Executing Aspect-Oriented Component-Based Software Architectures on .NET Technology

Jennifer Pérez, Nour Ali, Cristóbal Costa-Soria, Jose A. Carsí, Isidro Ramos
ISSI, Department of Information Systems and Computation
Universidad Politécnica de Valencia
Camino de Vera s/n, 46022, Valencia, Spain
{jeperez | nourali | ccosta | pcarsi | iramos} @dsic.upv.es

ABSTRACT

Component-Based Software Development (CBSD) and Aspect-Oriented Software Development (AOSD) have emerged in the last few years as new paradigms of software development. Both approaches provide techniques to improve the structure and reusability of the code. In addition, Aspect-Oriented Programming (AOP) permits the reduction of the maintainability and development costs of the final code by means of the separation of concerns in *aspects*. However, the .NET framework does not provide support for the Aspect-Oriented approach. In this paper, we present a solution for this lack found in .NET technology by means of a .NET middleware called PRISMANET. PRISMANET is based on the PRISMA approach, which integrates the advantages of AOSD and CBSD and supports dynamic reconfiguration of software architectures at run-time. This middleware has been completely developed using the .NET framework and has been tested with real case studies, such as the *Teach Mover Robot.* As a result, PRISMANET extends the .NET technology by the execution of aspects on the .NET platform, the reconfiguration of software architectures (local and distributed) and the addition and removal of aspects from components at run-time.

Keywords

Aspect-Oriented Programming (AOP), Component-Based Software Development (CBSD), dynamic reconfiguration of software architectures, addition and removal of aspects at run-time, concurrency, distribution, mobility.

1. INTRODUCTION

Complex structures, non-functional requirements, reusability and run-time evolution are leading properties that current software systems need to deal with. Two software development approaches have emerged to respond to these needs: Component-Based Software Development (CBSD) [Szy98] and Aspect-Oriented Software Development (AOSD) [AOS05].

On the one hand, CBSD decomposes the system into reusable entities called components that provide services to the rest of the system.

.NET Technologies'2005 *conference proceedings,*
ISBN 80-903100-4-4

On the other hand, AOSD allows for the separation of concerns by modularizing crosscutting concerns into a separate entity, called *aspect*. The encapsulation of the aspect allows for the reusability of the same aspect in different objects and the evolution of an aspect without affecting the rest of objects and aspects. The main effort in this approach has been made at the implementation level. As a result, this approach has emerged as a new paradigm of software development. However, the .NET framework does not provide support for this new approach, making the use of .NET technology by the "Aspect-Oriented Community" unfeasible.

In this paper, we present a solution that provides support to the AOSD by means of the PRISMANET middleware. PRISMANET implements PRISMA. PRISMA is an approach to develop complex information systems that provides a model and an Architecture Description Language (ADL).

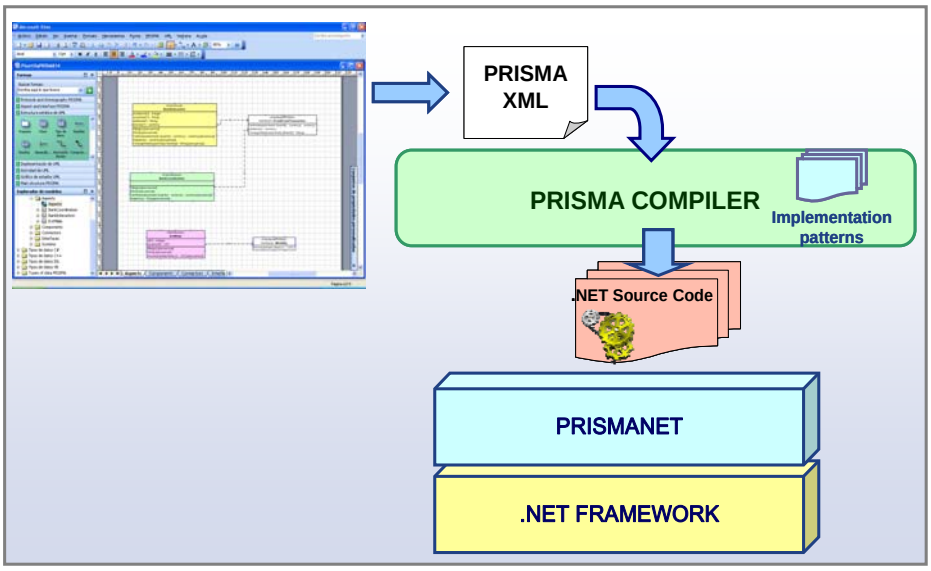

Figure 1. PRISMA approach

The PRISMA model defines software architectures by integrating AOSD and CBSD. In addition, PRISMA supports evolution by means of a meta-level. Its meta-level allows the evolution of types and the dynamic reconfiguration of architectures. The PRISMANET includes the PRISMA model, its meta-level and the distribution support for mobility.

PRISMANET not only extends the .NET technology by the incorporation of aspects, but it also provides the reconfiguration of software architectures (local and distributed) and the addition and removal of aspects from components at run-time. These complex features have been successfully implemented thanks to the mechanisms that the .NET technology provides to deal with them. The most important .NET technology mechanisms [Rob03] that we have used are: delegates, reflection, serialization, .NET Remoting [Mic05] and dynamic code generation.

As a result of the PRISMANET implementation, PRISMA software architectures can be developed and can be executed on the .NET platform. In addition, PRISMANET allows .NET programmers to develop applications with aspect-oriented, mobility and dynamic evolution properties.

Currently, the PRISMA approach allows the development of software systems with all its advantages by extending PRISMANET classes. The middleware has been developed and tested with real case studies, such as the *TeachMover* robot [Tea05] and the *EFTCOR* teleoperation system [EFT02] to clean the hulls of the ships. As a result of this work, we are able to move the robot with the Aspect-Oriented .NET technology and to develop the PRISMA CASE model compiler based on the middleware. For this reason, we are currently developing the compiler in order to automatically generate C# code from graphical diagrams (see Figure 1).

The goal of this paper is to show how the aspect-oriented, mobility and run-time evolution properties of PRISMANET have been implemented using the .NET technology mechanisms that have been previously mentioned.

