like interface, defineComponent, encapsulating the actual steps of its configuration. This configuration code does not belong in any client's code.

- **Extensibility** - the presented pattern abstracts the variations presented in the problem analysis item. Specializations of the pattern classes allow extensibility.

  The use of CompositeComponent allows the generation of *hierarchical structuring*. However, if a single CompositeComponent is used then a flat structure is generated.

  CBroker introduces component visibility. If the same instance is associated with several CompositeComponent objects then there is no *encapsulation* between them. Objects in one of these CompositeComponent will be visible from the others.

  CConfigurator allows *dynamic configuration* of component and structure. Components can be created and destroyed using methods generateComponent and destroyComponent. Connections can be established and broken using methods connectItfs and disconnectItfs.

  Typed connection can be enforced by dynamic checkings done at CBroker, ProvItf and ReqItf using type information. However, static typed connections is not possible.

- **Modularity** - decoupling of component functionality is achieved. Component encapsulates functionality. Moreover, the current solution does not need to consider partitioning configuration or distributed communication.

- **Reusability** - due to modularity, component functionality can be reused independently of functionality configuration. The same component functionality can be used in different functional configurations.

2.2 Partitioning Configurer Pattern

2.2.1 Intent

The Partitioning Configurer pattern presents an abstraction to the configuration of applications partitioned into nodes. It decouples this partitioning configuration from the functionality of the elements belonging to the application.

2.2.2 Problem

The development of a partitioned application involves deciding where and how are the elements of the application going to be deployed.

2.2.3 Problem Analysis

A distributed computing system is defined in [5] as a system of multiple autonomous (logical) processors that cooperate only by sending messages over a communication channel. A node is one of these autonomous processors.

In order to cooperate nodes should be configured and node configuration can be static or dynamic. *Static configuration* corresponds to the deployment of functional entities on nodes at node creation time. *Dynamic configuration* should allow the migration of these functional entities within nodes during execution.

2.2.4 Forces

An object oriented solution for the partitioning configuration problem should address the following forces:

- **Encapsulation** - distribution configuration issues should be well localized and not scattered throughout the application.

- **Extensibility** - the solution should be extendible to support static and dynamic configuration.

- **Modularity** - partitioning configuration should be independent from application functionality. In particular it should be independent of functionality configuration. Moreover, it should be orthogonal to distributed communication.

- **Reusability** - the solution for partitioning configuration should be reusable in different situations. In particular, it should be independent of functionality configuration reuse.

2.2.5 Structure and Participants

The Booch[4] class diagram in Figure 3 illustrates the application partitioning structure.

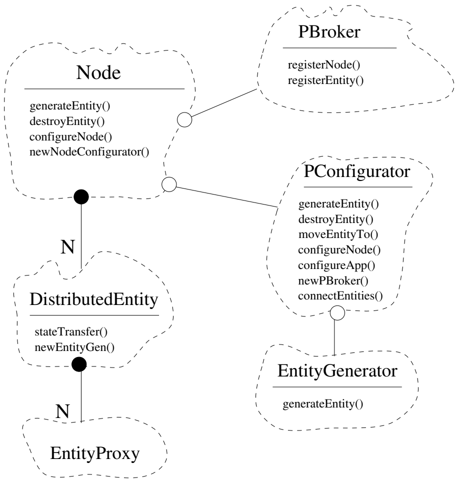

Figure 3: Partitioning Class Structure

The main participants in the partitioning configurer pattern are:

- `Node` - represents a unit of partitioning. It contains several `DistributedEntity` objects and defines methods for their manipulation. It is in charge of creating and destroying `DistributedEntity` objects.

- `DistributedEntity` - represents a functionality unit. A partitioned application is made up of a set of these distributed entities scattered among the several existing nodes. `DistributedEntity` defines method `stateTransfer` for migration of distributed entities. The association with `EntityProxy` represents a set of remote `DistributedEntity` with which it cooperates.

- `EntityProxy` - represents a level of indirection for cooperation between elements of `DistributedEntity`. A `DistributedEntity` can only access another one through a `EntityProxy`.

