.NET Technologies 2006

University of West Bohemia
Campus Bory

May 29 – June 1, 2006

Full papers proceedings

Edited by

Jens Knoop, Vienna University of Technology, Austria
Vaclav Skala, University of West Bohemia, Czech Republic

.NET Technologies – FULL papers conference proceedings

Editor-in-Chief: Vaclav Skala
University of West Bohemia, Univerzitni 8, Box 314
306 14 Plzen
Czech Republic
skala@kiv.zcu.cz
Managing Editor: Vaclav Skala

Author Service Department & Distribution:
Vaclav Skala - UNION Agency
Na Mazinach 9
322 00 Plzen
Czech Republic
Reg.No. (ICO) 416 82 459

Hardcopy: ***ISBN 80-86943-10-0***

CONFERENCE CO-CHAIR

Knoop, Jens (Vienna University of Technology, Vienna, Austria)
Skala, Vaclav (University of West Bohemia, Plzen, Czech Republic)

PROGRAMME COMMITTEE

Aksit, Mehmet (University of Twente, The Netherlands)
Giuseppe, Attardi (University of Pisa, Italy)
Gough, John (Queensland University of Technology, Australia)
Huisman, Marieke (INRIA Sophia Antipolis, France)
Knoop, Jens (Vienna University of Technology, Austria)
Lengauer, Christian (University of Passau, Germany)
Lewis, Brian,T. (Intel Corp., USA)
Meijer, Erik (Microsoft, USA)
Ortin, Francisco (University of Oviedo, Spain)
Safonov, Vladimir (St. Petersburg University, Russia)
Scholz, Bernhard (The University of Sydney, Australia)
Siegemund, Frank (European Microsoft Innovation Center, Germany)
Skala, Vaclav (University of West Bohemia, Czech Republic)
Srisa-an, Witawas (University of Nebraska-Lincoln, USA)
Sturm, Peter (University of Trier, Germany)
Sullivan, Kevin (University of Virginia, USA)
van den Brand, Mark (Technical University of Eindhoven, The Netherlands)
Veiga, Luis (INESC-ID, Portugal)
Watkins, Damien (Microsoft Research, U.K.)

REVIEWING BOARD

Alvarez, Dario (Spain)
Attardi, Giuseppe (Italy)
Baer, Philipp (Germany)
Bilicki, Vilmos (Hungary)
Bishop, Judith (South Africa)
Buckley, Alex (U.K.)
Burgstaller,Bernd (Australia)
Cisternino, Antonio (Italy)
Colombo, Diego (Italy)
Comito, Carmela (Italy)
Ertl, Anton,M. (Austria)
Faber, Peter (Germany)
Geihs, Kurt (Germany)
Gough, John (Australia)
Groesslinger, Armin (Germany)
Huisman, Marieke (France)
Knoop, Jens (Austria)
Kratz, Hans (Germany)
Kumar,C., Sujit (India)
Latour, Louis (USA)
Lewis, Brian (USA)
Meijer, Erik (USA)
Midkiff, Sam (USA)
Ortin, Francisco (Spain)
Palmisano, Ignazio (Italy)
Pearce, David (New Zealand)
Piessens, Frank (Belgium)
Safonov, Vladimir (Russia)
Schaefer, Stefans (Australia)
Scholz, Bernhard (Australia)
Schordan, Markus (Austria)
Siegmund, Frank (USA)
Srinkant, Y.N. (India)
Srisa-an, Witawas (USA)
Strein, Dennis (Germany)
Sturm, Peter (Germany)
Sullivan, Kevin (USA)
Tobies, Stephan (USA)
van den Brand, Mark (The Netherlands)
Vaswani, Kapil (India)
Veiga, Luis (Portugal)

Contents

Servicing Components with Connector Systems

Joachim H. Fröhlich
Software Engineering Department
Johannes Kepler University of Linz
Altenbergerstr. 69, A-4040 Linz, Austria
+43 70 2468 9432
joachim.froehlich@acm.org

Manuel Schwarzinger
Racon Software GmbH Linz
Goethestr. 80, A-4021 Linz, Austria
+43 70 6929 1732
schwarzinger@racon-linz.at

Abstract

Interfaces bind components at dedicated points. Usually, despite their central role, interfaces are packed either with functionality-implementing components (call interfaces) or with functionality-using components (callback interfaces). Components that reference other components in order to implement or to use interfaces are directly coupled. This kind of coupling affects component implementations: integration of component services leads to implementations that are dependent on the component container or to a multiplication of implementation efforts.

We propose connectors as a mechanism to completely decouple components from each other and from their underlying component container. Connectors are special-purpose components that isolate component interfaces. Connectors optionally provide services to communicating components, e.g., checking bidirectional communication protocols (operation call sequences and data flows), exchanging components during run time, and parallelizing or synchronizing service requests in a non-intrusive manner. This frees components to focus on their core business. Connectors foster the standardization of interfaces, accelerate the development of components, improve the testability, portability and maintainability of component-based programs, and hence promote component markets. .NET provides an almost ideal implementation basis.

Keywords

interfaces, connectors, components, configuration, software architecture

1. INTRODUCTION

Mainstream component systems facilitate component-based programming but do not enforce it. This can partly be ascribed to the sensible wish for downward compatibility with object-oriented programming techniques and a white-box reuse style. This holds for .NET as well as for the Java.

In practice, object-oriented programs are usually organized in complex class graphs. More often than not, class libraries and frameworks expose many details at unwieldy, complex interfaces that are intended to cover various broad application scopes. This negatively impacts component architectures when classes are blurred with components, as in .NET. A component-based architecture calls for a different programming style that employs black-box reuse, interfaces (types) and contracts. Component services (such as controlling access rights, monitoring/profiling, object pooling, controlling concurrent access, and controlling transactions) are attached to components via a mix of marker classes (such as System.ContextBoundObject and System.EnterpriseServices.ServicedComponent) and attributes (such as ObjectPoolingAttribute and SynchronizationAttribute, both defined in namespace System.EnterpriseServices). Thus component services are applied intrusively and serviced components are directly coupled to the component container. Implementation of component services along a message sink chain with call interception, program reflection and container-dependent base classes in a robust and efficient way proves a major challenge [Löw05]. Although not directly referencing constructs of the component container, clients that reference serviced components (classes) become dependent not only on these components but also on the underlying component container.

It is fundamentally clear that components should be designed with high cohesion and low coupling. This leads to advantages well-known from proper class and method design. Functional diversity unfolded at component interfaces as lengthy or deeply structured public classes packed into large assemblies complicate the application and implementation of components. The resulting problems are best documented by complicated test procedures – most evidently for

.NET Technologies 2006
Copyright UNION Agency – Science Press,
Plzen, Czech Republic.

components wired into intrusive application servers. These components are loaded with operations that are foreign to their core business. To overcome these difficulties, lightweight component containers with minimal impact on applications have been emerging. Spring [Har05] serves as a prototypical example in the Java world; although Spring achieves decoupling through interfaces interposed between beans (components), interfaces are not treated as independent contracts.

Interfaces connect communicating components (or classes) and thus should be independent pivotal elements. In practice, however, interfaces are attached either to service-providing components or to service-requiring components. This asymmetry impairs specification, development and testing of independently installable components; this, in the long run, hampers the wide adoption of component technology. To overcome this obstacle, we propose an architectural style where every pair of interacting components is fully separated with independent, special-purpose components that isolate component interfaces and optionally implement nonfunctional component services.

The paper is organized as follows: Section 2 details the goals of the proposed architectural style. Section 3 presents basic concepts of the connector/component architecture style. Section 4 sketches the application of connectors. Section 5 presents basic connector variants on which extended variants in Section 6 build. Related work and consequences conclude the paper.

We back the presentation with code snippets in .NET/C# and semantically rich system diagrams documenting real implementations by abstracting away unnecessary coding details rather than describing the design of prospective systems. The whole work is based on experience gathered with experimental implementations and with several variants of a generic program for analyzing data streams [Edl05], [Frö05], [Frö06].

2. GOALS

We seek an architectural style that enables components to focus on their business without being distracted by intrusive component containers. Such a style must enable economically feasible structuring of general-purpose programs as well as domain- or application-specific programs. Thereby a program is either self-contained or embedded in a component container (application server). The architectural style must facilitate separate specification, implementation, testing, guarding, installation, substitution and monitoring of components and their interactions. Component services must be transparent as far as possible. The architectural style must enable independent component evolution in in-house and open-market situations. For practicability, existing container technologies, if needed at all, should be supplemented rather than be replaced. The mechanisms enabling this architectural style must be configurable and thereby provide only as much flexibility and cost only as much in resources as needed in various stages of a project, such as development, test, launch or production stages.

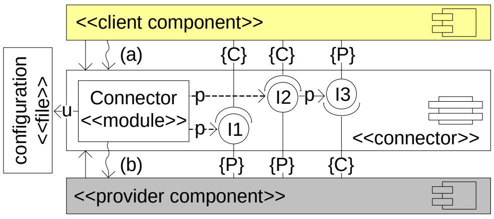
3. CONNECTOR BASICS

Interfaces rather than components carry software architectures. This contrasts with the usual view where software architectures focus on components and their interactions but tend to overlook the importance of component interfaces. We view software architectures as systems of component interfaces that service components. Like components, component interfaces are physical (i.e., binary) and identifiable concepts that we call *connectors*. Technically, a connector contains at least one interface in the sense of the programming language construct of the same name. All operations declared in interfaces of a connector form a functional closure; i.e., operations of connector interfaces use only parameters of basic data types, interfaces contained in the same connector or, in special cases, interfaces of neutral parts of .NET's framework class library, like System.Collection and System.Configuration. Logically, a connector specifies functional and nonfunctional properties of components using or implementing interfaces. Additionally, connectors may monitor, guard or change operation invocations and data transmissions across component boundaries as long as they conform to the contracted communication protocol without distracting adjacent components. Connectors do not execute any business- or application-specific functions.

Connectors define the points of variation at which components can be plugged in. At least two independent components communicate across the boundary that a connector establishes. We call them *functional components* (components for short where it is unambiguous) because they directly or indirectly implement functions that comprise the core business of a program. We speak of a *symmetric connector* when a functional component on the client side of connector uses the same interface(s) as the functional component on the provider side for communicating with each other. We speak of an *asymmetric connector* when a client component and a provider component use different interfaces and the connector maps interface concepts during communication. This article focuses on symmetric connectors.

Clutches serve as a metaphor for connectors. Clutches couple functional components, i.e., (driving) client components to (driven) provider components where these components might change their roles during communication. Thereby clutches transfer physical forces (data) in both directions, from clients to providers and vice versa. Real clutches optionally contain springs that dampen the transmission of exceptional forces. Connectors as defined above offer similar convenience. For example, they can log unspecified exceptions and map them onto exceptions specified in the connector because exceptions crossing component boundaries are part of the communication protocol. Another example is a connector that prohibits inadmissible input data or erroneous operation call sequences, e.g., faulty communication protocols.

Figure 1 illustrates a program that is minimal in terms of components and connectors.



{C}, {P}	Roles: Service-{C}lient, -{P}rovider
X ⟶ Y	X references Y at compile time
X ⇝ Y	X loads Y through (a) CLR, (b) connector
X -p→ Y	X provides Y
X -u→ Y	X uses Y
X (I) Y	X uses interface I implemented by Y

Figure 1. Connector and functional components

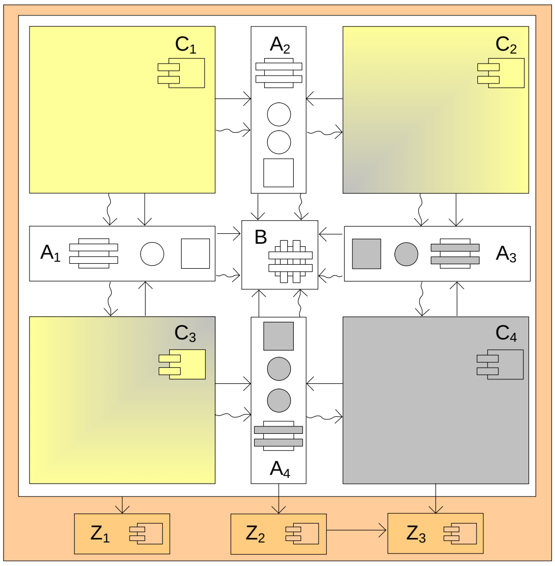
The connector in Figure 1 completely channels the communication between the sole service client and the sole service provider which includes the creation of service-providing objects. The connector module[1] processes data from the configuration file in order to relieve service clients as well as the connector itself from specifying concrete classes in the program code. The resulting constellation is characterized as follows:

- Components do not depend on each other.
- Components depend on connectors.
- Connectors do not depend on components.

The compilation procedure reflects this constellation:

```
csc /out:Connector.dll /t:library ...
csc /out:Provider.dll /t:library /r:Connector.dll ...
csc /out:Client.exe /t:exe /r:Connector.dll ...
```

Thus the architecture of a program can be modeled as a system of connectors that embed functional components (see Figure 2).

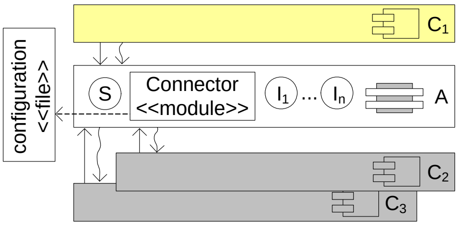


X ⟶ Y	X references Y at compile time
X ⇝ Y	X loads Y (if not done already)
□ ○ ●	Connect. class, interfaces: not serviced, serviced
A_i, B, C_i	Parts compliant to the connect./comp. style
Z_i	Parts not compliant to the connect./comp. style

Figure 2. A connector / component architecture

Figure 2 depicts components that follow the architectural style (A_i, B, C_i) and those that do not (Z_i). Functional components (C_i) are connected to a central connector manager (B). The connector manager provides for a communication interface by which external clients can monitor and control connectors (A_i). In order to control a connector, interfaces must be wrapped in proxy objects that pre- or post-process operation calls crossing component borders as indicated in Figure 2 for connectors A_3 and A_4. We call connectors *heavy connectors* if they wrap interfaces in order to transparently hook component services like logging, profiling, security checks and protocol checks. We call connectors *light connectors* if they contain only interface declarations. The run-time overhead of light connectors is negligible. Light connectors can be exchanged for type (interface) compatible heavy connectors just by program reconfiguration before run time.

Another type of connectors not sketched so far are multiple-part connectors (see Figure 3).



S	Service selection strategy
$I_1 \ldots I_n$	Service interfaces

Figure 3. Multiple-part connector

[1] Only classes with (static) class members are modules.

Upon exceeding a certain breadth, the functional interface of a connector (I_1 … I_n in Figure 3) can be implemented via several components (C_2 and C_3 in Figure 3) instead of just one component. These components build a group. A component group is defined by a common connector and one or more partitioning attributes. Each component of a group must publish a value for each partitioning attribute. The combination of attribute values characterizes a component within a component group. Thus partitioning attributes are used to diversify components. Diversification narrows the application scope of a single functional component, which eases its implementation while raising the domain-specific service level. A *multiple-part connector* is a connector that can bind (load) more than one interface-implementing component and uses a strategy (S in Figure 3) to choose a component whose attribute values best fit the client requirements. The strategy is provided either by the connector as part of the contract or by a client component. In contrast, *single-part connectors* bind (load) at most one interface-implementing component. Table 1 provides examples of multiple-part connectors and partitioning attributes.

Mp connector	Attributes
String matchers	automaton, e.g., NFA, DFA
Report generators	file format, e.g., PDF, HTML
Memory systems	access time, durability
Numeric systems	accuracy, precision, run time

Table 1. Some attributes of multiple-part connectors

4. A CONNECTOR IN TEST USE

Before delving into various extensions of connectors, let us examine a typical application scenario as seen from a service using (client) side. You can find the complete C# code of an almost identical implementation elsewhere [Frö06]. Figure 4 sketches the architecture of the program.