The structure of the paper is as follows: Section 2 briefly introduces the basic concepts of the PRISMA model to understand the middleware implementation. Section 3 explains the PRISMANET middleware implementation in detail: aspects and components, concurrency, mobility and run-time evolution. Section 4 presents a comparison with other approaches that introduce aspect-oriented programming in .NET technology and points to the disadvantages that PRISMA overcomes. Finally, conclusions and further work are presented in section 5.

2. PRISMA

The PRISMA model allows for the definition of architectures of complex software systems [Per03]. Its main contributions are the integration of the AOSD and the CBSD and its reflexive properties. In this way, it specifies different characteristics (distribution, safety, coordination, etc.) of an architectural element (component, connector) using aspects, and it is able to evolve its architectures by means of a meta-level.

A component is an architectural element that captures the functionality of the system and does not act as a coordinator among other elements. However, a connector is an architectural element that acts as a

coordinator among components. In the PRISMA model, the connector does not have the references of the components that it connects to and vice versa. In this way, both components and connectors are reusable. The channels between two connected architectural elements have their references. The channels that connect components and connectors are called *attachments.*

Architectural elements can be seen from two different views, internal (*white box view)* and external (*black box view*). The *white box view* shows an architectural element as a prism being an aspect of this architectural element each side of the prism (see Figure 2); whereas, the *black box view* encapsulates its functionality and publishes a set of services that offers to other architectural elements (see Figure 3).

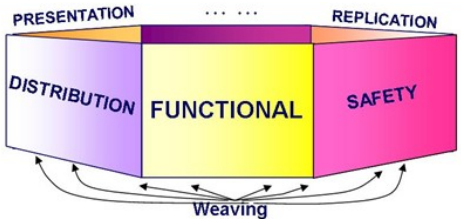

Figure 2. White box view of a component

A PRISMA aspect represents a specific *concern* (safety, coordination, distribution, etc) that crosscuts the software architecture. This means that those *concerns* that do not crosscut the architecture are not going to be an aspect. In order to avoid these *crosscutting-concerns*, a PRISMA architectural element is formed by a set of aspects that describe it from the different *concerns* of the architecture. The kinds of aspect (safety, coordination, distribution, etc) that form an architectural element depend on the *concerns* of the information system that is being specifying.The main elements that form an aspect are the following:

- *Attributes*: store information about the characteristics of the aspect.
- *Valuations*: specify the changes in attribute values by the execution of a service.
- *Services:* offer functionality of a specific *concern.*
- *Protocols:* describe the order and the state in which a service could be executed.

A component is formed by a set of aspects (functional, distribution, etc.), their synchronization relationships (*aspects weaving*) and one or more ports. These ports represent the interaction points among components. The type of ports is an interface that publishes a set of services.

The weaving is the glue of the aspects forming a prism. The weaving determines how an aspect is connected (synchronized) with the rest of the aspects. It indicates that the execution of an aspect service can generate the invocation of services in other aspects. However, to preserve the independence of the aspect specification from the aspect weaving, the weaving is specified outside the aspect and inside the component. The weaving methods are operations that describe the causality of the weaving services. The weaving methods are commonly used in the AOP. They are as follows:

- **after**: aspect1.service is executed **after** aspect2.service
- **before**: aspect1.service is executed **before** aspect2.service
- **instead**: aspect1.service is executed **in place of** aspect2.service

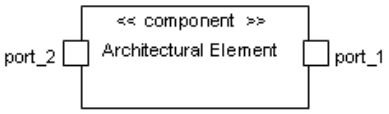

Figure 3. Black box view of a component

3. .NET MIDDLEWARE

PRISMA ADL (Architecture Description Language) is a specification language independent of the development platform. For this reason, an abstract middleware that sits above the .NET platform has been developed to implement .NET PRISMA applications. This middleware is called PRISMANET, and its implementation has been carried out in C# language using the standard techniques that the .NET framework provides, that is, without extending the development platform. As a result, PRISMANET can be executed in the .NET platform without having to do anything else other than starting the execution of the middleware.
PRISMANET offers the extra functionalities and characteristics which .NET does not directly provide. It allows for the execution of aspects, the reconfiguration of software architectures (local and distributed), the load of components, the creation of execution threads, the management of the local components, the addition and removal of aspects from components at run-time, the mobility and replication, etc.

PRISMANET architecture

The PRISMANET architecture is constituted by two modules: server and framework (see Figure 4):

- **PRISMA Server:** This module provides services to manage, move, maintain and evolve components.
- **PRISMA Framework:** This module is the user interface that offers the user the available services of the Server module. In addition, the state messages of the middleware are displayed on this user interface.

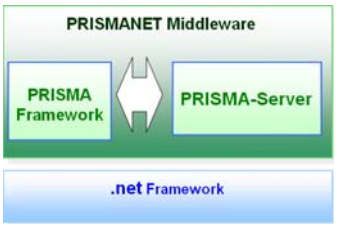

Figure 4. PRISMA Middleware

As PRISMA specifies software architectures of distributed systems, distribution needs has also been taken into account in the development of the middleware. PRISMANET has to run on each node where a PRISMA application needs to be executed (see Figure 5). Each middleware manages the architectural elements instances that are being executed in its specific node, providing the necessary distribution, mobility, maintenance and evolution services to the instances. In order to keep the consistence of distributed software architectures and to make the instances work as if they were local instances, each middleware is able to interchange information with the other middlewares of the different nodes of a software architecture.