- `PBroker` - responsible for `Node` and `DistributedEntity` registration. Through this class, distributed entities can locate each other in order to cooperate.

- `PConfigurator` - responsible for the application partitioning configuration. It is in charge of the deployment of distributed entities on their correct nodes. Defines methods `configureNode` and `configureApp` for node and functionality configuration. While the former is responsible for the basic node configuration, i.e. broker assignment and node registration, the latter is responsible for the deployment of the node's distributed entities, and the establishment of connections between them.

- `EntityGenerator` - responsible for the generation of objects of class `DistributedEntity`. Used by `PConfigurator` for distributed entities generation in the migration of entities between nodes.

2.2.6 Collaborations

Two collaborations can be addressed in the partitioning pattern: node configuration collaboration and entity migration collaboration.

Node Configuration Collaboration The node configuration collaboration is activated when a node is initialized by a call to method `configureNode` in `Node`.

As a first step, `Node` calls `newNodeConfigurator` method obtaining from it one `PConfigurator` object. Afterwards it calls the `configureNode` and `configureApp` methods in `PConfigurator`.

Method `configureNode` starts obtaining and delivering a `PBroker` object to `Node` by calling method `newPBroker`. Finally it registers the node using method `registerNode` of `PBroker`.

Method `configureApp` configures the node by creating all the required distributed entities in `Node`. Distributed entities are created by calling method `generateEntity` of `Node`. Afterwards, having created all the distributed entities, method `configureApp` establishes connections between them by calling method `connectEntities`. This method uses an `EntityProxy` object to establish the connection between each pair of `DistributionEntity`.

Entity Migration Collaboration The entity migration collaboration is activated by a call to method `moveEntityTo` in `PConfigurator`.

As a first step, `PConfigurator` obtains the entity and the source and target nodes of migration from the element `PBroker`. From the entity to be migrated it obtains an `EntityGenerator` object by calling method `newEntityGen`. This `EntityGenerator` will be capable of generating distributed entities with the same class of the migrating entity.

Next, the old entity is unregistered from `PBroker` and the new entity is generated in the target node by calling method `generateEntity` using the new `EntityGenerator` as parameter.

Using the reference to the newly generated entity, entity state is transfered from the old to the new entity by calling method `stateTransfer` on the old entity and using the new entity reference as parameter. `stateTransfer` includes transference of distributed entity proxies.

As a final step, the old entity is destroyed from the source node by calling method `destroyEntity`.

2.2.7 Forces Resolution

The presented solution addresses the following forces:

- **Encapsulation** - encapsulation is achieved because partitioning configuration is located in `PConfigurator`. Partitioning configuration is hidden from the client's point of view.

- **Extensibility** - for *static configuration* purposes only `PBroker` can be discarded and hard-coded into the `configureApp` method in `PConfigurator`. However, this solution excludes *dynamic configuration* because object location are defined at creation time. `EntityGenerator` objects, `EntityProxy` objects and method `stateTransfer` can also be discarded if *dynamic configuration* is not required. The proposed pattern allows both types of configuration.

- **Modularity** - decoupling of functionality configuration is partially achieved. Method `configureApp` in `PConfigurator` requires to know about functionality configuration. Method `stateTransfer` also requires to know about `DistributedEntity` functionality. Note that the current solution does not need to consider distributed communication.

- **Reusability** - as in the previous pattern, this results from modularity.

2.3 Patterns Composition

The composition of functionality configuration with partitioning configuration results on a composite design that describes configurable and distributed components. To describe the composition a variation of the approach proposed in [6] is used. Riehle describes patterns as participant diagrams[1] and composite patterns as participant diagrams composition enriched with composition constraints between participants.

As in [6] the composition is herein described by participant and collaboration diagrams composition. However, to represent the composition of pattern variations, the composition constraints are initially applied to the pattern variations. The composition constraints describe the consistent combinations.