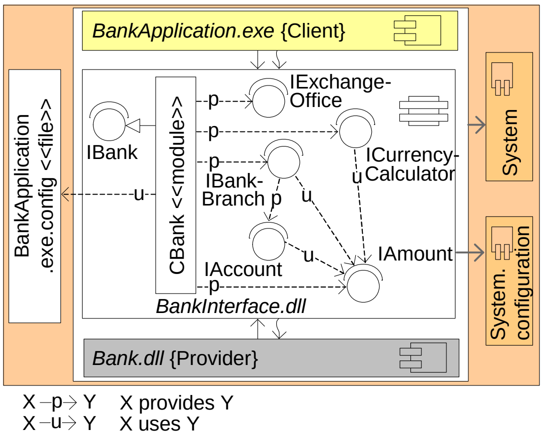


Figure 4. Architecture of a simplistic bankapp

The program implements a simplistic bank with several branches, accounts and customers. These concepts are directly reflected in the connector, whose interface operations build a functional closure, i.e., do only involve interfaces declared in this connector and basic data types:

```
namespace BankInterface { // Connector
 public interface IBank {
  void Provide(out IBankBranch branch);
  void Provide(out IAmount money, double val, string cy);
  …
 }
 public interface IBankBranch {
  IAccount SetupAccount(IAmount initialValue);
  IAccount SetupAccount(); // initialValue= 0.00 EUR
  bool Transfer(IAmount money,
   IAccount source, IAccount target);
  …
 }
 public interface IAccount {
  string Owner { get; set; }
  bool Deposit(IAmount money);
  …
 }
 …
}
```

The program applies a light, single-part connector (BankInterface.dll); i.e, the connector provides no services other then automatically loading one bank implementation (Bank.dll) at a time during first access by a bank client (BankApplication.exe). The concrete bank implementation is configured before run time, e.g., in the standard configuration file of a .NET application:

```
<configuration><appsettings>
 <add key="Provider" value="Bank.dll"/>
 …
</appsettings></configuration>
```

An application scenario taken from the client illustrates the coding style, which resembles that prevailing for clients of COM components. Several amounts of money are transferred from different source accounts to a common target account:

```
namespace BankApplication { // Client
 // Set up bank branch, target account
 IBank bank= CBank.Get();
 IBankBranch branch; bank.Provide(out branch);
 IAccount target= branch.SetupAccount(); // 0.00 Euro
 IAmount amount1, amount2, …;

 // Setup accounts
 bank.Provide(out amount1, 1000.00, "EUR");
 IAccount source1= branch.SetupAccount(amount1);
 bank.Provide(out amount2, 1500.00, "EUR");
 IAccount source2= branch.SetupAccount(amount2);
 …
 // Transfer money
 bank.Provide(out amount1, 500.00, "EUR");
```

```
  bank.Provide(out amount2, 800.00, "EUR");
  …
  branch.Transfer(amount1, source1, target);
  branch.Transfer(amount2, source2, target);
  …
}
```

The service provider (Bank.dll) can be exchanged without changing the client's implementation. For instance, a test stub that is applied during development and component test of a bank client can be replaced with a production version for integration tests. Moreover, the light connector can be replaced with a type (interface) compatible heavy connector. For example, from a technical point of view the heavy connector checks whether the client component passes to the provider component objects that the same provider has created before. From a business point of view this check is necessary, e.g., when a bank branch charges an account. The account must be set up by the same bank branch or by one of the other branches of the bank. We assume this integrity check to be necessary for every bank; hence it is part of the bank contract.

5. BASIC CONNECTORS

5.1 The Lightest Connector

An application scenario as simple as the sketched bank program is typical of tests of functional components. Although light connectors are by no means restricted to test scenarios, they obviously demand easy connector implementations. This directly leads to the question of how to design the lightest connector (see Figure 5).

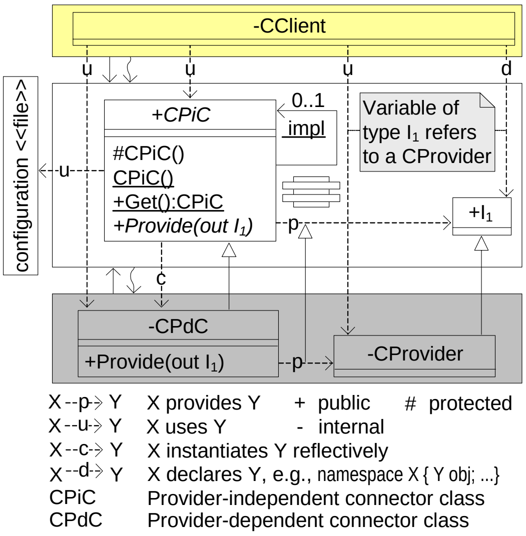

-CClient
u
u
u
d
configuration <<file>>
u
+CPiC
0..1
impl
Variable of type I1 refers to a CProvider
#CPiC()
CPiC()
+Get():CPiC
+Provide(out I1)
p
+I1
c
-CPdC
+Provide(out I1)
p
-CProvider
X --p-> Y X provides Y
X --u-> Y X uses Y
X --c-> Y X instantiates Y reflectively
X --d-> Y X declares Y, e.g., namespace X { Y obj; ...}
+ public
- internal
# protected
CPiC Provider-independent connector class
CPdC Provider-dependent connector class

Figure 5. The lightest connector

Besides combining logically coherent interfaces into a separate component, every connector must implement just one nonfunctional task: the establishment of the first connection between a service-using and a service-providing component while preserving their independence as well as its own independence. For this purpose a connector contains what we call a *provider-independent connector class* (CPiC in Figure 5). On the one hand this class is a module with (static) class methods and variables for loading and anchoring a provider; on the other side it is a type declaring factory methods [Gam95] for letting providers decide which objects to deliver as roots of business process chains (sessions). Thus each provider must subclass exactly one *provider-dependent connector class* (CPdC in Figure 5) per supported connector.

The provider-independent connector class uses reflection techniques to create the sole object of this class (a singleton [Gam95]), the connector object. This object is created automatically in the background during the first access to a provider (triggered by, e.g., bank= CBank.Get() in the bank application and executed by the class constructor of the provider-independent connector class) immediately after the provider component specified in the configuration file is loaded. Once the connector has supplied the connector object, a client queries it for the first business object by means of a factory method (via, e.g., bank.Provide(out IBankBranch) in the bank application) declared in the provider-independent connector class and implemented in the provider-dependent connector subclass.

The implementation of the managing stuff of a light connector is delightfully cheap. It costs about 10 lines of code executed only once per provider component upon first access (compare with the CPiC CBank in CBank.cs, directory Bank.src\BankInterface [Frö06]). All other operation calls across a light connector, i.e. across interfaces in the sense of the programming language, do only cost as much as invocations of instance function members [Hej04]. Thus light connectors completely separate communicating functional components with no run-time overhead.

5.2 The Lightest Heavy Connector

Heavy connectors factor out nonfunctional services from functional components. For this purpose, heavy connectors wrap interfaces in proxy classes [Gam95]. They provide hooks for affixing component services like profiling and protocol checks to both call interfaces and callback interfaces. Connectors wrap both interface types with the same procedure but at different moments: call interfaces on the way out of an interface function and callback interfaces on the way into an interface function. Figure 6 sketches the structure of a heavy connector.

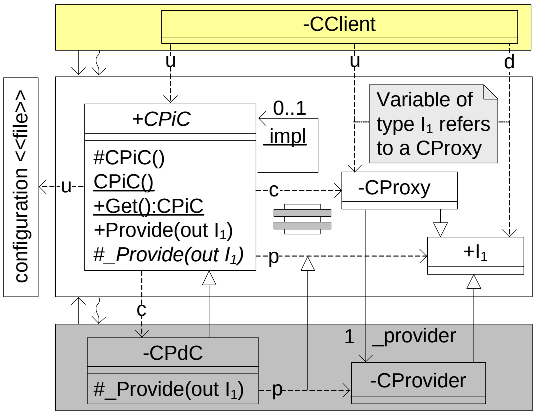


Figure 6. The lightest heavy connector

The decisive difference compared to a light connector is that all function calls of a client component first activate a wrapping function implemented in the heavy connector before they activate a function in a provider component. The prerequisite for wrapping all operation calls crossing a connector is template methods [Gam95] in the provider-independent connector class, as the following code excerpt demonstrates by wrapping the root object of a business process chain (implementing a call interface):

```
namespace Connector {
 public abstract class CPiC { // connector module
  public void Provide(out I1 p) { // the template method
   I1 provider; // the service provider
   this._Provide(out provider);
   p= new CProxy(provider); // wrap call interfaces on the
    // way out from a provider to a client
  }
  protected abstract void _Provide(out I1 provider);
   // the primitive operation of the template method
  ...
 }
 public interface I1 { ... }
 internal class CProxy : I1 {
  internal CProxy(I1 provider) { this._provider= provider; }
  ... // methods wrapping I1 functions
  private I1 _provider; // the wrapped service provider
 }
 ...
}
```

Syntactically, proxy objects and connected component services are completely hidden in the connector and therefore invisible to functional components.

5.3 The Lightest Multiple-Part Connector

Multiple-part connectors allow the differentiation and installation of several provider components that offer alternative or variant services. Moreover, heavy multiple-part connectors enable a different class of component services, like multiplexing (or parallelizing) of service request among several provider components and graceful failover from one service provider to another. Figure 7 sketches the structure of a light multiple-part connector.

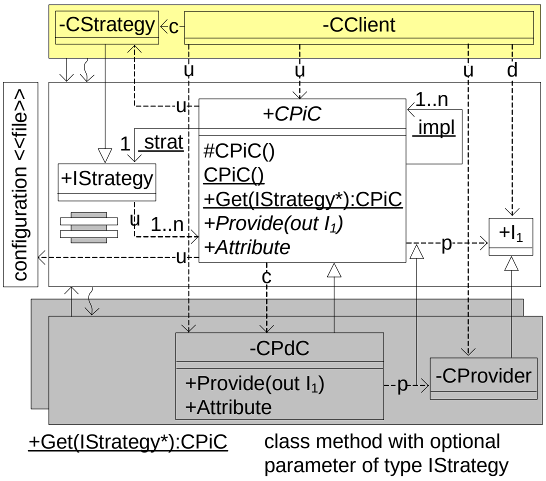


Figure 7. The lightest multiple-part connector

Provider components to hook into a multiple-part connector are specified in the configuration file with multiple-value entries such as

```
<configuration><appsettings>
 <add key="Provider" value="Bank1.dll;Bank2.dll"/>[2]
 ...
</appsettings></configuration>
```

All these provider components share one (structured) interface, i.e., one connector, and usually vary in their implementation with regard to at least one component attribute. A multiple-part connector offers clients the chance to dynamically select one of the configured providers. To make this work, the provider-independent connector class forces provider-dependent subclasses to return values that characterize their business with regard to a differentiating business attribute. The strategy pattern [Gam95] lends itself for a flexible implementation of the selection algorithm. In the context of the bank example, clients can now choose among several banks applying different interest and portfolio strategies.

6. EXTENDED CONNECTORS

Connectors can be extended at four sides (see Figure 8):

(a) Client side: several functional components use one connector.

(b) Provider side: several functional components provide alternative or supplementing services.

[2] Of course, the type-safe way for specifying an arbitrary number of provider components would be an xsd:element with a multiplicity range of minOccurs="1" maxOccurs="unbounded".

(c) Connector service side: several special-purpose components register component services for communicating functional components.

(d) Connector managing side: the behavior of a running program is monitored and controlled in terms of connectors and components.

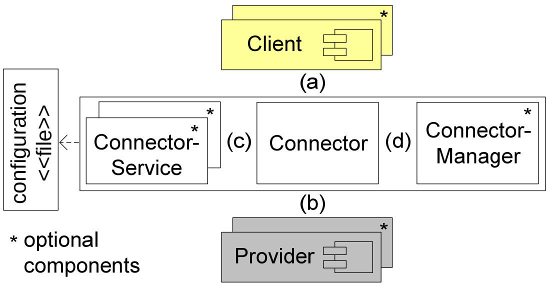


Figure 8. Connector extensions

6.1 Extending the Connector Service Side

Component services can be implemented in proxy classes directly in a heavy connector (see 6.1.1) or sourced out into separate classes in separate components (see 6.1.2).

6.1.1 Implementing Services Directly

Component services can be implemented with minimal effort directly in a connector. This implementation style well suits special component services like checks of highly specialized communication protocols while obviously compromising reusability of rather general applicable component services like logging[3]. To give an impression of a component service, we sketch a part of the life cycle management. The heavy connector checks objects that client components pass as operation parameters to a provider component for creation by the same provider.[4] Again, we demonstrate this for the sample bank application introduced in Section 4: A bank can only service its own bank accounts. We assume that this constraint is part of the contract holding for all banks. If this is true, then the connector is the place to implement the constraint. On violation of this constraint the connector throws a protocol exception:

```
namespace BankInterface { // Connector
 internal class CBBProxy  // Class Bank Branch Proxy
   : IBankBranch {
  public IAccount SetupAccount() { // public protocol
   IAccount provider= this._provider.SetupAccount();
   CAProxy accountProxy= new CAProxy(provider, this);
   this._issuedObjs.Add(accountProxy);
  }
  ... // more methods wrapping IBankBranch operations
  internal CBBProxy(IBankBranch bankBranch) {
   this._provider= bankBranch;
  }
  internal bool HasIssued(CAProxy proxy) {
   return this._ issuedObjs.Contains(proxy);
  }
  private IBankBranch _provider;
  private Utilities.ISet _issuedObjs= new Utilities.CSet();
 }
 internal class CAProxy // Class Account Proxy
   : IAccount {
  public void Withdraw(IAmount money) { // public protocol
   if (!this._creator.HasIssued(this))
    throw new CProtocolException("unknown account");
   this._provider.Withdraw(money);
  }
  ... // more methods wrapping IAccount operations
  internal CAProxy(IAccount provider, CBBProxy creator) {
   this._provider= provider;
   this._creator= creator;
  }
  private IAccount _provider;
  private CBBProxy _creator;
 }
 ...
}
```

6.1.2 Implementing Services Indirectly

Proxies that delegate requests for component services lead to service implementations that are extensible and reusable in the context of several connectors. Such proxies signal changes of relevant program states (method calls and returns across component boundaries) and delegate the provision of services to observers [Gam95] implemented in separate components. This raises the question of the sequence in which component service should be applied.

In general, component services can be applied in any sequence because they have no side effects. From a practical point of view, of course, it is useful to check, e.g., whether a client is allowed to use a provider before checking the communication protocol in case the two component services are implemented separately. Likewise, the communication protocol should be checked before a client is allowed to ask for exclusive usage of a provider. Thus ideally component services, their order of application and the associated connectors are specified in a program configuration file and set up with reflective programming techniques.

[3] During development and test phases of the generic data stream analyzer (mentioned in the Introduction) a heavy connector tests the communication protocol between the component providing the business logic (data stream pattern matcher) and various user interface components [Frö05]. The connector applies the state pattern [Gam05].

[4] This service is indeed rather generally applicable. It checks an integrity constraint for components that cast types of parameter objects to component-specific type implementations (classes).

6.2 Extending the Connector Managing Side

All connectors of a program may be connected at a central point which we call the connector manager. The connector manager is the place for querying and changing the state of a program in terms of components and connectors either from inside the program through API calls or from outside the program, e.g., through a web service. In particular the connector manager enables

- loading of functional components
- unloading of functional components
- switching component services on or off
- querying for current components and connectors
- querying for interaction states and histories
- coordinating several connectors

From a technical point of view, the most interesting feature of a connector manager is the coordination of connectors with regard to unloading stateful functional components. This requires life-cycle management of components.[5] A connector attached to a stateful functional component must check the communication protocols for each usage scenario (per business process chain) and indicate the functional component as being in a state allowing the component to be unloaded, as it is usually in initial states, end states, or error states (0-states for short). This requirement holds for all connectors directly attached to a component as well as for all dependent connectors.[6] Consider the program sketched in Figure 9.

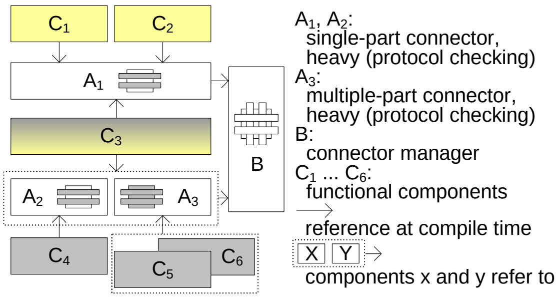


Figure 9. Managing connector systems

Provided that it is useful to unload C_3, all dependent connectors (A_1, A_2 and A_3) have to confirm dependent functional components to be in 0-states. These connectors contain at least 4 state machines that check the communication protocols between components

- C_1/C_2 and C_3,
- C_3 and C_4,
- C_3 and C_5, and
- C_3 and C_6.