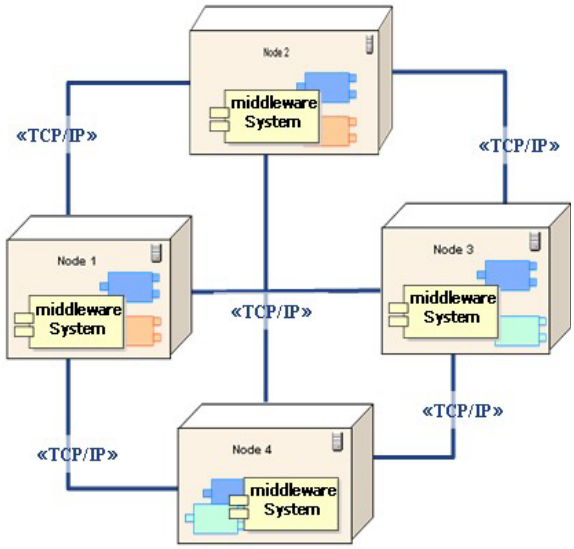

Figure 5. PRISMA Middleware running in distributed nodes

There are three kinds of communications concerning PRISMANET and the applications that run on it:

- Calls from the components to the middleware to ask for mobility and replication services.
- Communication among different components as a result of the execution of the application.
- Communication among different middlewares to find out locations of components, to move components, to evolve the architectures, etc.

PRISMA Model Implementation

Each concept defined in the PRISMA model has been implemented in the Server module of the PRISMANET. In this section, we focus on the aspects and components. The implementation has been carried out preserving the following features:

- The run-time evolution of applications must be possible. As a result, the dynamic code generation to add and remove aspects, components, connectors and attachments must be allowed.
- The implementation has to be as close as possible to the model in order to facilitate the future automatic code generation.
- The execution of attachments, connectors and components must be concurrent. In addition, the concurrency among the different aspects that form a component must be preserved.

3.2.1. Aspects

An aspect has been implemented as a C# class called *AspectBase* of the PRISMANET. This class stores the name of the aspect and its thread reference.

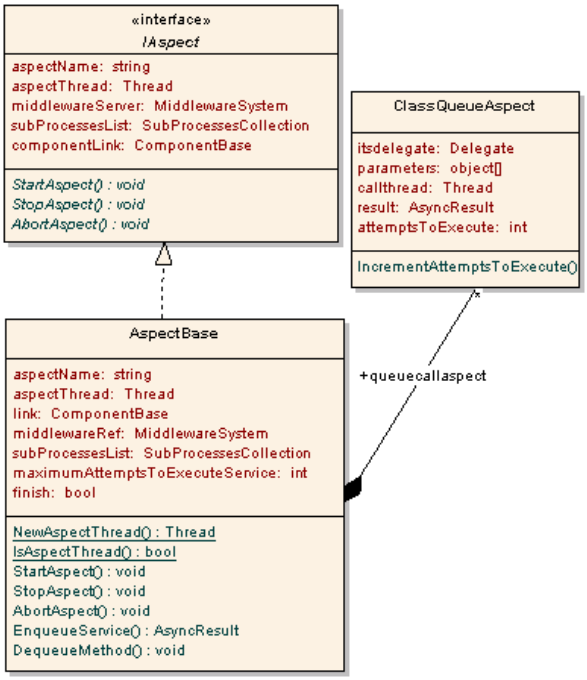

Figure 6. *AspectBase* class of PRISMANET middleware[1]

The *AspectBase* class has the references of the component and the middleware that it belongs to in order to request them services. In addition, as the

[1] The set of classes that appear in the figure have been automatically generated from the source code of PRISMANET using the Sparx tool [Spa05]

middleware must guarantee the execution of services without blocking the requesters, when a service of an aspect requires its execution while another service is being processed, the aspect stores the service that can not be immediately attended in a queue. As a result, the aspect thread is continuously processing the requests of the queue (see Figure 6). Finally, it is important to emphasize that the *AspectBase* class offers three services: *startAspect* to start the execution of the aspect thread, *stopAspect* to stop the execution of the aspect thread and *abortAspect* to definitively stop the execution of the aspect thread.

The kinds of aspects that can be defined in the PRISMA model are unlimited. However, each one has the functionality described above. For this reason, they are a subclass of the *AspectBase* class and inherit this functionality (see Figure 7).

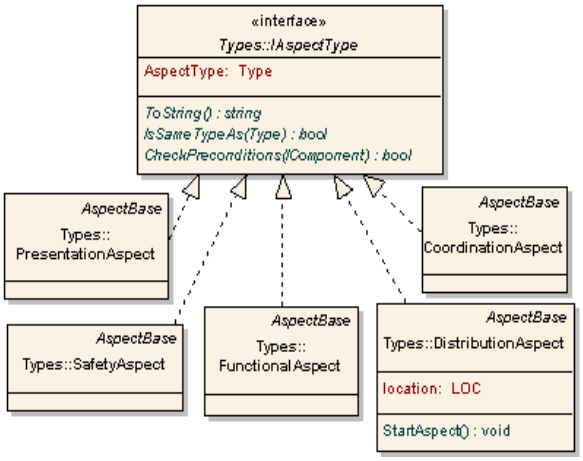

Figure 7. Classes of several kinds of aspects[1]

As a result, PRISMANET allows the implementation of a specific aspect by creating a C# class that inherits from one of the classes that represent one kind of aspect. It is important to keep in mind that aspects must be *serializable* in order to enable the mobility of aspects in distributed architectures.

In a specific aspect, the PRISMA *attributes* are implemented as private variables. The PRISMA *services* are programmed as private methods that implement their respective *valuations*. They also check whether their execution is enabled in accordance with the established order of the *protocol*. An example of a specific safety aspect is presented below:

```
using PRISMA;
using PRISMA.Aspects;
using PRISMA.Aspects.Types;
using PRISMA.Middleware;
using PRISMA.Attachments;

namespace Robot
[Serializable]
public class SMotion : SafetyAspect
{
  #region Definition of PRISMA Variables
      double minimun;
      double maximun;
  #endregion

  public SMotion(double initialMinimum,
                 double initialMaximum) :
                 base("Smotion")…

  public AsyncResult Check(double newAngle,
                 out bool secure)…
}
```

Finally, it is important to emphasize that specific aspects are packaged in an assembly in order to facilitate their distribution over the network and their integration in a library.

3.2.2. Components

A component has been implemented as a C# class called *ComponentBase* of the PRISMANET. This class stores the name of the component, its own thread reference and its middleware reference and the dynamic list of aspects. It stores two attributes to control whether the component is going to stop or move, as well as the references to the ports to be able to receive and request services. In addition, the *ComponentBase* class offers the following services: *Start* to initiate the component thread execution; *Stop* to stop temporarily the component thread execution; *Abort* to stop definitively the component thread execution; *IsWeaved* to query if an aspect of the component is weaved with another aspect; *AddAspect* and *RemoveAspect* to add and remove aspects from a component; and *AddWeaving* and *RemoveWeaving* to add and remove weavings from a component (see Figure 8).