2.3.1 Participants Composition

The participants of the composite pattern result from the composition of participants in Figures 1, 2 and 3. While some of the participants are kept unchanged other need to be composed. Composed participants combine the structure and responsibilities of their components. Note that this level of description abstracts from the specific composition techniques as inheritance or delegation. The use of composition techniques are subject of particular pattern implementations. See Section 3 for a C++ framework implementation of the patterns.

The composite pattern has the following composed participants:

- `DistributedComponent` - results of `Component` and `DistributedEntity` composition. It combines migration responsibility with component definition. Its has a list of `PReqItf` objects and another of `ProvItf` objects.

- `PReqItf` - results of `ReqItf` and `EntityProxy` composition. It has the responsibilities of `ReqItf` since it already includes `EntityProxy` responsibility for a level of indirection between `DistributedEntity` cooperation.

- `CPBroker` - results of `CBroker` and `PBroker` composition. It provides registration and location of `Node`, `DistributedComponent`, `PReqItf` and `ProvItf`.

- `CPConfigurator` - results of `CConfigurator` and `PConfigurator` composition. It combines functionality configuration responsibility with partitioning configuration responsibility.

2.3.2 Collaboration Composition

Functionality configuration collaborations are composed with partitioning configuration collaborations. The main composition occurs during node configuration collaboration. In this case method `ConfigureApp` of former `PConfigurator` should use the functionality configuration in former `CConfigurator`.

2.3.3 Composition Constraints

The composition has the following constraints:

1. Partitioning is always performed from a top composite component. Each top component sub-component, from now on called node component, is deployed

[1] In his approach participants are described as roles extending Reenskaug's[7] work on role modeling.

onto a node. If node components are also composite components their sub-components can not be deployed on different nodes. PConfigurator uses CConfigurator and not vice versa.

2. Partitioning is flat, hierarchical partitioning is not allowed. PConfigurator can not be composed with CConfigurators of node components, it is only composed with the top CConfigurator. There is a single CPConfigurator and several possible CConfigurators.

3. Dynamic partitioning configuration requires dynamic component functionality configuration. Migration of components within nodes implies generation and destruction of components. And therefore change of component implementation.

4. Dynamic/Static structure functionality configuration is not dependent on which variation of partitioning configuration is used, either dynamic or static. Static structure functionality configuration requires the existence of provided and required interfaces which provides the indirection necessary for dynamic partitioning configuration.

5. If a node component supports visibility of its internal sub-components then migration of its sub-components is forbidden. This constraint is a consequence of the first constraint.

3 Framework

In the previous section pure pattern descriptions, independent of implementation details, were presented. In this section we present a C++ object-oriented framework which implements these patterns and their composition. For this implementation a three-layered framework [8] is used. The three layers are: pattern, composition and application. The Pattern layer contains the modules which implement the design patterns. The Composition layer contains the composition modules which result from the combination of pattern modules. At the Application layer composition modules are integrated within the application and pattern modules are specialized according to the application-specific requirements.

Objects in pattern modules can be classified according to their properties: *Stable Objects* represent the objects which are encapsulated within pattern modules and do not need to be specialized. *Variable Objects* represent the objects which need to be specialized for integration or customization. Integration objects are used for integration at composition and application layers. Customizable objects are used for customization at the application layer.

Composition module is responsible for integrating pattern modules and to enforce composition constraints. The composition modules should also offer a simple interface for final integration in the application and hide the particularities of inter-pattern dependencies from programmers at the application layer.

In the application layer, composition modules are integrated in the final application and pattern variations may be customized.

3.1 Pattern Layer

Figure 4 represents the functionality configuration module.