Note that thereby we assume C_1 and C_2 to take part in a common usage scenario (session); i.e., they share one business process chain and therefore one protocol-checking state machine. Inversely, one component could take part in several usage scenarios of a connector so that, e.g., two or more state machines could be active in A_2 checking two or more applications of C_4 by C_3. Technically, unloading a component requires it to be installed in separate application domain (System.AppDomain) [Gun02], i.e., in a separate .NET process, which of course increases communication costs due to marshalling all calls between application domains.

Besides, the connector manager factors out code common to all connectors, such as that for loading and unloading functional components and standard component services such as logging operation call sequences.

7. RELATED WORK

This article focuses on physically separate connectors as a means to connect and at the same time to decouple components in the context of coherent programs or program parts.

Some of the presented concepts suggest concepts prevailing in the context of distributed programs. Here connectors are manifested as parts of the underlying infrastructure, e.g., in the form of networking protocols, pipes, SQL links between a database server and a database application program, event buses, and message brokers [Clem03], [Meh00], [Sha96]. Service–oriented architectures (SOA) provide the plumbing for the integration of components running on different technological foundations [Sko05]. Component interfaces are published, queried and translated into executable code for calling services across the Internet.

Connectors as separate compilation and deployment units of coherent programs are scarcely discussed elsewhere. In a coherent program, connectors usually occur at the abstraction level of a programming language as shared variables, buffers and procedure calls [Meh00], [Sha96]. This strongly contrasts with connectors at the architectural level of a program as discussed in this paper. At the architectural level a connector must not to be confused with a façade [Gam95] or a mediator [Gam95]. A façade provides a unified interface to a set of interfaces in a sub-

[5] Strictly speaking, only component instances can have state in a running program because components are just binary deployment units. As this should be clear from the context, we speak of stateful components.

[6] A functional component that implements (interfaces declared in) several connectors might indicate low binding or hint incomplete connector interfaces and so disobey the requirement for functional closure.

system. This is usually done when a class applies business logic to orchestrate (instances of) other classes. Thus a façade is coupled to the covered subsystem. A mediator coordinates interactions of a group of objects. Thus a mediator executes application-specific functions. A connector with its distinct orientation on improving nonfunctional system properties, such as reliability, adaptability, and testability, is independent of adjacent functional components and does not execute any application-specific functions.

However, the idea of including related interfaces in separate components is not new. Szyperski et al. [Szy02] emphasize the importance of viewing interfaces in isolation from any specific component that might implement or use such interfaces. Further-reaching concepts or implementation techniques are not discussed. In the context of .NET, Löwy [Löw05] suggests assemblies with interfaces to parallelize the development of adjacent components. Wienholt [Wie03] proposes a similar technique to shorten load time of assemblies and to save memory. He puts frequently and occasionally used types of an assembly into different netmodules[7] and separates them by netmodules that consist only of interfaces, which leads to multiple-module assemblies. This can also be achieved with the connector/component architectural style.

Interfaces play an important role in the realm of lightweight component containers; Spring [Har05] is a good example. Spring decouples components (beans) in the form of classes by externalizing the creation of instances of collaborating classes and injecting them at dedicated points of the class to be configured (dependency injection). Collaborating classes are expected to implement well-defined interfaces. Although the work on connectors presented in this article shares many of the goals of Spring, such as isolated component tests, externalization of component dependencies (in configuration files), and design in terms of the application domain (rather than in terms of the implementation domain or a mix of both), the solutions move in different directions. Spring abandons subclassing for Spring-conform components due to reflective programming techniques. In contrast, a functional component in the role of a service provider has to implement a provider-dependent subclass per connector, even though this subclass contains only domain-specific methods (in a special syntax). Spring does not support the transparent injection of non-functional services between communicating components. Spring has no notion of multiple-part components and provides no special means for coordinating semantic operations attached to related interfaces either in the form of protocol checking services or in the form of a connector manager for monitoring and controlling running programs.

8. SUMMARY AND CONSEQUENCES

Connectors as discussed in this article are special purpose components that embody boundaries of functional components in the form of binary contracts. This allows functional components to focus on their core business. Moreover, functional components can be

- developed in several alternate or supplementary variants
- specified and tested separately
- relieved of intermingled nonfunctional services like logging, caching and checking communication protocols
- dynamically monitored and controlled if a connector manager supervises the connector system

Connectors may interpose nonfunctional services between functional components in a completely non-intrusive manner. This is achieved by means of a pattern language [Cun87] that combines several design patterns [Gam95], such as Factory Method, Template Method, Proxy, Strategy, State and Observer, and by encapsulating these patterns in special components (connectors). Classes of functional components shed any special base types (such as System.ContextBoundObject) or attributes (System.Attribute) for profiting from component services. Certainly these techniques can be used for implementing component services within connectors. Proxy classes in connectors expose suitable method call joint points to implement component services as aspects in the sense of AOP (aspect-oriented programming). Services that have well-defined effects on particular operations support the use of AOP [Mur01]. This is the case, e.g., for synchronization and accounting services but not for checks of complex, application-specific communication protocols. Due to the localization of services in connectors, functional components remain unchanged regardless of how services are intercepted, such as with context bound objects, code generation, modification of IL (intermediate language) code or .NET's profiling API.

If a program does not depend on a nonfunctional service, a heavy connector can simply be replaced with an interface-compatible light connector without changing the implementation of adjacent components. The implementation of the skeletal structure of a light connector is almost for free with regard to both development time and run-time efficiency while

[7] A netmodule is a *raw* module that must be associated with a full-fledged component (assembly) prior to deployment.

still providing the fundamental advantages of connectors, i.e., separate specification, testing and development of functional components. The call of an operation across a light connector costs only as much as a call of an instance function member. The one-time loading of a component immediately before the first operation call does not impair performance in the long run. Even heavy connectors can boost the overall performance of a program. For instance, checks of communication protocols (pre- and post-conditions, invariants, operation call sequences) at clear-cut, contracted and rather stable component boundaries concentrate on essential and coherent system parts (components) while abstaining from checks of rather quickly changing implementation-specific (i.e. component-specific) objects scattered around the program.

Even demanding services like parallelizing service requests in a blocking or non-blocking manner among several service-providing components can be included in a heavy, multiple-part connector without distracting adjacent components. However, this holds only for unidirectional data flow where service clients just trigger service providers concurrently without needing any calculated value from them. Bi-directional data flow demands connectors that buffer data returned by providers and a special interface enabling clients to fetch this data for each provider. This exceeds the capabilities of symmetric connectors and moves towards asymmetric connectors that map deviating client and provider languages in terms of deviating interfaces.

In any case, separate connectors in different extension stages supply effective, non-intrusive mechanisms to solve challenges and issues in developing, testing and quality assurance of software components. Both isolated connectors and connector systems promote architecture-centric development of programs with variants. Connectors lend themselves for gluing common components and varying components with predictable capabilities even in order to build high-quality product families (product lines) [Wei99]. At the same time, connectors raise the productivity of component developers, testers and architects. Variants of a generic data stream analyzer [Frö05] and several experiments prove the practical feasibility of the connector/component architecture style. Coordinated life-cycle management (protocol checking) of several components is a key issue of further work.

9. REFERENCES

[Clem03] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., and Stafford J.: Documenting Software Architectures – Views and Beyond. Addison-Wesley, 2003

[Cun87] Cunnigham, W.: Design Methodology for Object-Oriented Programming. OOPSLA'87, ACM SIGPLAN Notices 23 (5), 1987

[Edl05] Edlmayr, J., Fröhlich, J.H., Schwarzinger, M., and Stranzinger T.: Components for All Cases. (in German) SIGS Datacom OBJEKT-spektrum 1/2005 (part 1), 2/2005 (part 2)

[Frö05] Fröhlich, J.H., and Schwarzinger, M.: Treating Interfaces as Components. In IVNET’05 ISBN 972-8688-31-8, 2005

[Frö06] Fröhlich, J.H., and Wolfinger, R.: .NET Profiling: Write Profilers with Ease Using High-Level Wrapper Classes. MSDN Magazine 21 (5), 2006

[Gam95] Gamma, E., Helm, R., Johnson, R., and Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. 1995

[Gun02] Gunnerson, E.: AppDomains and Dynamic Loading. http://msdn.microsoft.com/library/en-us/dncscol/html/csharp05162002.asp, 2002

[Har05] Harrop, R., and Machacek, J.: Pro Spring. Apress, 2005

[Hej04] Hejlsberg, A., Wiltamuth, S., Golde, P.: The C# Programming Language. Addison-Wesley, 2004

[Löw05] Löwy, J.: Programming .NET Components. O'Reilly, 2005

[Meh00] Mehta, M.R., Medvidovic, N., and Phadke S.: Towards a Taxonomy of Connectors. ICSE’00, conf.proc., Limerick Ireland, 2000

[Mur01] Murphy, G.C., Walker, R.J., Baniassad, E.L.A., Robillard, M.P., Lai, A., and Kersten, M.A.: Does Aspect-Oriented Programming Work? CACM 44 (10), 2001

[Sha96] Shaw, M., and Garlan, D.: Software Architecture-Perspectives on an Emerging Discipline. Prentice-Hall, 1996

[Sko05] Skonnard, A.: SOA: More Integration, Less Renovation. MSDN Magazine 20 (2), 2005

[Szy02] Szyperski, C., Gruntz, D., and Murer, S.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley, 2002

[Wei99] Weiss, D.M., and Lai, C.T.R.: Software Product Line Engineering. Addison-Wesley, 1999

[Wie03] Wienholt, N.: Maximizing .NET Performance. Apress, 2003

Transparent Mobility of Distributed Objects using .NET

Cristóbal Costa, Nour Ali, Carlos Millán, José Ángel Carsí
Department of Information Systems and Computation
Polytechnic University of Valencia
Camino de Vera, s/n
46022, Valencia, Spain
{ccosta, nourali, cmillan, pcarsi}@dsic.upv.es

ABSTRACT

Nowadays, information systems are becoming more distributed and dynamic in nature, where mobility is a solution for run-time adaptability. However, implementing software with such characteristics is a complex task. This is due to the fact that current middleware technologies do not provide a simple and direct way of implementing distributed objects that can move in a transparent way. In this paper, we are going to present an approach, implemented in .NET Remoting to allow transparent mobility of distributed objects. Our approach is based on separating the distribution and mobility concerns from the source code that contains the application logic in entities called attachments. Thus, attachments are high-level proxies that are responsible for creating communication channels and are capable of managing dynamic location changes without affecting the objects in the case of mobility. This approach has been implemented using a case study. The response time of distributed communication provided by our approach has been tested and compared with the remote communication provided by the primitives of .NET Remoting.

Keywords
Distributed communication, transparent mobility, autonomous mobility, .NET Remoting

1. INTRODUCTION

Currently, distributed systems are built by using middleware services [Ber96a]. The main idea behind middleware is to allow components at different hosts to collaborate in such a way that users perceive the system to be centralized. Information systems are becoming more dynamic at run-time where mobility plays an important role for adapting applications and solving problems such as fault tolerance and load balancing.

However, building mobile and distributed systems is not a simple task. The middleware technologies that are currently available do not provide the sufficient primitives that allow the deployment of distributed components which have a mobile nature at run-time. For example, one of the steps for implementing mobile objects in .NET is serializing the object states using the serializable attribute. However, an object that must be accessible remotely in .NET Remoting cannot be serializable at the same time [Obe02a]. Therefore, .NET Remoting does not allow the direct implementation remote objects mobility. Another drawback found in .NET Remoting is that to implement remote objects, the class must inherit from the *MarshalByRef* class. This limits the inheritance flexibility of remote objects because they cannot inherit from other classes as .NET does not offer multiple inheritance.

In this paper, we are going to present an approach for supporting distributed communication and mobility tolerance in a transparent way for .NET objects. The implementation of this approach is based on a concept called attachments offered by the PRISMA approach. PRISMA is an aspect-oriented component-based approach where attachments allow the transparent communication among components. In order to support the PRISMA approach, a PRISMANET [Per05a] middleware has been implemented. Based on the experience gained from this approach, we noticed that the attachment functionality could be extended to support

.NET Technologies 2006
Copyright UNION Agency – Science Press,
Plzen, Czech Republic.

transparent distributed communication and mobility for objects. Thus, the implementation presented in this paper can adapt object-oriented applications that were not initially designed to be distributed and mobile in order to obtain this functionality.

Our approach is based on separating the distribution and mobile concerns from the source code (which contains the application logic) in entities called attachments. Thus, the attachments are high-level proxies that are responsible for creating communication channels and are capable of managing dynamic location changes without affecting the objects in the case of mobility.

The structure of the paper is as follows: Section 2 presents some works that offer transparent distributed communication and mobility of objects. Section 3, explains the attachments concept and the implementation of our approach by using a case study of distributed mobile agents. Section 4, evaluates the communication costs introduced by our approach compared with the .NET Remoting framework. Finally, conclusions are presented in Section 5.

2. RELATED WORKS

The work in this paper is focused on providing an approach that allows objects to be accessible remotely and to be moved from one location to another during run-time.

Mobility is classified by Picco [Fug98a] into weak and strong mobility. Weak mobility involves the migration of the code and data of an object. In weak mobility, before interrupting the object for migration, the developer has to make sure that the object's threads have finalized their tasks. Strong mobility involves the migration of the code and the execution state (stack, program counter …). In strong mobility, mobile object execution is only interrupted for migration. Once the object has been migrated to its destination, it continues to execute from the interrupted point. However, strong mobility is difficult to implement as it greatly depends on the .NET CLR internals. In order to interrupt a thread in a transparent way, and to be able to restore it in a new destination, the following actions must be performed. On the one hand, we must be able to obtain the instruction pointer and the execution context of the threads to be moved. On the other hand, we must also be able to restore a thread from an instruction pointer and its thread context. In other words, to implement strong mobility, we must be able to serialize threads, which is not currently available in .NET. For these reasons, our approach is designed to provide weak mobility and not strong mobility.

Approaches that deal with communication transparency have been dealt mostly in Java. The work in [Hic99a] provides a run-time system and a compiler that generates remote references. This work requires having a process on each physical machine. Each of these processes has: a set of caches that maps object IDs to instances, a cache for the local instances, and a cache for each remote process filled with the instance references that are needed locally. A drawback of this approach is that the programmer must indicate where an instance is created, since the objects are always allocated in the same process (physical machine), and there is no way to change the references in the caches.

MobJeX [Rya04a] is a Java-based application framework that allows weak mobility as well as remote accessibility of objects. This is obtained by precompiling the mobile objects in order to generate two interfaces: a remote interface and a local interface. Two classes are also generated: a proxy class which provides a client with the reference to the server, and a serializable class which represents the original class that implements the two interfaces. In our approach, no precompilation is necessary; however, all mobile objects should be serializable classes. Another difference between MobJeX and our approach is that the mobility requests in MobJeX cannot be caused by the same object; they must be caused by a system controller. This eliminates the possibility for mobile objects to be autonomous. Also, if a MobJeX server object is moved a chain of calls is produced in order to find out its new location since the proxy object is not notified of the change directly. However, in our approach, the proxy is updated directly to the new location of the object. Another limitation in MobJeX is that it does not support the declaration of static methods in mobile objects. This is because it only supports interfaces to be shared between the client and the server. In our approach, it is up to the developer to choose between shared interfaces or classes.

Another approach that deals with mobility in Java is the Active Container approach [Cha03a]. This approach provides a compiler that dynamically generates the code for storing objects in containers. The communication among objects is made transparent by calling the active container. However, the mechanism of changing the proxy when the server moves is not described. To move an object, it is also necessary to indicate both the active container of the stored object and the new active container. This reduces mobility transparency and does not allow objects to self-initiate mobility.

One of the few works performed in the context of .NET is [Tro03a]. It provides weak mobility as our approach. It uses Aspect-Oriented Programming (AOP) to separate the mobility decisions from the objects code in order to allow objects to self-initiate the mobility decisions. Location changes that are caused by mobility are transparent to objects because a module is provided that forwards requests to find out object locations. In our approach, no forwarding requests are needed since the location references are dynamically updated. It is also important to comment that our approach can also use AOP. Thus, the PRISMANET middleware [Per05a] supports mobility and distribution of aspect-oriented components. However, since AOP is not standardized in the .NET framework [Per05a] the work presented in this paper does not use AOP.

3. AN OBJECT-ORIENTED APPROACH FOR TRANSPARENT COMMUNICATION AND MOBILITY

In the following, we present an overview of the background on which our approach is based. We then explain our approach using a case study of mobile agents.