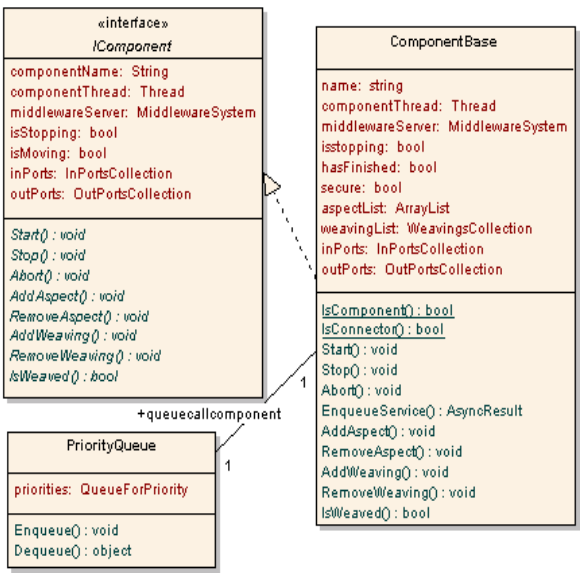

Figure 8. *ComponentBase* class of PRISMANET middleware[1]

3.2.3. Weavings

Weavings have been implemented as a dynamic linked list with three levels of depth. This list is part of the component that it belongs to. Thus, this weaving implementation facilitates the management and evolution of the weavings. The dynamic list is implemented by the *WeavingsCollection* C# class. Each element of this dynamic list is an instance of the *AspectTypeNode* C# class that contains the aspect type and another dynamic list called *weavingAspectList.* Each element of the *WeavingAspectList* is an instance of the *WeavingNode* C# class. This class stores the service name, which triggers the weaving execution as well as, a delegate of this service for its dynamic invocation. It also stores three more lists, each of which belongs to a weaving operator (*after, before, instead).* These lists contain instances of the *WeavingMethod* C# class. This class stores the delegate, which points to the method that must be executed as a result of the weaving (*methodDelegate).* It also stores the method that has triggered the weaving execution (*origMethod),* and the weaving type (see Figure 9).

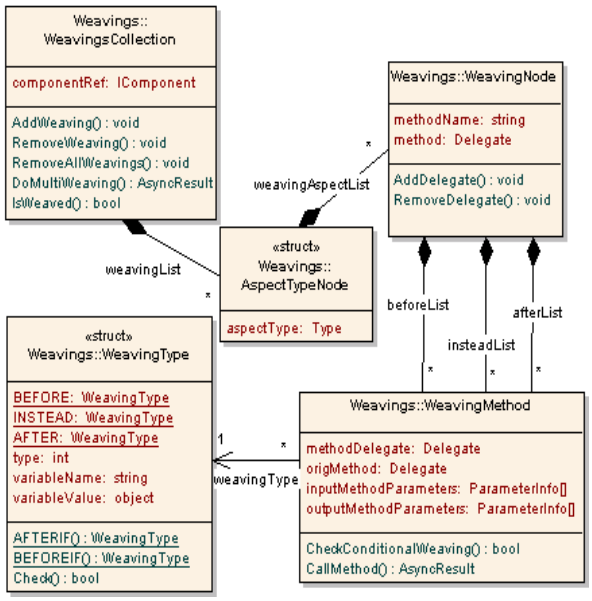

Figure 9. Dynamic list of weavings[1]

As a result, PRISMANET allows the implementation of a specific component by creating a C# class that inherits from the *ComponentBase* class. It is important to keep in mind that components must be *serializable* in order to enable the mobility of components in distributed architectures. An example of a component called *Actuator* is presented below:

```
using System;
using System.Reflection;
using PRISMA;
using PRISMA.Aspects;
using PRISMA.Aspects.Types;
using PRISMA.Components;
using PRISMA.Middleware;

namespace Robot
{
[Serializable]
 public class Actuator : ComponentBase
 {

   public Actuator(string name,
   MiddlewareSystem middlewareSystem) : base
   (name,middlewareSystem)
   {
   /* ***************************
    * * DEFINITION OF ASPECTS * *
    ***************************/
   // Creation of Functional Aspect
   AddAspect(new FActuator());
   // Creation of Safety Aspect
   AddAspect(new  SMotion(initalMinimum,
                          initialMaximum));

  // Achieving the references of the aspects
   IAspect functionalAspect =
        GetAspect(typeof(FunctionalAspect));
   IAspect safetyAspect =
            GetAspect(typeof(SafetyAspect));

  /* ****************************
   * DEFINITION OF  WEAVINGS * *
   ****************************/
   // Weaving MoveJoint
    AddWeaving(functionalAspect,"MoveJoint",
         WeavingType.AFTERIF("secure",true),
            safetyAspect,"Check",functions);

   /* ***************************
    * * DEFINITION OF PORTS * *
    ***************************/
    InPorts.Add("IMotionJointPort",
                "IMotionJoint",
                 functionalAspect);

    OutPorts.Add("IMotionJointPort",
                 "IMotionJoint");}}}
```

Execution Model

When the execution of a service is requested from a component, the request comes from the port that publishes the service (step 1, Figure 10). The port sends the request to the queue of the component (step 2, Figure 10). Once the component thread extracts the requested service from the queue, the component checks if the requested service has weavings associated to it (step 3, Figure 10). If the service does not have any weavings, its delegate is asynchronously executed so that the component can process another request from the queue. The delegate execution consists of adding the service to the queue of the corresponding aspect. Next, the aspect thread executes the service (step 5, Figure 10). However, if the service has weavings associated to it, before executing step 5, the service is sent to the weaving manager (step 4, Figure 10). The manager processes weavings creating its own thread and freeing the component from this task.