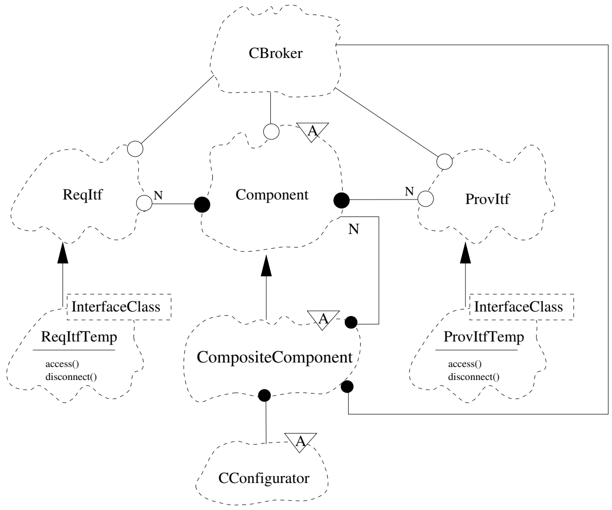

Figure 4: Functionality Configuration Module

Two additional classes, ReqItfTemp and ProvItfTemp, are introduced to simplify customization of component interfaces. At the application layer programmers only need to instantiate the template argument.

The module collaborations are a stable part. Classes ReqItf and ProvItf are also stable, they do not need to be further redefined. Classes CBroker, CConfigurator, Component and ReqItfTemp are integration classes as described below. Classes CConfigurator, Component, CompositeComponent, ReqItfTemp and ProvItfTemp are customization classes as described bellow.

Figure 5 represents the partitioning configuration module.

The module collaborations are a stable part. Classes PBroker, PConfigurator, EntityProxy and DistributedEntity are integration classes as described below. Classes PConfigurator,

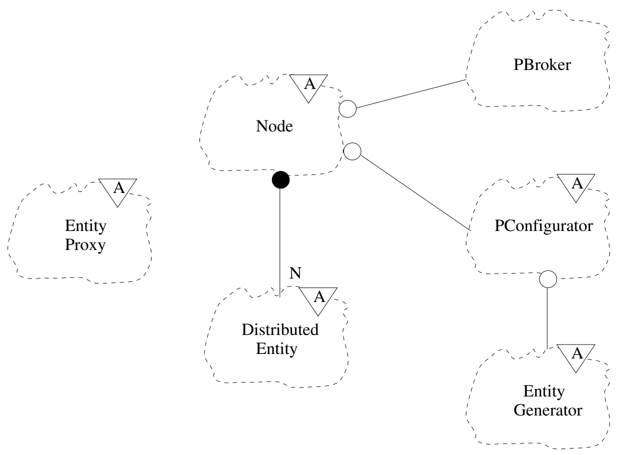

Figure 5: Partitioning Configuration Module

`DistributedEntity`, `Node` and `EntityProxy` are customization classes as described bellow.

3.2 Composition Layer

Multiple inheritance is the main used composition technique for pattern module integration. The new types, resulting from multiple inheritance, allow the composed classes to be manipulated by both pattern modules.

- `DistributedComponent` - multiple inherits from `Component` and `DistributedEntity`.
- `PReqItf` - multiple inherits from `ReqItf` and `EntityProxy`.
- `CPBroker` - multiple inherits from `CBroker` and `PBroker`.
- `CPConfigurator` - multiple inherits from `CConfigurator` and `PConfigurator` interface. Figure 6 represents the composition. `CPConfigurator` uses a `TopCConfigurator` which is the top functionality configurator. These two configurators should cooperate to compose their collaborations for distributed component configuration. Moreover, sub `CompositeComponents` are restricted to use `SubCConfigurator`.

The proposed composition satisfies the composition constraints. (1) partitioning is associated with top component only (`CPConfigurator`); (2) sub composite components are associated with `SubCConfigurator` which does not deal with partitioning; (3) dynamic component functionality configuration requires an abstract component generation object which must be combined with `EntityGenerator`; (4) is trivially enforced; (5) `CPConfigurator` should be implemented in order to enforce this constraint.