Attachment Overview

3.1.1 The Attachments in PRISMA

PRISMA [Ali05a] is an approach that allows the construction of complex, reusable, dynamic, and distributed architectures by interconnecting architectural elements. Thus, an architectural element must only request and receive petitions through ports of an interface. However, an architectural element instance is unaware of with whom it is interacting, and how the interaction is being performed. This allows the architectural elements to communicate in a transparent way thanks to the attachment functionality.

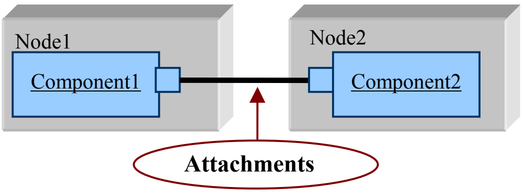


Figure 1 Attachments in a distributed software architecture

Attachments (see Figure 1) are the artefacts that are responsible for the connections among the ports of the architectural elements instances. This way, the attachments can connect architectural elements whether they are distributed or not. In addition, if an instance moves, it is the attachments that change the references and not the architectural element. The PRISMA approach has been implemented using .NET through the PRISMANET middleware [Per05a]. In order to offer the attachment functionality not only to a component based approach but also to object-oriented approaches, the attachments implementation in [Per05a] has been adapted to provide a middleware to connect mobile .NET objects.

3.1.2 Design of the Attachment Approach for .NET Objects

Our middleware permits client objects and server objects to communicate locally or remotely in a transparent way. In addition, the client and server objects can be mobile. Therefore, these objects must be serializable.

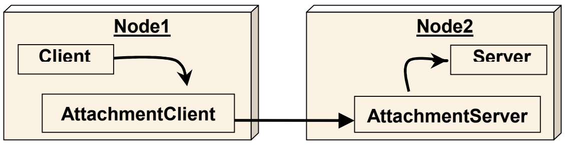


Figure 2 Attachment structure

Figure 2 shows the design of a communication between a client object and a server object in the attachment approach. The communication transparency is performed because each client has a reference to an *AttachmentClient* instance. An *AttachmentClient* instance is always local to the client object. The responsibility of the *AttachmentClient* is to redirect the client's requests to an *AttachmentServer* object. The *AttachmentServer* object is always local to a server. Therefore, depending on whether the server object is local or remote to the client the *AttachmentClient* object may or may not make a remote call. Therefore, for the cases where the server must be accessible remotely, the *AttachmentServer* class inherits from *MarshalByRef* class, as is specified by .NET Remoting technology.

In this approach, the client object always sends requests locally to the *AttachmentClient* object and does not have to take into account the location of the server. Thus, if the server object moves, it is the *AttachmentClient* that must change its references. In addition, as the *AttachmentServer* object is of *MarshalByRef* type the server object does not have to publish its services by .NET Remoting. This solves the problem that objects cannot be both serializable and *MarshalByRef*.

Distributed mobile agent case study

In order to explain the application of this approach, we present a case study of distributed mobile agents. The case study lies in a system composed of several distributed databases, of which we need to collect information. Mobile agents are sent to the databases in order to perform local searches, and then they return to their source host to process the search results.

In many situations the search might have to be done in large and complex systems, such as the Internet, where it is appropiate to use as many agents as sites to search. For this reason, we decided to use a small number of agents that perform the search. These agents are distributed dynamically among databases depending on their search results. The first agent to finish its work moves to the next database and notifies the other agents so that they do not process the same database twice. This solution requires each agent to be capable of moving in an autonomous way and also to be connected with the other distributed agents in order to share services and information.

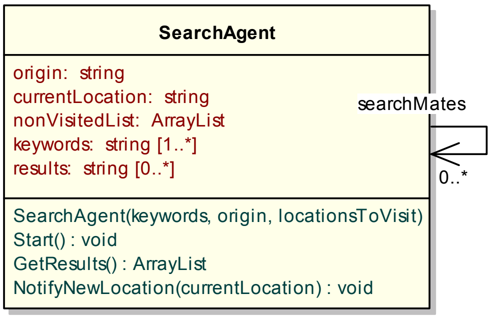


Figure 3 SearchAgent class

The SearchAgent class is defined in Figure 3. Each agent requires a list of keywords for the search, its host origin, and the initial database list where the search is to be performed. The *Start()* method is invoked to search in a current database. After an agent finishes its search, it needs to move to the next unvisited database. Then, it notifies the other agents of its new location by invoking the *NotifyNew Location()* method. It is important to note that each agent could be in a different location each time. Finally, when there are no more locations to visit, each agent returns to its host origin and all the collected data is processed.

Applying the Attachment Approach to .NET Remoting

Our approach provides a lightweight middleware to build distributed applications with the following features:

- Objects can move autonomously among computers without having to take into account how distributed communications with other objects are performed.
- Objects use the middleware to:
    - Register themselves in order to offer their services to other objects,
    - Request the creation of a connection to objects to use their services,
    - Ask for mobility when they need it.
- There is no need for a centralized infrastructure to manage these mobility and object registration services. The infrastructure has been designed in a decentralized way.
- Neither client nor server objects need to precompile code as in other approaches, because reflection and code generation is used.

The communication infrastructure is built on .NET Remoting in a transparent way. The additional communication cost introduced between two objects depends on the network traffic and the derived costs of invocation methods through delegates.

However, this approach requires a few constraints:

- Every computer must run this middleware in order to use mobility and object-registration services.
- A client object needs to know where the server object is located when it establishes the connection. However, location-awareness is provided since connection is established.
- Due to the fact that the middleware provides weak mobility implementation, objects must take care of their threads before moving. When the object is restored in the new location, an initialization method can be provided to initialize new threads at a specific point.
- In order to support the mobility of the object state, both client and server object classes must be marked as *Serializable*.

In the following sections we explain the implementation of our approach using the case study presented in the previous subsection.

3.2.1 The AttachmentManager class

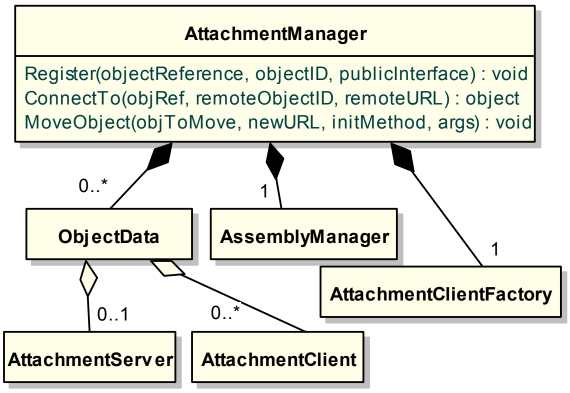

AttachmentManager
Register(objectReference, objectID, publicInterface) : void
ConnectTo(objRef, remoteObjectID, remoteURL) : object
MoveObject(objToMove, newURL, initMethod, args) : void
0..*
1
1
ObjectData
AssemblyManager
AttachmentClientFactory
0..1
0..*
AttachmentServer
AttachmentClient

Figure 4 AttachmentManager class

The *Attachment Manager* class (see Figure 4), is the main class of our middleware, and must be running on each computer in order to offer the following services:

- server-behaviour registration services,
- client-behaviour connection services,
- mobility services

- transference of required assemblies when mobility takes place
- dynamic generation of server proxies on demand

For each object that uses the attachment concept, the *AttachmentManager* maintains an *ObjectData* structure that contains information about the attachments that are used. On the one hand, if an object provides services to other objects (that is, it acts with server behaviour), it will have an *AttachmentServer* associated to it. On the other hand, if an object requires services from other objects (that is, it acts with client behaviour), it will have an *AttachmentClient* associated to it.

In our case study, a SearchAgent object has both client behaviour and server behaviour. On the one hand, it needs to notify its new location when it arrives to a new site; i.e. it invokes *NotifyNew Location()* method of other SearchAgents. On the other hand, it must be notified about sites being visited by other SearchAgents; i.e. it provides the *NotifyNewLocation()* method to be invoked remotely.

3.2.2 Server behaviour

A *SearchAgent* object (from now on, the Server object) invokes the *Register()* service of the *AttachmentManager* class in order to be accessible remotely. The following parameters are needed:

- *object reference*: reference of server object, which will be used to create the *AttachmentServer* part.
- *objectID*: custom ID to uniquely identify a server object. This must be known by each client object in order to establish a proper connection.
- *publicInterface*: an optional parameter that allows us to restrict services that would be offered to clients. Otherwise, all services from the server object are provided.

As a result of this invocation, an *AttachmentServer* object is created and made accessible remotely (see Figure 5). This object represents the SearchAgent object and is responsible for offering the following services:

- incoming request services are forwarded towards the server object.
- mobility notification of the server object to client objects that are connected to it.

The *AttachmentServer* is composed by the *AttachmentServerMediator* class, who publishes the services that can be invoked remotely and is responsible for invoking Server methods.

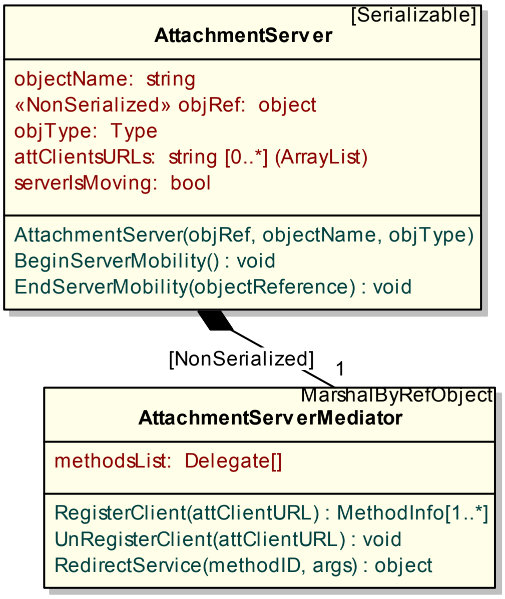

[Serializable]
AttachmentServer
objectName: string
«NonSerialized» objRef: object
objType: Type
attClientsURLs: string [0..*] (ArrayList)
serverIsMoving: bool
AttachmentServer(objRef, objectName, objType)
BeginServerMobility() : void
EndServerMobility(objectReference) : void
[NonSerialized]
1
MarshalByRefObject
AttachmentServerMediator
methodsList: Delegate[]
RegisterClient(attClientURL) : MethodInfo[1..*]
UnRegisterClient(attClientURL) : void
RedirectService(methodID, args) : object

Figure 5 AttachmentServer class

Due to the fact that the method signatures of the Server are not known until runtime, direct call invocation cannot be used. We had to use dynamic method invocation. We decided not to do this through reflection (using Type.InvokeMember()) because it has the worst performance [Gunn04a]. Instead of this, we have used dynamic code calling through Delegates. When *AttachmentServer* is created, a delegate is created for each method provided by the server, following these steps:

1. Method information is obtained by means of reflection at runtime. With this information, a delegate type is created by emitting its MSIL code.
2. This delegate type is instanced and stored in an array.
3. The index of the array where the delegate is stored is used to uniquely identify the method to be executed. We have called it *MethodID*. This index is stored together with related method information in a structure called *MethodInfo*.

Thus, clients forward methods by the invocation of the *RedirectService()* method and by providing the correct *MethodID* of the delegate to be executed. We chose this alternative in order to avoid searches in the delegate list, which can slow method invocation. Clients get all the *MethodID*s and their related information (*MethodInfo* list) when they subscribe to the *AttachmentServer* through the *RegisterClient()* method. Moreover, client subscription to the *AttachmentServer* provides a way to be notified when the server is moving.

3.2.3 *Client behaviour*

A SearchAgent (the client) that wants to call methods from another object (remote or local) needs the reference of this object to do that. This reference is provided by the *ConnectTo()* service of the *AttachmentManager* class. The *objectID* and its current location must be provided in order to get its reference. The reference provided is in fact an *AttachmentClient* that acts as a proxy. From now on, the client object will not have to take care of distributed communications nor location changes of the remote object (the server).

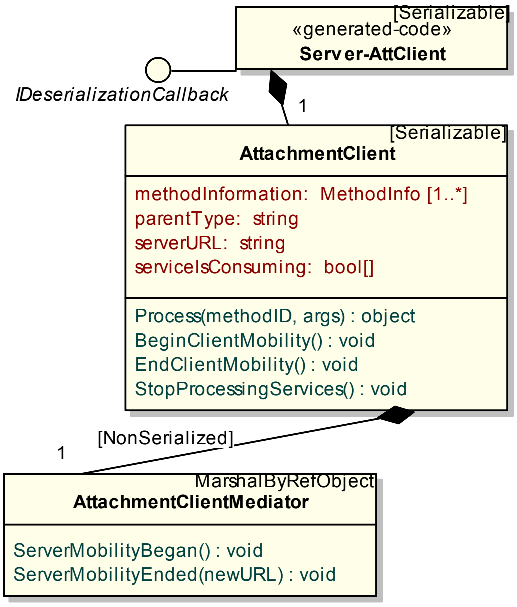

[Serializable]
«generated-code»
Server-AttClient
IDeserializationCallback
1
[Serializable]
AttachmentClient
methodInformation: MethodInfo [1..*]
parentType: string
serverURL: string
serviceIsConsuming: bool[]
Process(methodID, args) : object
BeginClientMobility() : void
EndClientMobility() : void
StopProcessingServices() : void
[NonSerialized]
1
MarshalByRefObject
AttachmentClientMediator
ServerMobilityBegan() : void
ServerMobilityEnded(newURL) : void

Figure 6 AttachmentClient classes

The creation of the *AttachmentClient* is done in several steps:

- If the client computer does not have the assemblies of the server object, it downloads them from the computer where the server is located at this point in time.
- An *AttachmentClient* object is created. It registers itself in the *AttachmentServer Mediator* of the server object. Thus, it obtains method information about available remote services.
- With this information, a proxy of the server is generated at runtime. The purpose of this proxy is to forward called methods through the infrastructure of attachments in a transparent way. We call it *Server-AttClient*, although its real name will depend on the server type that it represents.
- An instance of the generated *Server-AttClient* is returned to the client object.

The *Server-AttClient* class is generated by emitting MSIL code. It can be created in two ways: by implementing a specified server interface or by inheriting the server type. When it is instanced, a reference to an *AttachmentClient* object is provided, to which methods are forwarded. For each method, the generated code looks like this:

```
void NotifyNewLocation(string currentLocation) {
    object[] args =
        new object[1] {currentlocation};
    MethodID = 2;
    attClient.Process(MethodID, args); }
```

Each method has its related *MethodID* defined at generation time in order to provide it correctly to the *AttachmentServerMediator*. In .NET Remoting, by creating a derived class from the *RealProxy* class, proxies can be built in an easy way instead of emitting MSIL code. However, we cannot use this feature because this infrastructure only accepts objects that inherit from the *MarshalByRef* class. To support mobility, our generated proxy must be serializable, as discussed in section 3.2.4.

In order to minimize generated MSIL code, the *Server-AttClient* class is composed of an *AttachmentClient* class that defines all the functionality of method forwarding and mobility. The *Process()* method is responsible for forwarding the services to be executed to the *AttachmentServer Mediator*. Finally, the *AttachmentClientMediator* class contains the services that *AttachmentServer* is going to invoke in order to notify its mobility, which will be discussed below.

To illustrate, we describe how SearchAgents are created and connected with each other following our approach. First, SearchAgents are created in the host origin and registered in the *AttachmentManager* by providing a different *objectID* for each one. Next, they are connected to each other through the *ConnectTo* service and by providing the *objectID*s obtained in the previous step. Finally, the *Start()* method of the SearchAgents are invoked, so they will begin to move to remote databases to collect information.

3.2.4 *Object mobility*

In order to move an entire object (code + state) to a new host, the *AttachmentManager* class provides the *MoveObject()* service. As mentioned above, there must be an *AttachmentManager* object running at the target host in order to be able to receive the object and restore its state properly. The *MoveObject* service moves the specified object to the new specified computer taking into account the current communication processes. Communication processes are "frozen" while mobility takes place, and they are restored properly when mobility ends. Thus, the other objects to which the moved object was connected to are not aware of the mobility process. Moreover, an object can request to move itself autonomously. In this case, the object thread that requested the mobility is aborted when the mobility begins.