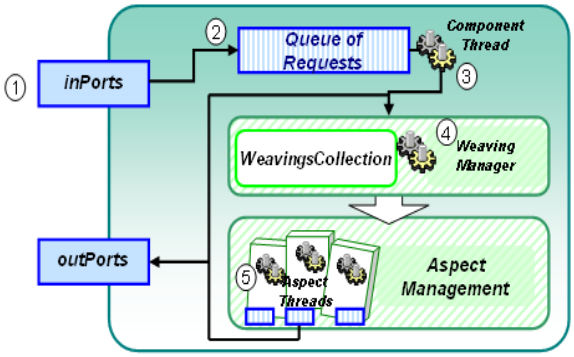

Figure 10. The execution model of a component

With regard to starting or stopping a component, when the middleware calls the *start* service of a component, the component calls the *startAspect* service of each one of its aspects. On the other hand, when the middleware calls the *stop* or *abort* services of a component, the threads of its aspects must also be stopped (*stopAspect)* or aborted (*abortAspect*). In the case of stopping a component in a secure way (*stop* service), a set of operations must be performed in order to achieve a secure state that will permit the start of the component execution in the future. A component is in a secure state when it does not have requests in its aspect queues and there are no executing services. These operations consist of not allowing anymore services in their queues and processing every service that was stored in the queue before the stop execution.

Adding and removing aspects at run-time

Aspects can be added to and removed from a component at run-time. The *addAspect* service inserts a new aspect inside a component. This method verifies that the kind of aspect that is going to be added does not already exist in the component, since only one aspect of each kind can exist in a component. The method updates the references of the aspect to the component and middleware and adds the aspect to the aspect list of the component. Finally, dynamic code generation is used to update the component constructor in order to make the changes consistent. The *removeAspect* service deletes an aspect from a component. First, the method stops the aspect that is going to be removed in a secure way. Second, it removes the aspect from the aspect list of the component and its associated weavings. Finally, the dynamic code generation is used to update the changes.

Distribution Model and Mobility

PRISMANET supports the distributed communication and the mobility of the components. It provides the distributed communication among components without making components aware of each other.

3.5.1. Distributed Communication among elements through Attachments

To make components as reusable as possible, they do not have references to other components they communicate with. Therefore, the components are unaware of the components they communicate with. The distributed communication among components is the responsibility of attachments. Thus, an attachment has the references of the communicating components.

To support attachments, the middleware contains three classes: the *Attachment* class, the *AttachmentServerBase* class and the *AttachmentClientBase* (see Figure 12). For each component port, there is at least one instance of an *Attachment* class. When a component instance is created, the PRISMANET middleware creates the instances of the attachments associated to each port. Each PRISMA port has been implemented into two queues, a client (outPort) and a server queue (inPort) (see Figure 10), there also exists a Server Attachment and a Client Attachment for each Attachment. An instance of the *Attachment* class automatically instantiates an *AttachmentClientBase* and an *AttachmentServerBase* class.

An *AttachmentClientBase* instance has a thread that listens to a specific outport of a component instance. When the *AttachmentClientBase* instance detects that there is a petition in the queue, the petition is redirected to the instance of an *AttachmentServerBase*. Thus, the *AttachmentClientBase* instance has a reference or a proxy of the *AttachmentServerBase*. The *AttachmetServerBase* is a MarshalByRefObject class of the .NET Remoting framework. This has been necessary to create a proxy of the instance to allow the *AttachmentClientBase* instance to access to it remotely.