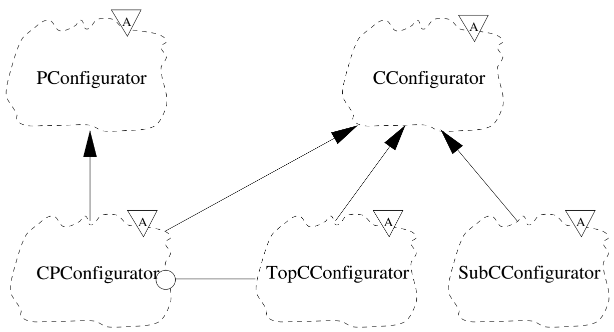

Figure 6: Structural Composition

3.3 Application Layer

At this layer the customization classes are adapted to the application specificity. This adaptation follows from class derivation and template method definition.

Several implementations are admissible for node configurators and brokers. Decision on the implementation to be used is driven by their partitioning and deployment. For instance:

- **Centralized Node Broker**. The broker can be implemented as a single instance shared and accessed by every node. Applications where migration is heavily used can benefit a lot from this implementation because of synchronization problems. However, this implementation penalizes fault tolerance and availability.
- **Replicated Node Broker**. The broker can be implemented as an instance per node. Applications gain on availability and fault tolerance because a node failure does not affect non-dependent nodes. However, this implementation penalizes synchronization and communication being more suitable for static configuration.
- **Centralized Node Configurator**. The configurator can be implemented as a single instance shared and accessed by every node. It shares the same advantages and disadvantages of centralized node broker. Moreover, it promotes easier application maintenance through centralization of configuration instructions.
- **Partitioned Node Configurator**. The configurator can be implemented as an instance per node. It shares the same advantages of replicated node broker. However, it reduces application maintenance.

Incremental development is enforced at this layer. Basic functionality is developed first, then functionality configuration, and afterwards partitioning configuration. In each of the steps customized code can be reused in the next step. In this perspective the centralized node configurator is best suited than the partitioned node configurator implementation.

4 Related Work

Some other works have developed solutions to the configuration problem. Some of these works are presented and analyzed according to its characteristics: component interface definition, component structure definition, decoupling between component functionality and configuration, type of configuration supported according to the classification of configuration as component configuration, logical structure configuration and geometry configuration, given in [9].

Regis/Darwin [1] [10] is a programming environment for the development and execution of distributed programs. The *Regis* approach is based on code skeleton generation out of *Darwin* configuration specifications. The *Darwin* configuration language [11] [12], represents a language support for the configuration problem. Its objective is to describe and define the control of the configuration structure of a component-based application. It presents the following characteristics:

- Allows component interface and structure definition. It allows the definition of hierarchical structures. This definition is all done by the *Darwin* configuration language.
- Promotes the decoupling between functionality and configuration. Component functionality implementation issues are orthogonal to configuration issues.
- Supports static configuration of components, logical structure and geometry. Both, through the *Darwin* language and by *Regis* the run-time support, a weakened form of dynamic configuration of logical structure is supported through the use of dynamic and lazy structures.

Olan [13] [14] incorporates both an architectural description language and a run-time support for component-based distributed applications. It adds to *Regis/Darwin* typification of connectors at the specification level, event handling as part of the interface description, attributes as part of a component's interface and associative naming for identification of components within collections. Configuration of distribution is achieved by the use of interface attributes accessible to the run-time support managers.

Polylith [15] is a software interconnection system. It provides a module interconnection language and a software bus for inter-component cooperation. It presents the following characteristics:

- Allows component interface and structure definition through the use of the *Polylith* module interconnection language. It only supports flat structuring of components.
- Decouples component functionality from component static configuration. Component functionality implementation are orthogonal to configuration issues.
- Supports static configuration of components, logical structure and geometry not supporting any kind of dynamic configuration.

In [9] an extension to the work of *Polylith* is presented. This extension is focused on the introduction of reconfiguration capabilities to *Polylith*. This work adds the following characteristics. It supports all kinds of dynamic configuration. However, it does not decouple component functionality from dynamic configuration. and dynamic configuration instructions are embedded in the component functionality implementation code.

In *CORBA/Orbix* [16] an architecture supporting an object cooperation model and its implementation are presented. It presents the following characteristics:

- Allows component interface definition through the use of the *CORBA* interface description language. It does not allow any kind of component structure definition.
- Promotes the decoupling between functionality and configuration. Implementation of components is well separated from the configuration issues associated with the broker.
- Supports static configuration of components through registration of components on the broker. It supports dynamic configuration of components through registration and deregistration of components from the broker. The concepts of logical and geometric configuration are not present.