It is important to note that, in order to provide the objects with a high level of mobility transparency, we considered the objects as black boxes which we do not know anything about (i.e., their threads or the location where remote object references are stored) For this reason, the object to be moved is responsible for finishing all of its executing threads before starting mobility. In other words, the object must reach a secure state before requesting mobility. This cannot be done transparently by the middleware for two reasons. On the one hand, it is difficult to obtain all the running threads of a particular object. On the other hand, it is not possible in .NET (without modifying the CLR) to get the thread execution state (stack and instruction pointer) and to restore it in a new computer. In order to do that, we would need thread serialization capabilities. However, to overcome these limitations, an initialization method and its arguments can be provided to restore the execution state of the object when the mobility process ends.

Mobility is carried out in several steps. First, both the object to be moved and its communication processes (the attachments) are packaged by creating a *MobilityPackage* object (see Figure 7). Second, this object is serialized and transferred to the target host. Then, before deserializing the transferred object, the middleware checks whether the required assemblies are available at the current host. If not, they are downloaded from the host where the object comes from. Finally, the *Unpack*() method is invoked to restore the object and the attachments. If anything fails, the service *UndoMovement*() restores the object to its initial location.

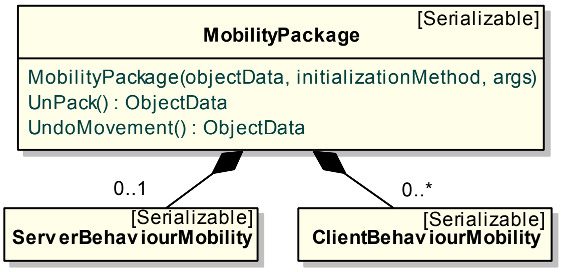


Figure 7 MobilityPackage classes

The mobility process depends on the role of the object to be moved: client or server behaviour. In the case of client behaviour, the *ClientBehaviourMobility* class obtains the *AttachmentClient* data (server location, server type and its unique ID) in order to rebuild it at the target host. This is because *MarshalByRef* objects cannot be serialized, as we stated above. Then, it invokes the *BeginClient Mobility* service of *AttachmentClient* to wait for pending requests to finish properly. Both the *Server-AttClient* and the object are serialized together, so on deserialization the object preserves the *Server-AttClient* reference without forcing the object to provide a setter property to update remote object references. However, as *Server-AttClient* is a dynamic assembly, it must be regenerated at the target host if this was not done before. Finally, at the target host, *EndClientMobility* service is invoked and the connection is restored to the *AttachmentServer* by notifying the new location of the client object.

In the case of server behaviour, each client object must be notified of the server mobility process so that services are not requested during this process. Similar to the *ClientBehaviourMobility* object, the data of the *AttachmentServer* (objectID, objectType and locations of connected *AttachmentClients*) is stored on a *ServerBehaviourMobility* object in order to rebuild it at the target computer. Then, the *ServerBehaviourMobility* object invokes the *BeginServerMobility* service of the *AttachmentServer* to notify the *AttachmentClients* of server mobility. Thus, each *AttachmentClient* blocks the arrival of new requests (by suspending incoming threads) and waits until current processing requests finish. When there are no more requests being processed by the server object, the mobility process can continue. Finally at destination, *EndServerMobility* service is invoked and connection is restored to the *AttachmentClients* by notifying its new location. In such the case that an object has both server and client behaviour, its mobility process will be the union of the above.

Simultaneous mobility is also supported. In other words, an object can move to another host while other objects, that are connected to it, are moved at the same time. Let's suppose that two SearchAgents 'Agnt1' and 'Agnt2' are connected, and 'Agnt1' is being moved. Then, 'Agnt2' also wants to be moved, but if it moves, 'Agnt1' will not be able to connect to it when 'Agnt1' ends its move. In order to do this, a message is left in the host where 'Agnt2' was. When 'Agnt1' ends its move and tries to connect to the last location of 'Agnt2', it will be notified with the new 'Agnt2' location.

In the SearchAgents case study, mobility takes place when an agent finishes collecting data at a certain database. Then, it invokes the *MoveObject*() service by specifying the next unvisited database where it wants to move and the service to be called when the mobility process ends (the *Start* method). When it arrives to the new database, the *Start* method is executed, and the SearchAgent continues its data collecting process.

4. EVALUATION AND RESULTS

Our approach has been implemented to compare the communication costs added by the attachments. We have measured these costs from when a client object requests a service until the results are returned. Without attachments, the average communication costs on a 100Mbit LAN are 0.9030ms. With the attachments (in the same conditions), the average communication costs are 1.0144ms (10.98%). The additional costs introduced, are due to 3 direct calls + 1 delegate dynamic invocation. Therefore, costs are increased because of dynamic invocation costs. For this reason, we also evaluated the performance by using a dynamically generated custom class [Gunn04a] instead of using delegates. This class was invoked by the *AttachmentServerMediator* in order to make direct method calls to the server object. Thus, the average costs have been reduced: 1.0010ms (9.79%). In the case of mobility, the costs are higher: there are communication and processing costs. The object, its related attachments, and the required assemblies are transferred. There are also several notification messages. The most important processing costs are due to the deserialization of transferred data and to the dynamic generation of *Server-AttClient* types.

5. CONCLUSIONS

In this paper, we have presented a lightweight middleware that can be easily included in other middlewares to provide mobility capabilities to its objects. Our approach supports weak mobility by using the attachments concept. Autonomous mobility for distributed objects is provided transparently and simultaneously, so the objects are not aware of the mobility process or the connection process of other objects to which they are linked. Moreover, the communication costs introduced are not very high, so an application can be mobility-adapted easily without slowing its performance. However, there are a few constraints. First, mobile objects must be *Serializable,* and they must manage their own threads before moving. Also, in order to establish the initial connection to a remote object, its current location must be known in advance. Nevertheless, our approach provides location-awareness after establishing a connection. The most common solution to obtain current locations of mobile objects is by having a centralized object that is updated with location changes from which clients can request these locations. However, this is not a decentralized approach and more work has to be done.

Furthermore, attachments add an abstraction layer over the communication infrastructure, so objects do not have to take into account what technology is used. Therefore, even though we implemented our approach in .NET Remoting, in the future we can adapt it to a Service-Oriented infrastructure such as Indigo, so that current running objects do not have to be aware of the underlying technology.

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.

7. REFERENCES

[Ali05a] Ali, N., Ramos, I., Carsí, J.A. *A Conceptual Model for Distributed Aspect-Oriented Software Architectures*. In International conf. on Information Technology Coding and Computing, (ITCC 2005), IEEE Computer Society, Las Vegas, USA 2005.

[Ber96a] Bernstein, P.A. *Middleware: a model for distributed system services*. Communications of the ACM, Volume 39, Issue 2, ISSN: 0001-0782, 86-98, 1996.

[Cha03a] Chaumette, S. and Vignéras, P. *A Framework for Seamlesly Making Object Oriented Applications Distributed*. In International conf. on Parallel Computing (PARCO 2003): 305-312, 2003.

[Fug98a] Fuggetta, A., Picco, G.P., and Vigna, G. *Understanding Code Mobility*. In IEEE Transactions on Software Engineering, 24(5): 342-361, 1998.

[Gunn04a] Gunnerson, E. *Calling Code Dynamically*. MSDN Library, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncscol/html/csharp02172004.asp, 2004

[Hic99a] Hicks, M., Jagannathan, S., Kesley, R., Moore, J.T. and Ungureanu, C. *Transparent Communication for Distributed Objects in Java*. In ACM Java Grande Conference, 160-170, June 1999.

[Obe02a] Obermeyer, P. and Hawkins, J. *Object Serialization in the .NET Framework*. MSDNLib.: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/objserializ.asp, 2002.

[Per05a] Pérez, J., Ali, N., Costa C., Carsí J.A., Ramos I. *Executing Aspect-Oriented Component-Based Software Architectures on .NET Technology*. International Conference on .NET Technologies, Plzen, Pilsen, Czech Republic, 2005.

[Rya04a] Ryan, C. and Westhorpe, C. *Application Adaptation through Transparent and Portable Object Mobility in Java*. In proc. of 2004 International Symposium on Distributed Objects and Applications (DOA 2004), Agia Napa, Cyprus, 2004, Springer-Verlag LNCS3291

[Tro03a] Troger, P. and Polze, A. *Object and Process Migration in .NET*. The 8th IEEE Intern. Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2003), Mexico, January 2003.

Aspect.NET — aspect-oriented toolkit for Microsoft.NET based on Phoenix and Whidbey

Vladimir Safonov
St. Petersburg State University, Russia
v_o_safonov@mail.ru

Mikhail Gratchev
St. Petersburg State University, Russia
9r@mail.ru

Dmitry Grigoryev
St. Petersburg State University, Russia
gridmer@mail.ru

Alexander Maslennikov
St. Petersburg State University, Russia
khan@tepkom.ru

28 Universitetsky prospect
Petrodvorets, St. Petersburg
198504 Russia

ABSTRACT

Aspect-oriented programming (AOP) methodology is evolving from research projects towards commercial applications. Most of the existing AOP tools suitable for commercial projects are intended for Java platform only which limits their applicability. Known AOP tools for Microsoft.NET such as Aspect#, Loom.NET, etc. are still at experimental stage. Most of them lack flexibility and comfortable user interface.

Aspect.NET, our AOP framework for Microsoft.NET, offers a new approach taking the best of Microsoft .NET specifics. Aspect.NET allows to define aspects using any language implemented for .NET that supports the concept of attribute. For aspect specification, we developed very simple and compact language-agnostic AOP meta-language - Aspect.NET.ML. At the source code layer, aspect definition in Aspect.NET looks like the code of a compilation unit annotated by Aspect.NET.ML constructs. The AOP annotations are converted into specific AOP custom attributes used by the Aspect.NET tool. Thus, an aspect assembly keeps all necessary information for aspect weaving whose result is represented as an augmented assembly.

Aspect.NET implementation is based on Microsoft Phoenix – state-of-the-art multi-targeted optimizing infrastructure for developing compilers and other language tools, in particular, comfortable for creating and editing .NET assemblies. The weaver uses Phoenix IR for scanning target applications and weaving aspects.

Aspect.NET Framework (GUI and aspect editor) is implemented as add-in to Microsoft Visual Studio.NET 2005 (Whidbey) and is seamlessly integrated into it. Important features of Aspect.NET Framework are: visualization of join points at source code layer, and user-controlled filtering potential join points before weaving.

Keywords

Aspect-oriented programming, Microsoft.NET, AOP meta-language, join point, weaving, Phoenix, Visual Studio.NET 2005, add-in.

1. INTRODUCTION

Modern AOP approach to software development is intended to solve a lot of issues related to increasing complexity of architecture, development and maintenance of software products. Aspect-oriented approach is helpful to simplify the business logic of an application, due to explicit separation of its cross-cutting concerns.

Well known examples of cross-cutting concerns are MT safety, security and logging.

More complicated example close to the authors' area of expertise is the task of extending a compiler by implementation of a new source language feature – e.g., generics in C#. It is clear that all the phases of the compiler should be updated for this purpose. So it is not enough to add new modules to the compiler but is it also necessary to insert into its code a number of tangled fragments to glue the new modules of the compiler to the existing ones.

Theoretical foundations of AOP are well defined by a variety of researchers [1, 17]. However, even basic AOP concepts are still understood and interpreted different way by different researchers and developers. Except for widely known AOP tools for Java – AspectJ [9] integrated into Eclipse, there are no AOP tools yet that could be easily integrated to the existing software development environments.

The goal of the Aspect.NET project [10, 11, 51] described in this paper is to create an AOP tool for Microsoft.NET [40] which would be flexible, language-agnostic and integrated to the latest Microsoft software development environment – Visual Studio.NET 2005.

A version of Aspect.NET for academic shared source .NET implementation - SSCLI / Rotor is also developed.

Aspect.NET allows to visualize the result of weaving at source code level, and to manually select or unselect potential join points.

The paper describes Aspect.NET principles, architecture, components, functionality, perspectives and ideas of future work on Aspect.NET.

.NET Technologies 2006
Copyright UNION Agency – Science Press,
Plzen, Czech Republic.

2. RELATED WORK

The AOP methodology founded by Gregor Kiczales [1] is similar to a number of approaches already used in software technologies for a few years - subject-oriented programming [25], composition filters [26], [14], adaptive programming [15], intentional programming [27], generative programming and transformational programming [16].

The papers [28 - 30] provide introduction to AOP and describe pluses and minuses of different approaches to AOP.

In [31, 32] the most popular AOP tool – AspectJ is described in detail. The Web site [22] contains a variety of information on all AOP approaches and tools. The paper [21] describes one of possible approaches to AOP for .NET based on interceptors.

Papers [35, 36] show the applicability of AOP approach for implementing object communication protocols' design patterns.

In [37], design-by-contract foundations are described, as a reliable software development technology. In our opinion, design-by-contract principles can be applied using AOP tools, Aspect.NET in particular.

The paper [36] proposes an approach to handling using AOP, provides some examples and gives some recommendations of AOP applicability at this software lifecycle stage.

Since Java was the most advanced software development platform in mid-1990s, the first AOP tools were developed for the Java platform. In particular, AspectJ [9] provides the following Java extensions

- *Aspects* – implementations of cross-cutting concerns;
- *pointcuts* – collections of patterns for join points selection and aspect weaving;
- *advices* – actions to be performed on reaching the aspect's joinpoints;
- inter-type declarations (*introduce*) — definitions of aspect members to be inserted into a target application in aspect weaving, but visible by the aspect only, rather than by the target application;.
- dynamic updates of control flow before, after or instead the code of a join point.

One of the key ideas of AspectJ - to perform a given action on reaching a given join point in the code – can be considered as an enhancement of the concept of breakpoint used in debuggers. But the most fundamental principle of AspectJ is to define a new kind of modules for aspect definitions. The paper [17] provides a systematic look at the existing AOP tools – AspectJ, HyperJ, Demeter, DemeterJ, and AOP models – PA, TRAV, COMPOSITOR and OC.

Another group of problems related to AOP is *aspect mining* [18 – 20], or as we call it *aspectizing* [10] — extracting aspects from non aspect-oriented applications. Aspectizing can be very helpful to improve readability and maintainability of applications. There are several research projects and tools for aspectizing implemented for the Java platform: Aspect Mining Tool (AMT) [18], Aspect Browser [19] and FEAT [20].

When the Microsoft.NET platform was developed, it appeared necessary to implement multi-language aspects, in the spirit of .NET language interoperability, rather than to limit aspects to be only extensions of Java or any other concrete programming language.

There are lots of examples of real cross-cutting concerns and their implementation, both in commercial and in research software projects, in particular for Microsoft.NET platform and its non-commercial SSCLI implementation. When looking at the code developed by the SSCLI team to port Rotor to MacOS, or at the code developed by Gyro (generics for Rotor) team, it is quite clear that both of these are actually aspects.

Currently there are a number of research projects to support AOP for Microsoft.NET. Among them are: Aspect#, Loom.NET, R#, Weave.NET, Wicca [53], Compose* [54]. The existing approaches to implementation of AOP for Microsoft.NET can be divided into four groups, according to the ways of representing aspects [56]:

- Using XML schemes for defining AOP specifications, e.g. SourceWeave.NET [52], Weave.NET [8], first versions of AspectDNG [47].
- Using COM+ style interceptors for dynamic weaving and activating AOP functionality. The configuration of the whole AOP system is described by XML files [21], [50].
- Using Composition Filters Model (CF) as extending special classes – Compose* [54].
- Using both custom attributes and XML - Aspect# [5]
- Using custom attributes - Aspect.NET [10], Phx.Morph [55], AspectDNG [47].

So our Aspect.NET approach relates to the fourth group, according to the above classification.

The most advanced of the existing AOP integrated development environments (IDE) for the Java platform is referred to as *AspectJ Development Tools (AJDT)* [45] developed by the AspectJ team as a plug-in to the Eclipse IDE to support using AspectJ tools.

There is another tool similar to AJDT for Eclipse — *AspectJ Development Environment (AJDE)* [42-44] that can be plugged into Emacs, JDEE, Sun Studio, NetBeans and JBuilder.

Phx.Morph is another research AOP project which uses the same weaving techniques based on MS Phoenix [12]. In this tool, weaving is performed using attribute-based annotations. The tool does not offer any AOP meta-language for aspect specifications. Lack of AOP meta-language makes readability of aspects and specification of non-trivial join points much more complicated.

Currently none of the existing AOP IDE for Microsoft .NET, prior to our Aspect.NET tool, has comfortable GUI. We think this is because of initial stage and research nature of the majority of AOP implementations for Microsoft.NET.