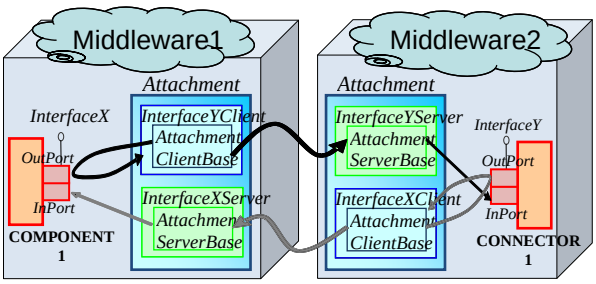

Figure 11. Two distributed architectural elements connected by attachments

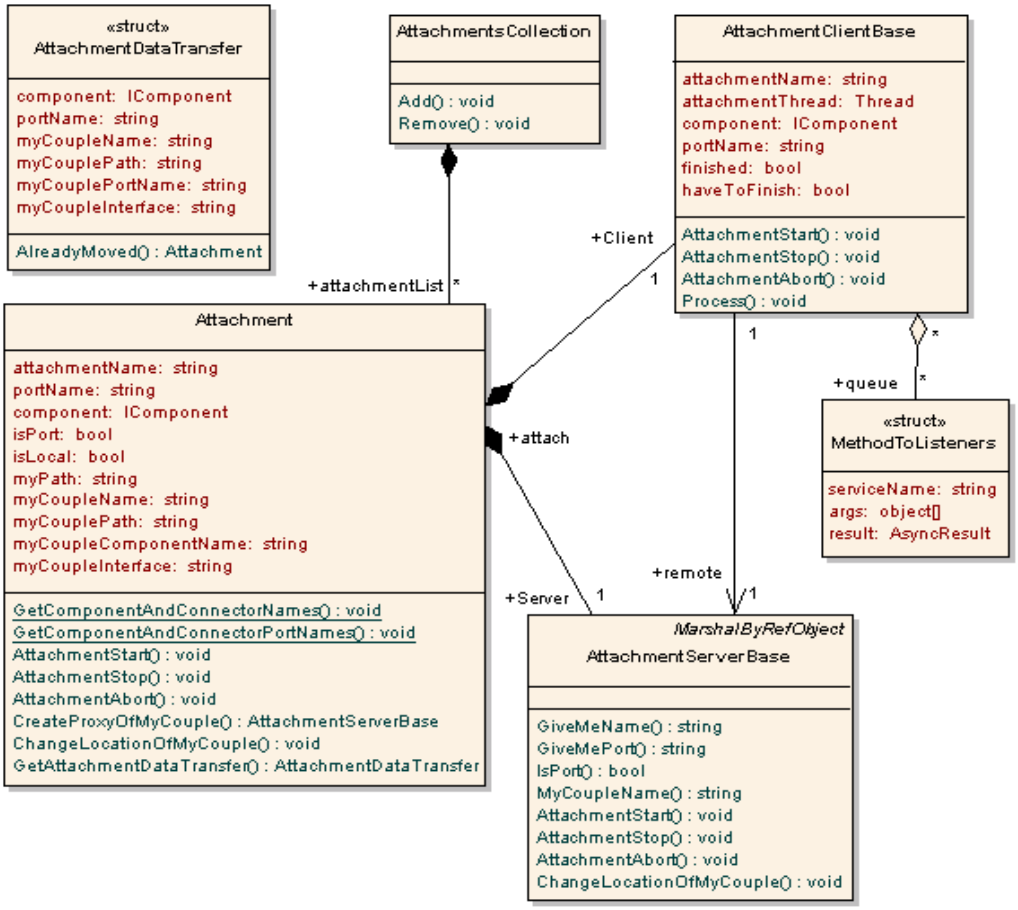

Figure 12. A logical view of the attachments in the middleware[1]

Figure 11, shows how two distributed components are connected together. Component1 has an AttachmentClientBase that listens to its Output queue and redirects services to the AttachmentServerBase of Component2. In addition, Component2 also has an AttacmentClientBase that listens to its Output queue and redirects the services to the AttachmentServerBase. Each AttachmentClientBase and AttachmentServerBase of a component are associated by an Attachment.

The attachments are solely responsible for the distributed communication of the components. The PRISMANET middleware only participates in the creation of attachments instances between its component instances. To store the list of attachments in its site, each middleware has an *AttachmentCollection* class (see Figure 12).

The use of attachments approximation for distributed communication does not only allow for the reusability of the elements but also makes distributed applications independent of a centralized Domain Name Server (DNS). Thus, the attachments of a component can be seen as a distributed DNS that contains the necessary references that allow an instance to perform the needed communications. Our approach prevents the failures which may be generated as a result of failures produced by a centralized DNS such as load saturation and deadlock. In addition, if a certain attachment between two architectural elements fails, their communication among others is not affected.

3.5.2. Mobility of the elements in PRISMA

Mobility is defined as the process of transferring a component instance and its code to a new host. The transferred component instance must continue executing at the new host, while conserving its state and maintaining the same execution point.

Current technologies do not offer this definition of mobility nor does .NET. Therefore, the mobility has to be simulated. To implement the mobility, we have marked all classes of components, aspects and the inPorts and outPort queues of the component with the [Serializable] attribute. The ability to serialize (to pass them by value) is provided by the .NET Framework. However, this is not enough. The mobility process has to ensure that the instance is at a consistent state before it is serialized. The steps to enable a mobility process are presented in the following section.

3.5.2.1. The execution of a mobility decision

A distribution aspect of a component encapsulates the different decisions related to mobility. This enables the reusability of the different mobility decisions in different components. As a result, the component controls the mobility decisions and even if the environment wants to make a mobility decision, it has to go through the distribution aspect of a component.

Figure 13 shows an interaction diagram of part of the mobility process performed at the site where the mobility decision of a component has been executed. It shows the interchange of messages until the component instance is transferred to the RemoteMiddleware.

When a mobility decision is satisfied, the distribution aspect asynchronously calls the PRISMANET middleware on its site to indicate that it is willing to move (*move*, Figure 13). The distribution aspect also notifies its component thread to prepare itself to be in a secure state so that it can be serialized (by executing the *Stop* of a Component, Figure 13). A component is at a secure state when the queues of its aspects are empty and when there is no service being executed. Therefore, the component thread stops processing services from its queue. However, services can be queued in the component inport because the queue is also serializable. In addition, the component thread notifies the aspect threads to stop when they finish processing services from their queues and when they finish executing all the services (*StopAspect*, Figure 13). When the component and aspect threads finally stop, the PRISMANET is notified and it is able to move the component.

3.5.2.2. Preparing to Move Attachments

The attachments also have to be prepared for mobility if a component is moved. This is because the attachments are the communication channels that allow others to communicate with a specific component. Therefore, it is also important to involve the attachments when a component is moved.

When a component instance is completely stopped, PRISMANET executes a service called *PrepareTo-MoveAttachments* (Figure 13). This service fetches the attachments of a component by going through its ports and finding the listeners to these ports. It checks which attachments connect the mobile component with distributed ones. The information associated with each attachment is saved in a structure called the *AttachmentDataTransfer* (see Figure 12). The information contains the references of the *AttachmentServerBase* instances that are connected to the *AttachmentClientBase* instances of the component. It also contains the name of the component instance that is connected to the mobile component instance, and others.

When this task is completed, the middleware stops the thread of the *AttachmentClientBase* instances of the component. In addition, the *AttachmentServer-Base* instances of the distributed component instances, connected to the mobile component instance, are unregistered from Remoting by using the *Disconnect(MarshalByRefObject)* service. This is previously performed in order not to allow the transfer of services to and from the mobile component object.