The *Service Configurator* pattern [17] presents a solution for service configuration within a server. It deals with the capability of adding or removing at run-time services and its implementation from a server. This way, a server can change the set of services it provides without having to be stopped. This pattern does not address component-based functionality or partitioning application configuration, focusing instead on server configuration.

The *Pipes & Filters* pattern [18] presents a design solution to pipeline based application development. This solution is specific to data flux processing applications. In this pattern, processing pipelines are defined as sequences of filter components where replacement or recombination of filter components are taken into account. It presents the following characteristics:

- The components interface are fixed and defined in the pattern. Structuring of the application is not foreseen by the pattern.

- Promotes the decoupling between component functionality and configuration. Filter functionality is independent of possible configurations.

- Because of the nature of flux processing applications configuration of pipelines must be heavily restricted on pipeline and flux state.

The patterns presented in this paper builds on the concepts and configuration capabilities found in these works. While these works all consider the subject of separation between functional an partitioning configuration important, they still do not make a clear separation between these two. Based on the classification in [9], the presented patterns allow the different kinds of configuration static and dynamic of component, logical structure and geometry. Component configuration is made possible by the use of generation and destruction of components. Logical configuration is made possible by the use of generation, destruction of components and connection establishment and disconnection. Geometry configuration is made possible at the partitioning configuration level by generation, destruction and migration of components from nodes.

5 Conclusions

In this paper two patterns where presented for the problem of functional and distribution configuration. The combination of both patterns was also presented and its integration in a framework was shown.

Being independent of each other, the presented patterns offer solutions to two complementary problems: application functionality configuration and application distribution configuration. The following qualities are achieved:

- The presented patterns decouple component functionality implementation from configuration issues.

- Component interface definition is possible at the functionality configuration pattern by the use of the primitive component structure. Specifically class `Component` and its method `defineComponent`.

- Component structure definition is possible at the functionality configuration pattern by the use of the composite component structure. Specifically class `CConfigurator` and its method `configureStructure`.

- Hierarchical structuring is possible by the use of class `CompositeComponent`.

- Reuse of component functionality implementation is achieved by its decoupling from configuration issues.

- Typed connection can be achieved by the patterns through variations in the patterns.

The presented patterns address the configuration specific problems:

- *Heterogeneous system configuration*, the presented patterns support configuration in heterogeneous systems. They make no assumptions on running platforms allowing distributed configurations with nodes in heterogeneous platforms.

- *Reconfiguration during system execution*, the presented patterns do not require the application to abort execution during reconfiguration. To components not directly involved in the reconfiguration process, this should occur transparently.

- *System consistency after migration*, the presented patterns guarantee consistency after migration. Brokers support independence of location guaranteeing consistency after migration.

- *Reconfiguration on quiescent states [19] only*, the presented patterns do not directly support the existence of a quiescent state during reconfiguration. This state could, however, be enforced through the correct implementation of methods `stateTransfer` from the partitioning configuration pattern.

- *Programmed and ad-hoc reconfiguration support*, the presented patterns support both programmed and ad-hoc reconfiguration. Programmed reconfiguration is supported by the existence of the basic instructions for reconfiguration. A mapping from configuration languages descriptions constructs on the structure provided by the pattern was achieved. The basic language constructs for static and dynamic configuration of component-based applications have their counterparts on the patterns.

- *Different communication mechanisms support*, the use of the presented patterns is independent of any communication mechanisms used. This way several communication mechanisms can be employed in conjunction with the patterns.

The advantages pointed out in introduction section, traceability, incremental development, application maintenance, software prototypes development and flexible systems development are achieved by the patterns. We believe that fault tolerance support can be simplified after enriching these pattern with fault tolerance specific techniques, as fault instrumentation and monitoring.