3. ASPECT.NET BASICS

An aspect in Aspect.NET [10, 11] is defined as a source code of a class (more generally speaking, a compilation unit) in C# or other .NET language, annotated by our simple AOP meta-language (referred to as *Aspect.NET.ML*) statements to highlight the parts of aspect definition. They are: *aspect header* (with optional *parameters*), (optional) *aspect data*; *aspect modules* (methods or functions), and *aspect weaving rules* which, in their turn, consist of *weaving conditions* and *actions* (to be woven into a target assembly, according to these rules). The structure of Aspect.NET.ML meta-language is so simple and self-explanatory that we decided to explain it using our examples given below, rather than provide its precise EBNF definition.

The aspect weaving rules determine the *join points* within a target application where the actions of the aspect are to be woven. The aspect actions provide the aspect's functionality.

Speaking in terms of knowledge management, aspect weaving rules can be considered as special kind of knowledge (a rule set) defining how to apply the aspect to a target application.

There can also be weaving rule sets separate from concrete aspects, similar to pointcuts in AspectJ.

Unlike AspectJ, a Java extension for AOP, in Aspect.NET, due to use of language-agnostic AOP annotations, it becomes possible to avoid the issue of extending each of the .NET languages by its own AOP extensions specific of that language.

The *Aspect.NET pre-processor* converts the AOP annotations to definitions of AOP custom attributes (*AspectDef*), specially designed for Aspect.NET, to mark classes and methods as parts of the aspect definition (see fig. 1). Next, an appropriate common use .NET compiler transforms the AOP custom attributes to the aspect assembly's metadata stored together with the MSIL code.

Join points in Aspect.NET are determined by weaving rules which are parts of aspect definition, or are defined in a separate rule set module. The weaving rules contain: *conditions* of calling aspect actions (*before, after,* or *instead*); *context* of the action call (a *call* of some method, *assign* to a variable (field), or *use* of some variable (field); *wildcard* to find the context of the aspect action's call.

The process of aspect weaving consists of *two phases – scanning* (finding join points within the target application) and *inserting (weaving)* the calls of the aspect actions into the join points found.

Unlike many other AOP tools, Aspect.NET allows the user to *select or unselect* any of the possible join points using Aspect.NET Framework GUI, to avoid "blind" weaving that could make the resulting code much less understandable and actually non-debuggable.

4. ASPECT.NET DESIGN

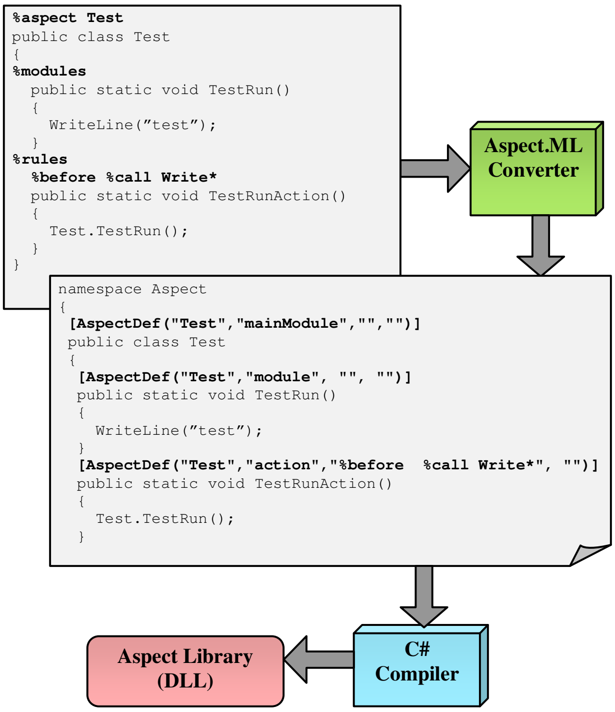
The Microsoft.NET platform is based on the principles of peer-to-peer multi-language programming. For any of the .NET languages, a very comfortable toolkit for software development and maintenance is provided – Microsoft.NET Framework and Visual Studio.NET.

The need to support multi-language programming makes the task of weaving and locating aspects more complicated, as compared to the Java platform.

For example, AspectJ [9] is actually an implementation of Java extension by AOP constructs and concepts. AspectJ consists of the extended Java language compiler and a set of specific utilities that can work with this Java extension only.

To avoid developing a separate compiler for each of the .NET languages for the purpose of implementing multi-language AOP, Aspect.NET uses custom attributes to represent information on aspects. Due to that, an aspect definition in Aspect.NET is a syntactically and semantically correct source code of a compilation unit, with AOP custom attributes added to annotate parts of aspect. Typically, an Aspect.NET aspect is converted to a class with its fields and methods, marked by AOP custom attributes, intended for compilation into a .NET assembly by the appropriate common use .NET Framework compiler. The AOP custom attributes are usable and understandable by Aspect.NET only. They are stored together with the rest of the aspect assembly and don't prevent from normal functioning of the other .NET tools. Due to our approach, there is no need to make a special "AOP-aware" version of the .NET Framework or Visual Studio.NET.

Full compatibility of Aspect.NET aspects to all the .NET tools makes it possible to use all the code refactoring, analysis, profiling and other features of .NET tools, while working with an aspect definition. Moreover, all the existing OOP quality criteria and metrics are applicable to .NET aspect-oriented applications based on Aspect.NET.



```
%aspect Test
public class Test
{
%modules
  public static void TestRun()
  {
    WriteLine(”test”);
  }
%rules
  %before %call Write*
  public static void TestRunAction()
  {
    Test.TestRun();
  }
}
```

```
namespace Aspect
{
 [AspectDef("Test","mainModule","","")]
 public class Test
 {
  [AspectDef("Test","module", "", "")]
  public static void TestRun()
  {
    WriteLine(”test”);
  }
  [AspectDef("Test","action","%before  %call Write*", "")]
  public static void TestRunAction()
  {
    Test.TestRun();
  }
```

Figure 1. Aspect.NET.ML conversion to custom attributes

Aspect weaving is performed "statically" (see fig. 2), at the layer of .NET intermediate representation language (MSIL) and metadata, rather than at source code layer. All weaving-related transformations are made by the Aspect.NET toolkit. There is no need to transform in any way either source or intermediate code of a target application before weaving Aspect.NET aspects.

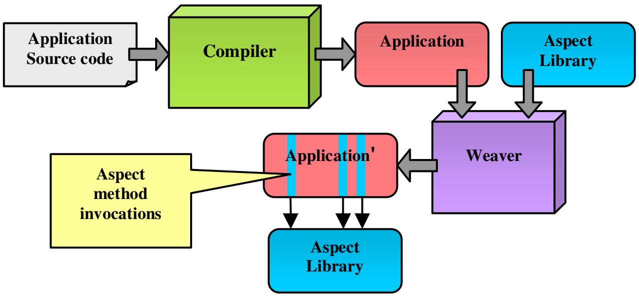


Figure 2. Static weaving in Aspect.NET

The advantages of static aspect weaving in Aspect.NET, as compared to dynamic weaving (e.g., in LOOM.NET [2]) and load-time weaving (e.g., in Weave.NET [8]), are higher performance and better understandability of a target application with the aspects woven. Dynamic weaving is usually implemented with the help of some debugging API which makes the operating system perform checking of each executable code instruction to satisfy some specific conditions, and to enable jumping to some other appropriate part of code when the condition is satisfied. Such dynamic checks may dramatically decrease performance. On the contrary, due to Aspect.NET approach, when using MSIL code of the resulting assembly it is quite possible to track the results of aspect weaving in vast detail by .NET utilities (*ilasm/ildasm,* debuggers, etc.) Thus, a developer who uses Aspect.NET is guaranteed to get a predictable and understandable resulting application after weaving. So the user does not need to use any kind of tricky checks of the results of weaving aspects, any non-trivial kinds of debugging, etc.

Up to the present moment, the main reason why similar AOP toolkits haven't yet been developed for .NET was the lack of adequate common use tools for analyzing and updating .NET assemblies (whose structure is very complicated) at the layer of MSIL intermediate code and metadata. To handle assemblies, some of the developers had to use RAIL [50] or to reinvent a wheel by developing their own, limited toolkit for this purpose.

Our Aspect.NET tool is based on Microsoft Phoenix [12] – a multi-targeting optimizing compiler back-end development environment. Phoenix provides a convenient high-level API to create, handle and update .NET assemblies by transforming it into high-level Phoenix IR (HIR) suitable for any program transformations like weaving. The resulting assembly unit is converted back to MSIL and metadata format. The latest version of Phoenix is dated November 2005 and is available within the framework of Phoenix Academic Program [12].

One of main shortcomings of the existing experimental AOP tools for .NET (Aspect# [5], AOP.NET [3], etc.) is the lack of functionality for analyzing and debugging the results of weaving aspects.

As for Aspect.NET, all its components have the central part, Aspect.NET Framework, implemented as an add-in to Microsoft Visual Studio.NET 2005. Due to that, the user can, for example, visualize the results of aspect weaving at the source code level.

Also, a version of Aspect.NET compatible to the Shared Source Common Language Infrastructure (Rotor) is developed. Currently it is based on command-line interface using Perl scripts. This version also uses Phoenix and is based on the same weaver.

The main components of Aspect.NET (see fig. 4) are as follows:

- Weaver
- Meta-language converter
- Aspect.NET Framework

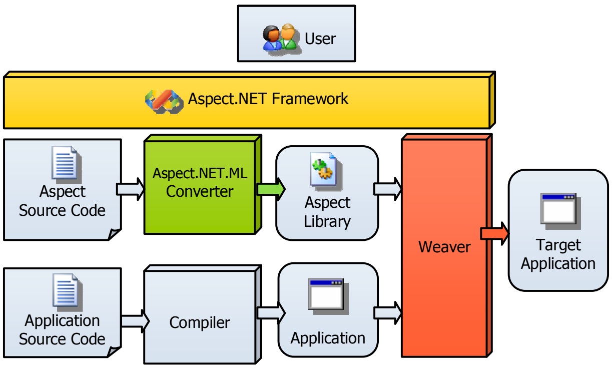


Figure 4. Components of Aspect.NET

Aspect.NET Framework allows the user to define aspects in Aspect.NET.ML meta-language, by creating a new kind of project (*Aspect*) and using a skeleton of the aspect source code generated by our wizard (see fig. 3), to map potential join points into the original target assembly's source code, and to visualize the results of weaving.

To collect information on the potential join points in the target assembly, as well as to perform aspect weaving itself, Aspect.NET Framework uses the functionality of the weaver. At the *scanning* phase, the weaver matches the code of the target assembly against the aspect (using its weaving rules), and creates the list of the potential join points. At the *weaving* phase, the actions of the aspect are woven into the target assembly. The user can edit the list of the (potential) join points, based on visualizing the join points within the target assembly's source code.

5. ASPECT.NET.ML CONVERTER

Aspect.NET.ML converter transforms user-defined aspects from AOP meta-language into source code fully written in the aspect's implementation language, annotated by AOP custom attributes. Also, the converter calls the appropriate common use .NET language compiler to compile the resulting source code into a .NET assembly.

Implementation of the converter is based on *CodeDom* – a set of .NET Framework classes for generating and handling object-style representation of a .NET source code. The aspect definition is transformed into a CodeDom graph which allows to modify the source code and to use language-independent form of aspect definition inside Aspect.NET.

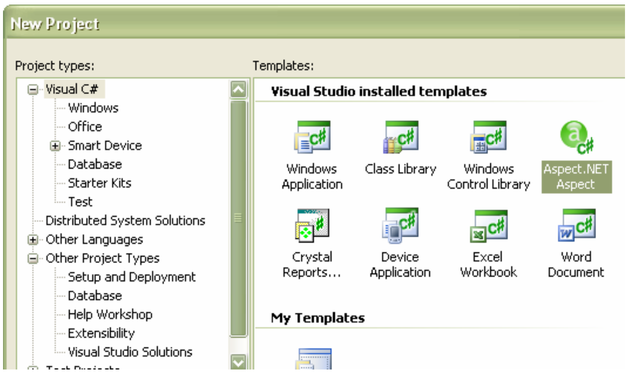


Figure 3. Creating a new Aspect.NET aspect project

Specifically for Aspect.NET, we introduced a new kind of Visual Studio.NET project – *Aspect* that includes a code pattern for aspect definition and all related resources. Thus, seamless integration into the Visual Studio IDE is enabled, and aspect reuse becomes easier.

On creating a correct aspect definition by the user, it is converted, then compiled into an assembly, and automatically passed to the *aspect browser* for its subsequent use within the Aspect.NET Framework.

In the aspect example in AOP meta-language (please see Appendix A), the keyword *%aspect* starts the *aspect header* that contains its *name* (in this example - *Politeness*), and can also contain *parameters* (lacking in this example). Then goes the *%modules* part where the aspect modules (methods) are defined. In the *%rules* part, the aspect *actions* are defined, each of them preceded by its *weaving rule*. In this example, the first action is to be inserted *before* calling each method of the target application, the second one - *after* each of its method calls.

In the next listing (see appendix B), the source code of the *Politeness* aspect generated by the converter is presented. All the members of the aspect's implementation class are marked by appropriate AOP custom attributes.

6. WEAVER DESIGN APPROACH

In Aspect.NET, weaver is implemented as a separate application, which allows to distinguish between weaving itself and its mapping into the source code. So, access to source codes of a project is not mandatory for subsequent weaving which is performed at the level of binary representations of the target assembly and the aspect assembly.

To find and analyze join points, the weaver uses high-level intermediate representation (HIR) of the binary target assembly generated by Phoenix [12]. Each executable module of the assembly is represented by a graph of high-level instructions which enable access to their source and destination arguments, debugging information, information on the parent unit, etc. The Phoenix API enables, on loading a MSIL assembly represented by a PE file, to get access to control and data flow, to the list of modules and instructions, to detailed information on types and symbols, to information on variable dependencies, etc. This makes possible to find a variety of the kinds of join points, and makes the weaving independent of concrete aspect implementation language. In scanning mode, the weaver scans this instruction stream, finds the join points (based on the weaving rules), and passes their coordinates in the target application to the Aspect.NET Framework add-in which presents them to the user.

Next, on getting from the framework the list of the user-selected join points, the weaver starts its weaving mode, scans the instruction stream of the target application, and finds the user-selected join points. Then, the weaver generates instructions for calling aspect's actions with the appropriate arguments. The arguments of the action can be the target method name and the pointer to the target object whose method is called. The weaver injects the generated aspect's action call instructions into the join point specified by the weaving rule, - before, instead or after the target call.

7. CASE STUDY: ASPECT.NET IN ACTION

Now let's consider in more detail the scenario of using Aspect.NET and the principles of its functioning.

1. The user defines an aspect in AOP meta-language and passes the source code of the aspect (as part of the Visual Studio's *Aspect* project) and the target application's source code (also a Visual Studio project) into the Aspect.NET Framework (see fig. 5).

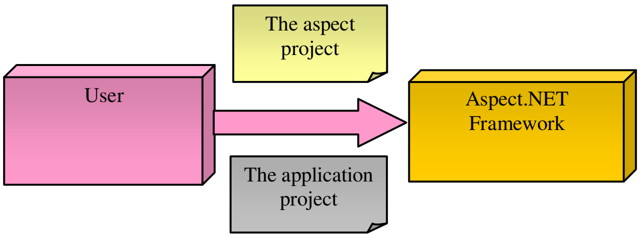


Figure 5. Creating the aspect and the application projects

2. Aspect.NET Framework initiates the compilation of the source code of the target application by the .NET compiler from the appropriate language, to create the target assembly with its debugging information (.pdb file). Also, Aspect.NET Framework passes the source code of the aspect to the AOP meta-language converter which, in its turn, converts the source code with meta-language annotations into a source code with AOP custom attributes, and generates a ready-to-use aspect assembly (by calling the .NET compiler). See fig. 6.

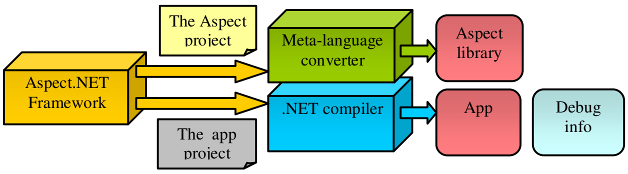


Figure 6. Preparing the aspect and the target application for weaving

3. To create a list of all possible join points within the target application, Aspect.NET Framework invokes the scanning phase of the weaver. To map the join points to the source code of the target application, Aspect.NET Framework provides the weaver with its debugging information (for Microsoft .NET Framework – represented as .pdb file, for Rotor – as .ildb file) The weaver performs the scanning and generates the join points list as an XML document (see fig. 7).