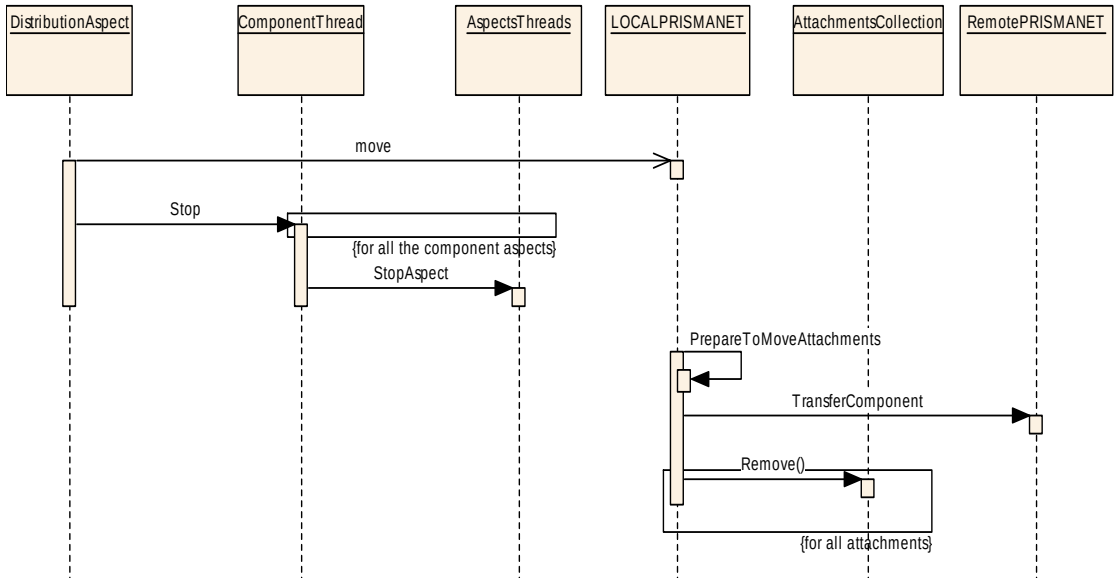

Figure 13. A simple interaction diagram showing the tasks done by at the local middleware site of the transferred component

Finally, a list with all the *AttachmentDataTransfer* structures of a component is created. This is necessary for the new middleware, where the component is going to be transferred, to allow it to recreate the attachments on its site.

3.5.2.3. Transferring component instances

When all the information for the mobility is prepared, the component instance with InPorts and OutPorts is serialized, and the list with the information of the attachments is transferred to the middleware of the new host. The transfer process is performed in a try/catch block in order to recover from any failure that may occur while making the transfer (*TransferComponent*, Figure 13). When the object is correctly serialized, the original component object is destroyed.

In addition, the attachments associated with the mobile component are removed from the list of attachments that exist in the site of the current middleware by executing the *Remove()* method of the *AttachmentsCollection* (*Remove*, Figure 13).

3.5.2.4. Process after transferring a component instance

When the component instance is moved, the receiving middleware updates the list where it stores the components that are executing on its site by adding the component instance moved (*componentList.Add* Figure 14).

Then the middleware uses the information stored in *AttachmentServerBase* structure list in order to create the Attachments of the component instance (*createLocalAttachment*, see Figure 14). However, this is not enough because the instances of the *AttachmentClientBase* of other component instances that are connected to the instances of the *AttachmentServerBase* still have the old references or proxies. Therefore, the new proxies of each *AttachmentServerBase* of the component instance are sent to the connected instances of the *AttachmentClientBase* (*sendNewLocationToCouple*, Figure 14). Afterwards, the thread of each *AttachmentClientBase* of the component instance is started as well as the thread of the component instance.

As the inPorts have not been stopped while the moved component instance was preparing itself to be in a secure state and to be moved, the component instance can start executing the services which where queued at the first middleware and were not processed.

In this way, a mobility approximation has been implemented preserving the state of the object after moving it.

Our approach clearly distinguishes between moving an object and allowing remote calls to it. This can be done thanks to the implementation of the attachments. In .NET Remoting, instances cannot be MarshalByRefObjects and serializable at the same time. As well, even if a MarshalByRefObject is serialized a proxy is created, and it is not the real object that is transferred. Therefore, using attachments the indirect reference remoting to component instances is allowed as well as their mobility.

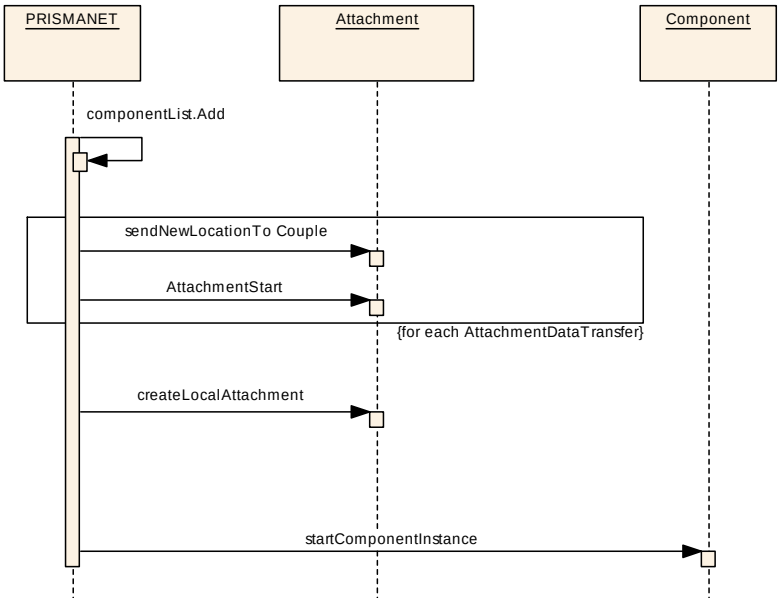

Figure 14. An interaction diagram showing the tasks done by the new middleware site after the component is transferred

4. RELATED WORKS

Currently, there is an increased interest in Aspect-Oriented Programming (AOP) which is becoming a widely used programming technique. AOP was initially developed for Java environments through AspectJ [Kic01] and is being transferred to other platforms such as .NET by means of extensions. However, the existing .NET approaches for supporting AOP are still in an early phase.

AspectC#[Kim02] and SourceWeave.Net [Jac04] support AOP in .NET having available the source code of the base code, the aspects and the weavings. These approaches propose joining the base code with the aspects by specifying the weavings in an XML file. Weave.Net [Laf03] and AspectDNG [Asp05] also define the weavings through an XML file; however, they only use the assemblies of the base code, the aspects and weavings to join the code without being available the source code. Loom.Net [Sch02] is another .NET approach for supporting AOP. It has a graphical interface that allows the addition of defined aspects by means of reusable code templates and allows the performance of weavings.

The approaches mentioned above clearly separate the base code, the aspects and the weavings in different entities. However, none of them supports mechanisms for dynamically adding or removing aspects. The Rapier-Loom.Net [Sch03] approach does allow dynamic addition and removal of aspects, but it defines the weavings inside the aspects thereby losing their reusability. SetPoint [Set05] also allows for dynamic addition and removal of aspects. Its weaving is based on the evaluation of logical predicates in which the base code is marked with meta-information that permits the evaluation of such predicates. EOS [Raj03] is another dynamic approach which is able to attach aspects at instance-level by means of events.