The configuration patterns were defined in the context of an approach to the development for distributed applications with separation of concerns (DASCo) initially described in [20]. Partitioning and functionality configuration are instances of DASCo concerns and the presented design patterns define solutions for them. Another concern relevant for partitioning configuration is distributed communication. The Distributed Proxy pattern [21] decouples the communication between distributed objects by isolating distribution-specific issues from object functionality. The composition of Distributed Proxy with Partitioning Configurer provides distributed communication allowing the transparent use of different communication technology, e.g. sockets or CORBA.

As future work we intend to relax the first two composition constraints by testing several possible combinations of `CConfigurator` with `PConfigurator`. Relaxing the former corresponds to allow vertical partitioning of components while relaxing the latter corresponds to support processes inside nodes. In particular, we believe that the latter can easily result from the current composition, however, the current framework implementation does not support it yet.

The three-layered framework for functionality and partitioning configuration is publicly available from the page `http://albertina.inesc.pt/˜ars/dasco`.

References

[1] J. Magee, N. Dulay, and J. Kramer. Regis: A Constructive Development Environment for Distributed Programs. Technical report, Department of Computing, Imperial College, London SW7 2BZ, UK.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[3] Peter Wegner. Frameworks for Active Compound Documents. Technical report, Brown University, 1997.

[4] Grady Booch. *Object-Oriented Analyis and Design with Applications*. The Benjamin/Cummings Publishing Company, Inc., 1994.

[5] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[6] Dirk Riehle. Composite Design Patterns. In *Conference on Object-Oriented Programming Systems, Languages and Applications, Proceedings*, pages 218–228, Atlanta, Georgia, USA, October 1997.

[7] Trygve Reenskaug, Per Wold, and Odd Lehne. *Working With Objects: The OOram Software Engineering Method*. Manning Publications Co., 1996.

[8] António Rito Silva. Development and Extension of Frameworks. In Saba Zamir, editor, *Handbook of Object Technology*. CRC Press, 1998.

[9] C. Hofmeister and J. Purtilo. A Framework for Dynamic Reconfiguration of Distributed Systems. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 560–571, 1991.

[10] J. Magee, N. Dula, and J. Kramer. A Constructive Development Environment for Parallel and Distributed Programs. Technical report, Department of Computing, Imperial College, London SW7 2BZ, UK, 1994.

[11] J. Magee, N. Dulay, and J. Kramer. Structuring Parallel and Distributed Programs. *IEEE Software Engineering Journal*, 8(2):73–82, March 1993.

[12] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. *Fifth European Software Engineering Conference*, September 1995.

[13] L. Belissard, S. Atallahm, F. Boyer, and M. Riveill. Distributed Application Configuration. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, Hong Kong, May 1996.

[14] Luc Belissard and Michel Riveill. Olan: A Language and Runtime Support for Distributed Application Configuration. *Journies du GDR de Programmation*, November 1995.

[15] J. Purtilo. The Polylith Software Toolbus. Technical Report CSD 2469, University of Maryland, 1990.

[16] IONA Technologies. Building Distributed Applications with Orbix and CORBA, 1996.

[17] Prashant Jain and Douglas Schmidt. Service Configurator: A Pattern for Dynamic Configuration and Reconfiguration of Communication Services. In *3rd Annual Pattern Languages of Programming Conference*, Allerton Park, Illinois, 1996.

[18] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley and Sons, 1996.

[19] Jeff Kramer and Jeff Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, November 1990.

[20] António Rito Silva, Pedro Sousa, and José Alves Marques. Development of Distributed Applications with Separation of Concerns. In *Proceedings of the 1995 Asia-Pacific Software Engineering Conference*, pages 168–177, Brisbane, Australia, December 1995.

[21] António Rito Silva, Francisco Assis Rosa, and Teresa Gonçalves. Distributed Proxy: A Design Pattern for Distributed Object Communication. In *The* 4^{th} *Conference on Pattern Languages of Programming, PLoP '97(Washington University technical report #WUCS-97-34)*, Allerton Park, Illinois, September 1997.