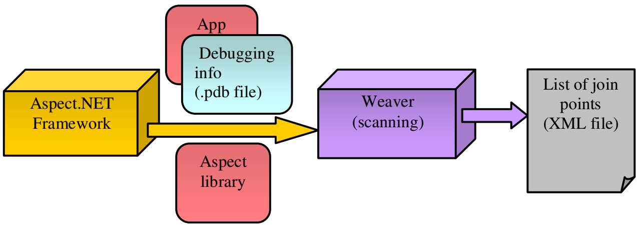


Figure 7. Generating the list of joinpoints

4. Based on the XML file, Aspect.NET Framework creates a GUI representation of the join points list, so that the user could visualize each of the join points within the editor of the code of the target application. The user can also filter the set of the join points by unselecting any of them. Then, Aspect.NET Framework passes the updated list of the join points and the other relevant working files to the weaver for the phase of weaving itself. As the result of weaving, the user obtains the updated target application's assembly (see fig. 8).

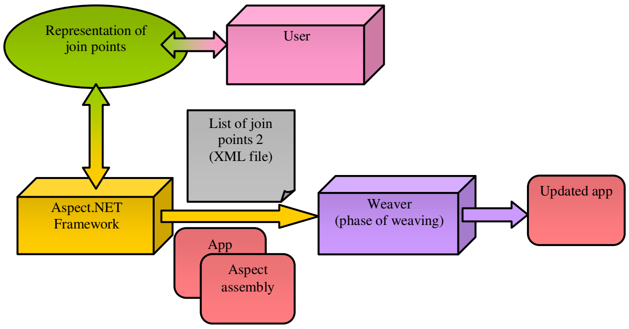

Representation of join points
User
List of join points 2 (XML file)
Aspect.NET Framework
Weaver (phase of weaving)
Updated app
App
Aspect assembly

Figure 8. Join points filtering and weaving

Due to such scenario, the phases of scanning and weaving are separated. This opens great opportunities for software maintenance and configuration. For example, instead of passing to the client an updated version of a big monolithic application, it will be enough to pass the list of join points (internally represented in Aspect.NET as an XML file), the assemblies of the aspects implementing new functionality, the weaver application itself, and a simple script to initiate weaving on the client side. Thus, if the user would like to create a new version of the application with extended functionality, she just needs to configure the weaving of the appropriate aspects.

8. ASPECT.NET FRAMEWORK: FUNCTIONALITY OVERVIEW

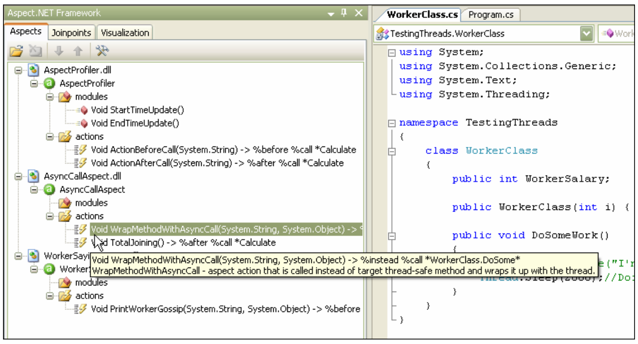
Aspect.NET Framework provides user-level functionality for examining, studying and understanding Aspect.NET aspects. It contains:

- *Aspect browser*, to examine aspect DLLs, their weaving rules, and comments to them provided at aspect design stage in Aspect.NET.ML.

- *Join points tree*, displaying the hierarchy of namespaces, classes and methods of the target assembly's project, whose leaves are the possible join points.

- *Visualizer*, to display the mapping of the join points onto the source code of the target assembly.

8.1 Aspect browser



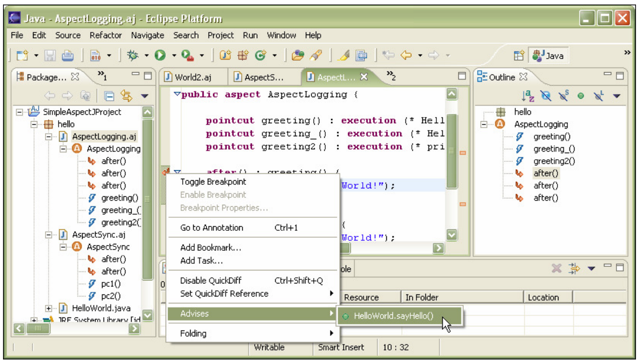
AspectProfiler.dll
AspectProfiler
modules
Void StartTimeUpdate()
Void EndTimeUpdate()
actions
Void ActionBeforeCall(System.String) -> %before %call *Calculate
Void ActionAfterCall(System.String) -> %after %call *Calculate
AsyncCallAspect.dll
AsyncCallAspect
modules
actions
Void WrapMethodWithAsyncCall(System.String, System.Object) ->
Void TotalJoining() -> %after %call *Calculate
WorkerSayi
Worker
modules
actions
Void PrintWorkerGossip(System.String, System.Object) -> %before

Void WrapMethodWithAsyncCall(System.String, System.Object) -> %instead %call *WorkerClass.DoSome*
WrapMethodWithAsyncCall - aspect action that is called instead of target thread-safe method and wraps it up with the thread.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;

namespace TestingThreads
{
    class WorkerClass
    {
        public int WorkerSalary;

        public WorkerClass(int i) {

        public void DoSomeWork()
        {
                                    e("I'r
            Thread.Sleep(2000);//Do
        }
    }
}
```

Figure 9. Aspect browser

Fig. 9 illustrates the Aspect.NET aspect browser functionality. The user can take a look at any of the available aspects, their modules and actions, and comments to them. The functionality of the browser is similar to the Outline View in the AJDT for Eclipse [45] (see fig. 10).



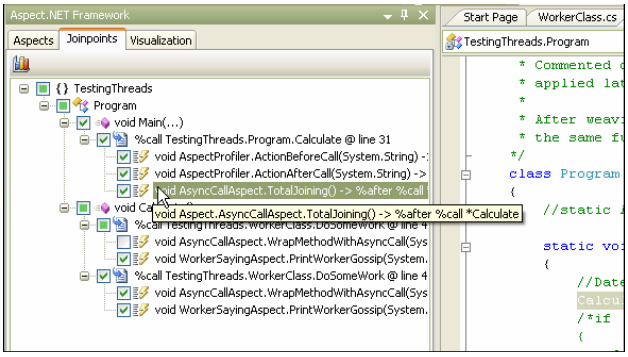
```
public aspect AspectLogging {

    pointcut greeting() : execution (* Hell
    pointcut greeting_() : execution (* Hel
    pointcut greeting2() : execution (* pri

    after() : greeting() {
                                World!");

                            {
                                World!");
```

Figure 10. The aspect browser in AJDT for Eclipse.

The browser allows to change the order of the aspects, to resolve possible conflicts related to the order of weaving aspects to an application. So, if actions of the two aspects affect the same join points in the application, the rules of the aspect displayed higher will be applied before the rules of the one displayed lower.

8.2 Join points tree

On completion of scanning the target assembly by the weaver, the Aspect.NET Framework creates a join points tree, and displays it for the user (see fig. 11). The join points are represented by information on their actions to be called, and on how they will be woven according to the weaving rules – before, after or instead the join point code. By clicking at the join point leaves of the tree, the user can take a look at the appropriate points in the source code of the target application.



```
□ {} TestingThreads
  □ Program
    ☑ void Main(...)
      ☑ %call TestingThreads.Program.Calculate @ line 31
        ☑ void AspectProfiler.ActionBeforeCall(System.String) -:
        ☑ void AspectProfiler.ActionAfterCall(System.String) ->
        ☑ void AsyncCallAspect.TotalJoining() -> %after %call
    □ void Ca
      □ %call TestingThreads.WorkerClass.DoSomeWork @ line 4
        □ void AsyncCallAspect.WrapMethodWithAsyncCall(Sys
        ☑ void WorkerSayingAspect.PrintWorkerGossip(System.
      ☑ %call TestingThreads.WorkerClass.DoSomeWork @ line 4
        ☑ void AsyncCallAspect.WrapMethodWithAsyncCall(Sys
        ☑ void WorkerSayingAspect.PrintWorkerGossip(System.
```

void Aspect.AsyncCallAspect.TotalJoining() -> %after %call *Calculate

```
       * Commented c
       * applied lat
       *
       * After weav
       * the same fu
      */
      class Program
      {
          //static i

          static voi
          {
              //Date
              Calcu
              /*if
              {
```

Figure 11. Join points tree

Due to the join points tree, the user can get full information on possible effect of weaving, and visually check the correctness of possible weaving into each join point before the weaving is actually done, so that undesirable join points could be unselected.

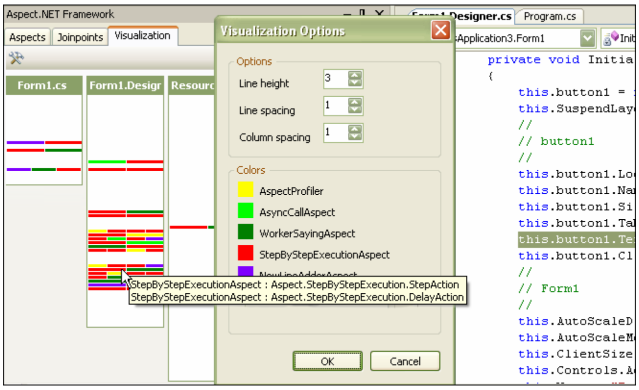
So, as opposed to AJDT for Eclipse, in Aspect.NET the user can visualize and control the process of join points filtering. In AJDT, the user can affect the selection of join points only by changing pointcut definitions in AspectJ language, which is not so comfortable and promptly, since it requires recompilation.

“Blind” weaving on the basis of wildcards only (i.e., based on lexical level of the source code instead of its semantic level) can be very dangerous. For example, if the user of an AOP tool would like to insert some actions before and after updating some common global resource to be synchronized on, and expresses the pattern for seeking the operation that updates the resource just by the *Set** wildcard for the name of the method, the result of weaving could be also inserting the aspect’s actions before and after the calls of “harmless” methods like *SetColor*.

So, we do think our design decision and functionality for manual filtering join points could be beneficial, until an appropriate semantic level approach is invented for this purpose, which should be the matter of a further research.

8.3 Visualization of aspect weaving effect

In order to help user understand aspect weaving effect on the target application, a specific component of Aspect.NET Framework was developed - *aspect weaving visualizer* (see fig. 12).



Options
Line height 3
Line spacing 1
Column spacing 1

Colors
🟨 AspectProfiler
🟩 AsyncCallAspect
🟩 WorkerSayingAspect
🟥 StepByStepExecutionAspect
🟪 NewLineAdderAspect

StepByStepExecutionAspect : Aspect.StepByStepExecution.StepAction
StepByStepExecutionAspect : Aspect.StepByStepExecution.DelayAction

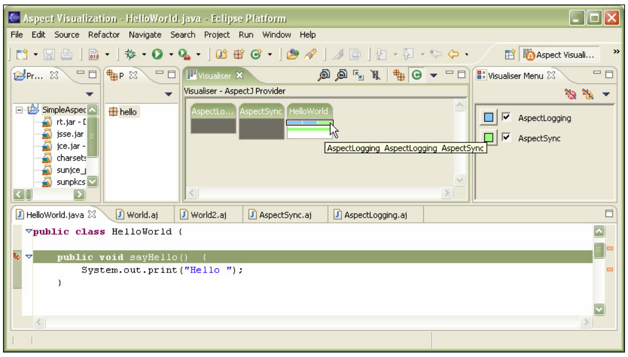
```
private void Initia
{
    this.button1 =
    this.SuspendLay
    //
    // button1
    //
    this.button1.Lo
    this.button1.Na
    this.button1.Si
    this.button1.Ta
    this.button1.Te
    this.button1.Cl
    //
    // Form1
    //
    this.AutoScaleD
    this.AutoScaleM
    this.ClientSize
    this.Controls.A
```

Figure 12. Aspect weaving visualizer

Each aspect woven into the target application is indicated by its own color that can be reselected by the user.

Visualization of each of the aspects can be turned on or off.

In Aspect.NET visualization is implemented similar to AJDT for Eclipse (see fig. 13) which, in its turn, inherited it from Aspect Browser [19].



🟦 ☑ AspectLogging
🟩 ☑ AspectSync

AspectLogging AspectLogging AspectSync