None of the approaches mentioned above takes into account the emerging relations that result from the aggregation of various aspects at the same point of the base code (joinpoint). However, JAsCo.Net [Ver03] provides an expressive language that permits the definition of relations among aspects. JAsCo.Net integrates AOP and CBSD. It introduces the concept of connectors for the weaving between the aspects and the base code which allows for a high level of aspect reusability. An inconvenience of this approach is that the dynamic weaving of aspects to the base code is referential but not inclusive. This requires an execution platform to intercept the application and insert it into the aspects at execution time.

The principal disadvantage of these approaches is that none of them integrates the needed properties at the same time to allow the mobility, the reusability and the evolution of aspect-oriented components. These properties are the dynamic weaving, the join of the base code and the aspects inside the same entity and the reusability of aspects. Therefore, the code mobility is limited because not all the properties of the object code can be moved. However, PRISMA defines a model that combines AOP and the dynamic reconfiguration of the CBSD models. The aspects are separately defined from the weavings and are highly reusable. The components are formed from aspects which are inclusively and can be dynamically aggregated. In addition, PRISMA permits the dynamic mobility of its components, and the concept of base code does not exist, so the component is solely formed by aspects. The implementation of the PRISMA model in .NET permits the dynamic addition and removal of aspects as well as the dynamic modification of the weavings without stopping the execution of the component.

5. CONCLUSIONS AND FURTHER WORK

In this paper, an innovative middleware called PRISMANET has been presented. This middleware is based on PRISMA model and in this way, it allows the implementation of complex, dynamic, distributed, aspect-oriented and component-based software systems using C# language. PRISMANET has been developed with C# language using the standard techniques that the framework provides, that is, without extending the development platform. As a result, PRISMANET can be executed in every computer that has the .NET framework installed.
PRISMANET offers extra functionalities for the .NET platform. It allows the execution of aspects, the reconfiguration of software architectures (local and distributed), the addition and removal of aspects from components at run-time, mobility, etc. As explained in the paper, these functionalities have been implemented using .NET mechanisms such as delegates, reflection, serialization, .NET Remoting and dynamic code generation.

PRIMANET has also been tested in industrial case studies such as the EFTCoR teleoperation system and the *TeachMover* robot. We are now working on the PRISMA model compiler in order to integrate the PRISMA graphical interface and the middleware in a CASE tool and to automatically generate code from the graphical diagrams.

6. ACKNOWLEDGMENTS

This work has been funded by the Department of Science and Technology (Spain) under the National Program for Research, Development and Innovation, *DYNAMICA* project TIC2003-07776-C02-02. PRISMANET and industrial case studies developments are funded by the Microsoft Research Cambridge, *"PRISMA: Model Compiler of Aspect-oriented component-based software architectures"* Project [PRI05].

All the class diagrams shown in this work have been automatically generated from the source code of the PRISMANET middleware, by using the Enterprise Architect Modeling tool from Sparx Systems [Spa05]. This tool was selected because it supports the automatic generation of class diagrams from source code, as well as the traceability between the generated classes and the original source code.

7. REFERENCES

[Aos05] Aspect-Oriented Software Development, http://aosd.net

[Asp05] AspectDNG Project, http://aspectdng.sourceforge.net

[EFT02] EFTCoR Project: Friendly and Cost-Effective Technology for Coating Removal. V Programa Marco, Subprograma Growth, G3RD-CT-2002-00794, 2002.

[Kic01] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G., An Overview of AspectJ. In ECOOP 2001, (Budapest, Hungary), Springer, pp.327-355.

[Kim02] Kim, H., AspectC#: An AOSD implementation for C#. MSc. Thesis, Comp.Sci, Trinity College, Dublin, Dublin, 2002.

[Jac04] Jackson A., Clarke S., SourceWeave.NET: Cross-Language Aspect-Oriented Programming. In Proc. of Generative Programming and Component Engineering (GPCE). Vancouver, Canada, 2004.

[Laf03] Lafferty D., Cahill V., Language-Independent Aspect-Oriented Programming. In Proc. of Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2003). Anaheim, California, USA, 2003.

[Mic05] Microsoft .Net Remoting, A Technical Overview, http://msdn.microsoft.com/library/default.asp?url=/library/en-/dndotnet/html/hawkremoting.asp

[Per03] Perez J., Ramos I., Jaén J., Letelier P., Navarro E., PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures. In proceedings of 3rd IEEE Int. Conf. on Quality Software (QSIC), Dallas, Texas, USA, November 2003, IEEE, pp. 59-66.

[PRI05] PRISMA, http://prisma.dsic.upv.es

[Raj03] Rajan, H., Sullivan, K., Eos: Instance-Level Aspects for Integrated System Design. In the proc. of the Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), Helsinki, Finland, September 2003.

[Rob03] Robinson S. et al., Professional C#, 2nd Edition. Wrox Programmer to Programmer.

[Sch02] Schult, W. and Polze, A., Aspect-Oriented Programming with C# and .NET. In 5th IEEE International Symposium on Object-Oriented Real-time Distributed Computing (Washington, DC, 2002), IEEE, pp.241-248.

[Sch03] Schult, W. and Polze, A., Speed vs. Memory Usage – An Approach to Deal with Contrary Aspects. In 2nd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS). Boston, Massachusetts, 2003.

[Set05] SetPoint! Project, http://www.dc.uba.ar/people/proyinv/setpoint/

[Spa05] Sparx Systems: Enterprise Architect Modeling Tool, http://www.sparxsystems.com.au

[Szy98] Szyperski C., Component software: beyond object-oriented programming. *ACM Press and Addison Wesley*, New York, USA, 1998.

[Tea05] The *TeachMover* Robot, http://www.questechzone.com/microbot/teachmover.htm

[Ver03] Verspecht, D., Vanderperren, W., Suvee, D. and Jonckers, V., JAsCo.NET: Capturing Crosscutting Concerns in .NET Web Services. In Proc. of Second Nordic Conference on Web Services NCWS'03, Vaxjo, Sweden. Published in Mathematical modeling in Physics, Engineering and Cognitive Sciences, Vol. 8, November 2003.