```
public class HelloWorld {

    public void sayHello()  {
        System.out.print("Hello ");
    }
```

Figure 13. Aspect weaving visualizer in AJDT for Eclipse.

In accordance to the Seesoft graphic notation [38], each of the vertical columns represents one of the source files of the application. The height of the column is proportional to the size of the file. Colored marks inside the columns correspond to the join points where aspects are woven. Each mark corresponds to one action of an aspect. One horizontal line with one or more marks corresponds to a line in the source code. When clicking at any of the marks, a popup window is displayed with comments to the corresponding action of an aspect. By clicking at a horizontal line, the user can view the corresponding source code lines in the common use editor of the source code. Filtering join points with the help of join points tree is synchronized with the functionality of visualizing the effect of aspect weaving.

9. FUTURE WORK

9.1 Aspect debugger

In near future, we plan to add to the common use Visual Studio debugger an add-in for full-fledged debugging of Aspect.NET applications in terms of aspects. Due to that, the user will be able not only detect bugs in the aspect code, but to also trace and watch step by step the behavior of the resulting target application in terms of aspects.

9.2 Display changes in the source code

As our research shows, the main difficulty of applying AOP for commercial projects is the impossibility to estimate interaction of the woven aspects and business logic code. Aspects are woven at compile time or dynamically, and the result is a ready-to-use binary assembly. It is currently not possible to predict the behavior of the target application after aspect weaving. Aspect.NET can help to solve this task, either by aspect coloring in terms of the source code, or by creating unit tests by Visual Studio development environment. We also plan to provide the user with an opportunity to “finger” how his source code has been changed after aspect weaving, by visualizing appropriate fragments of decompiled code of the modified target assembly.

9.3 Weaving rules analysis and transformations

We think a prospective addition to Aspect.NET Framework could be functionality for automated simplification of weaving rules or converting them to more readable form. For example, a rule of the kind:

(%after %call *) || (%after %call MyMethod)

can surely me replaced by a simpler but equivalent rule:

%after %call *.

If an aspect is being developed for some concrete target application, a functionality to convert its weaving rules based on the specifics of the target application could also be helpful. For example, if the application contains the two methods only, "MyMethod1" and "MyMethod2", then the weaving rule:

(%after %call MyMethod1) || (%after %call MyMethod2)

could be converted to a shorter one:

(%after %call MyMethod*)

or, vice versa, the latter rule could be converted to the former one.

9.4 Weaving rule reader

Similar functionality was discussed in [39]. The weaving rule reader will provide functionality for generating an adequate comment to a weaving rule in English. For example, the weaving rule:

%call %before (private *.set*(..,int))

could be commented by the following phrase: "before any call of a private method whose name starts with "set", defined on any type, and whose last argument is an int."

9.5 Interactive generator of weaving rules

Logical enhancement of the idea of weaving rules wizard could be a functionality to support generation of weaving rules in interactive mode, based on the existing code of the target application, for example, by clicking at the points of the source code of the target application to be affected by the aspect weaving rule being designed. This task requires a separate research.

9.6 Refactoring

In Visual Studio.NET 2005, advanced code refactoring functionality is supported - automated renaming members of an application, extracting interfaces from classes, transforming fragments of code into separate methods, etc.

For more enhanced support of Aspect.NET, this set of refactoring transformations could be extended by actions like "transform a method to an aspect's action", or "convert a data definition in an application into an inter-type declaration". Supporting these two functions would be actually equivalent to a basic built-in aspectizer [10].

9.7 Aspects repository and aspect knowledge

To enable enterprise or higher level reuse of aspects, an aspect repository could be created and maintained. Aspect.NET Framework could perform searching in this repository, based on the problem domain and other parameters. When finding a suitable aspect, Aspect.NET Framework could weave it into the target project.

In longer perspective, we think a separate research could be helpful to investigate more formal and semantic-level representation, extraction and use of aspect knowledge, since in our viewpoint aspects can be regarded as special kind of knowledge on how to transform, enhance and maintain software projects and applications.

11. CONCLUSIONS

The growth of popularity of Microsoft .NET among software developers stimulates development of AOP tools for that platform. But the "single language" approach to AOP, i.e. implementing AOP features as extensions to some concrete language, may dramatically limit their applicability, and their integration to common use .NET software development tools and technologies. Other shortcomings of the single-language approach are lack of tools for visualizing the results of aspect weaving, and low performance of the resulting target applications.

The goal of our group is further developing of Aspect.NET which we hope is an adequate AOP tool for Microsoft.NET. Due to our general and simple approach, it provides comfortable mechanism for ubiquitous use of AOP as part of one of the most advanced software development environments – Visual Studio.NET. The proposed approach is based on AOP custom attributes, on static aspect weaving at .NET assembly level, and on using the source code of target projects to visualize the results of weaving. The proposed simple, expressive and powerful AOP meta-language enables language-agnostic AOP for the .NET platform.

Aspect.NET Framework, the user-oriented part of our system, provides a rich set of features to analyze and understand aspects and target applications subject to weaving.

We think the functionality of Aspect.NET Framework is approaching to that of the most advanced AOP tool - AspectJ Development Tools for Eclipse [45], and has no analogs for Microsoft.NET platform.

The first working prototypes of Aspect.NET versions for Visual Studio.NET 2005 and for Rotor, with the Aspect.NET articles and examples, are available at [51]. The pre-requisites of using Aspect.NET are to install Visual Studio.NET 2005 and Phoenix.

12. REFERENCES

[1] G. Kiczales, J. Lamping, A. Mendhekar, etc. Aspect-oriented programming.- In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Finland, Springer-Verlag LNCS 1241. June 1997

[2] The LOOM .NET Project: ttp://www.dcl.hpi.uni-potsdam.de/research/loom/

[3] M. Blackstock.. Aspect Weaving with C# and .NET. http://www.cs.ubc.ca/~michael/publications/AOPNET5.pdf

[4] Y. Xiong, F. Wan. CCC: An Aspect Oriented Intermediate Language on .NET Platform

http://www.fit.ac.jp/~zhao/waosd2004/pdf/Xiong.pdf

[5] Aspect# home page: http://aspectsharp.sourceforge.net/

[6] AOP.NET home page: http://sourceforge.net/projects/aopnet/

[7] D. Lafferty, V. Cahill. Language Independent Aspect Oriented Programming. Proceedings of OOPSLA March 2003

[8] Weave.NET: www.dsg.cs.tcd.ie/sites/Weave.NET.html

[9] The AspectJ Project, www.aspectj.org

[10] V.O.Safonov. Aspect.NET: a new approach to aspect-oriented programming. - .NET Developer's Journal, 2003, #4.

[11] V. O. Safonov. Aspect.NET: concepts and architecture. - .NET Developer's Journal, 2004, # 10.

[12] Microsoft Phoenix home page. http://research.microsoft.com/phoenix

[13] The R# project: http://rsdn.ru/projects/rsharp/article/rsharp_mag.xml

[14] M. Aksit, L. Bergmans, and S. Vural. An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach. – In: Proceedings of the ECOOP'92 Conference, LNCS 615, Springer-Verlag, 1992

[15] K.Leiberherr. Component Enhancement: An Adaptive Reusability Mechanism for Groups of Collaborating Classes. – In: Information Processing'92, 12th World Computer Congress, Madrid, Spain, J. van Leeuwen (Ed.), Elsevier, 1992, pp.179-185

[16] Krzysztof Czarnecki, Ulrich Eisenecker Generative Programming: Methods, Tools, and Applications, Addison-Wesley, Paperback, Published June 2000.

[17] Masuhara, J., Kichales, G. Modeling Crosscutting in Aspect-Oriented Mechanisms. Proceedings of ECOOP'2003

[18] Hannemann, J., Kichales, G. Overcoming the Prevalent Decomposition in Legacy Code. Proceedings of Workshop on Advanced Separation of Concerns, International Conference on Software Engineering (May 2001, Toronto, Canada)

[19] Aspect Browser: Bill Griswold's Web pages (University of California, San Diego): www.cs.ucsd.edu/users/wgg

[20] FEAT: Martin Robillard's and Gal Murphy's Web pages (University of British Columbia, Canada): www.cs.ubc.ca/~mrobilla/feat/index.html

[21] Shukla, D., Fill, S. and Sells, D. Aspect-Oriented Programming Enables Better Code Encapsulation and Reuse. MSDN Magazine, March 2002.

[22] Aspect-oriented software development Web site: www.aosd.net

[23] K.Czarnecki. Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models. PhD thesis, Technische Universitat Ilmenau, Germany, 1998.

[24] Rational Software Corporation: www.rational.com

[25] Homepage of the Subject-Oriented Programming Project, IBM Thomas J. Watson Research Center, YorktownHeights, New York, http://www.research.ibm.com/sop/

[26] Homepage of the TRESE Project, University of Twente, The Netherlands, http://wwwtrese.cs.utwente.nl/; also see the online tutorial on Composition Filters at http://wwwtrese.cs.utwente.nl/sina/cfom/

[27] Ch. Simony. The Death of Computer Languages, The Birth of Intentional Programming, Microsoft Research, 1995, http://research.microsoft.com/pubs/view.aspx?tr_id=4.

[28] G. Kiczales, J. Lamping, A. Mendhekar, etc. Aspect-oriented programming. *P*ublished in proceedings of the European Conference on Object-Oriented Programming (ECOOP). Finland, Springer-Verlag LNCS 1241. June 1997.

[29] E. Zhuravlev, V. Kiryanchikov. On the opportunity of dynamic aspects integration in aspect-oriented programming. – Proc. of Electro-Technical University, Informatics, control and computing technologies, 2002, vol, 3, pp.. 81 — 86 (in Russian)

[30] Laddad, R. (2002). I want my AOP part 1. JavaWorld. Avaliable at http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html.

[31] The AspectJ Programming Guide, 1998-2002, Xerox Corporation

[32] I. Kiselev. Aspect-Oriented Programming with AspectJ. Indianapolis, IN, USA: SAMS Publishing, 2002.

[33] E. Gamma, R. Helm, R, Johnson, J. Vlissides, Methods of object-oriented design. Design patterns. – Piter publishers, St. Petersburg, 2001 (Russian translation).

[34] S. Stelting, O. Maassen. Applying Java patterns. –Williams publishers, St. Petersburg, 2002 (Russian translation)

[35] J. Hannemann, G. Kiczales. Design pattern implementations in Java and AspectJ. In: OOPSLA 02, New York, USA, November 2002. P. 161 — 173.

[36] M. Lippert, C Videira Lopes. A Study on Exception Detection and Handling Using Aspect-Oriented Programming. Xerox PARC Technical Report P9910229 CSL-99-1, Dec. 99

[37] B. Meyer, Applying Design by Contract, Prentice Hall, 1992

[38] Eick, S.G., J.L. Steffen, and E.E. Sumner, Seesoft – A Tool For Visualizing Line Oriented Software Statistics. IEEE Transactions on Software Engineering, 1992. 18 (11).

[39] Aspect-Oriented Programming with AJDT, Andy Clement, Adrian Colyer, Mik Kersten http://www.comp.lancs.ac.uk/computing/users/chitchya/AAOS2003/Assets/clemas_colyer_kersten.pdf

[40] Richter, J. Programming for Microsoft.NET Framework. Microsoft Press, 2002

[41] HyperJ: www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm

[42] AJDE for Emacs and JDEE: http://aspectj4emacs.sourceforge.net/

[43] AJDE for SunONE/NetBeans: http://aspectj4netbean.sourceforge.net/

[44] AJDE for JBuilder: http://aspectj4jbuildr.sourceforge.net/

[45] Eclipse AspectJ Development Tools project: http://www.eclipse.org/ajdt

[46] Eclipse.org - Main Page: http://www.eclipse.org

[47] AspectDNG: http://sourceforge.net/projects/aspectdng/

[48] PostSharp: http://gael.fraiteur.net/postsharp.aspx

[49] EOS: http://www.cs.virginia.edu/~eos

[50] RAIL: http://rail.dei.uc.pt

[51] Aspect.NET: http://www.msdnaa.net/curriculum/?id=6219

[52] SourceWeave.NET : http://www.dsg.cs.tcd.ie/index.php?category_id=438

[53] Wicca: http://www1.cs.columbia.edu/~eaddy/wicca/

[54] Compose*: http://composestar.sf.net/

[55] Phx.Morph: http://www.columbia.edu/~me133

[56] AOP goes .NET Community Site http://janus.cs.utwente.nl:8000/twiki/bin/view/AOSDNET/CharacterizationOfExistingApproaches

APPENDIX

A. ASPECT DEFINITION SAMPLE

```
//aspect header, contains aspect name
%aspect Politeness
using System;
using AspectDotNet;

public class Politeness
{
//aspect modules
%modules
    public static void SayHello ()
    {
        Console.WriteLine("Hello");
    }
    public static void SayBye ()
    {
        Console.WriteLine("Bye");
    }
//aspect rules and actions
%rules
    %before %call *
    %action public static void SayHelloAction() { Politeness.SayHello();}
    %after %call *
    %action public static void SayByeAction() { Politeness.SayBye();}
}
```

B. CONVERTED ASPECT SAMPLE

```
namespace Aspect {
    using System;
    using AspectDotNet;

    [AspectDef("Politeness", "MainModule", "")]
    public class Politeness {

        [AspectDef("Politeness", "module", "")]
        public static void SayHello() {
            Console.WriteLine("Hello");

        }

        [AspectDef("Politeness", "module", "")]
        public static void SayBye() {
            Console.WriteLine("Bye");

        }

        [AspectDef("Politeness", "action", "%before %call * ")]
        public static void SayHelloAction() {
            Politeness.SayHello();
        }

        [AspectDef("Politeness", "action", "%after %call * ")]
        public static void SayByeAction() {
            Politeness.SayBye();
        }
    }
}
```

Phalanger: Compiling and Running PHP Applications on the Microsoft .NET Platform

Jan Benda
Charles University in Prague
Malostranske namesti 25
11800 Prague
Czech Republic
jbe@php-compiler.net

Tomas Matousek
Charles University in Prague
Malostranske namesti 25
11800 Prague
Czech Republic
tomas@php-compiler.net

Ladislav Prosek
Charles University in Prague
Malostranske namesti 25
11800 Prague
Czech Republic
lada@php-compiler.net

ABSTRACT

This paper addresses major issues related to compilation of applications written in the PHP language and their solutions proposed and implemented in the Phalanger system targeting the Microsoft .NET platform. Main focus is given to those PHP features that are specific to the interpreted and dynamic nature of this language and that are making the compilation process more challenging. Since a language compiler and runtime are usually tightly coupled, this paper also presents parts of the Phalanger runtime related to the discussed language features. Additionally, the support for various web application execution scenarios within the ASP.NET server is outlined as PHP applications usually target web servers. The effectiveness reached by the compilation to the intermediate language of the .NET platform is demonstrated in a comparison with existing products addressing an optimization of PHP code execution.

Keywords

PHP language, .NET Framework, compiler, web applications

1. INTRODUCTION

The PHP became the most popular interpreted language for web application development due to its ease of use and availability. On the other hand, the interpretation yields sub-optimal performance and also requires presence of the source code on the web server.

This work is not the first one to address these issues. One of today's most common optimizations relies on converting PHP source code units into a binary representation stored in the interpreter cache. The cached binary representation eliminates the need to read the source files and build the structures necessary for their interpretation repeatedly. The *Zend Optimizer* [23] is an example of this approach.

Another approach consists of a translation of the PHP source code into the language whose compiler already exists. Products using this technology are the *Roadsend Compiler* [19], which translates the PHP language to the C language, and recently released *Resin Quercus* [4] whose target language is Java.

Despite these efforts, the Phalanger [9] discussed in this paper still stands as the only existing PHP language compiler [2] with the support for the latest PHP features (version 5.1.2 at the time of writing this paper) and virtually all PHP runtime libraries. It brings the PHP language to the family of the .NET languages [1] and makes it possible for other .NET applications to cooperate with PHP applications regardless of the programming language they are written in. Therefore, the Phalanger enables seamless integration of the existing PHP applications with the new technologies of ASP.NET [10], and thus saving resources that would otherwise be needed for reprogramming them. On the other hand, the .NET programmers can also utilize the advantages of using a dynamic language in their new applications.

This paper is laid out as follows. Section 2 describes how specific PHP language constructs are handled by the Phalanger compiler to achieve high performance of the compiled code. Section 3 outlines the run-time environment provided to the PHP programs compiled

.NET Technologies 2006
Copyright UNION Agency – Science Press,
Plzen, Czech Republic.

by the Phalanger. Section 4 discusses the related works and Section 5 compares them with the Phalanger in a performance benchmark. Finally, Section 6 concludes and outlines the future work.

2. PHP LANGUAGE COMPILATION

The PHP language [3] is a procedural language originally developed to be processed by an interpreter. This is why some features cannot be compiled in a straightforward manner. The challenges of compiling the PHP language and our proposed solutions are presented in this section.

Scripts

A PHP script is a compilation unit in the Phalanger. It consists of snippets of HTML and PHP code one penetrating the other, with the code enclosed in a special type of tags. The pieces of HTML code outside the PHP brackets are treated as if they were printed out by the PHP code via the *echo* statement.

Therefore, from the compiler's point of view the script consists of a sequence of statements. Apart from the statements available in the commonly used procedural languages, function, class and interface declarations are also statements in PHP. Phalanger compiles classes and interfaces into separate CLR types [5]. Functions and other non-declarative statements are compiled into a single static *script type*. This CLR type contains public static methods corresponding to the functions declared in the script and a single public static method containing all the non-declarative statements of the script (the *global code* of the script – the code that is supposed to run when the script is executed).

All code defined explicitly in the script as well as the code created at run-time is executed in a common *script context*. Script context is an object associated with the running script, keeping track of the script state – the defined constants, global variables, functions, classes, script dependent configuration etc. The current script context object is accessible to each user function and method via a reference passed as an argument along the execution path. If the script is running on a web server, the script context object is created for each request and is held by the *request context* object, which contains additional data specific to the request processing.

The following PHP source code sample shows three pieces of global code: '<html>', '$x = 1;' and 'if ($y)', and two declarations, one of them conditional.

```
<html>                            ... HTML snippet
<?                                ... opening script tag
  $x = 1;                         ... global variable assign.
  function f() { }                ... unconditional decl.
  if ($y) { class C { } }         ... conditional decl.
?>                                ... closing script tag
```

Raw structure of the compilation result follows.

```
class C#1 : PhpObject { }
static class ScriptType
{
  public static f(ScriptContext sc) { }
  public static Main(ScriptContext sc)
  {
    sc.Echo("<html>");
    sc.SetVariable("x", 1);
    sc.DeclareFunction("f", f);
    if (Ops.IsTrue(sc.GetVariable("y")))
      sc.DeclareClass("C", typeof(C#1));
  }
}
```

Declarations

Declarations of functions, classes and interfaces stated directly in the global code (i.e. not enclosed in another statement) are *unconditional declarations* (the function declaration in the above example). In addition to this common usage, PHP allows declarations inside a function body, *if* statement block, etc. Such a declaration is a *conditional declaration* (see the class declaration in the example). It depends on run-time conditions whether and when this declaration takes effect.

Once a declaration statement is executed (the declaration becomes *active*) it cannot be undone and a redeclaration is not allowed. However, multiple declarations of the same entity (function, class or interface) using the same name can appear in the code or be defined at run-time provided that at most one becomes active at run-time. Such declarations of an entity are referred to as its *versions* in the Phalanger. Hence, all versions except for at most one must be conditional. Note that if an unconditional version is present the conditional ones shall never be activated. On the other hand, it is not an error to declare them. There are no explicit means for conditional compilation in the PHP language so the regular conditional statements are used for that purpose. Versions are maintained by the Phalanger runtime ensuring that at most one gets activated.

Active versions of functions are stored in the script context in a hash table mapping a function name to an instance of a delegate. The delegate instance represents the CLR method implementing the active version. Another hash table is designated to store the *type objects* representing the active class and interface versions. Each declaration statement adds an entry to the respective hash table at the point of its execution or at the beginning of the global code for unconditional declarations.

A multi-version function call operator then looks up the active version in the table and calls it via the delegate. Analogously, the *new* operator looks up the active version of the type in the hash table and instantiates it. These operators are emitted by the compiler only if the actual target of a function invocation or a class instantiation is not known at compile time. This includes not only the multi-version targets but also targets unknown at compile-time and targets referenced by the name stored in a variable. Otherwise, for the targets known at compile-time, the direct method invocation and class instantiation IL instructions [7] are emitted into the resulting byte code.

Inclusions and Run-time Evaluated Code

The PHP language contains several inclusion statements. Their behavior is almost equivalent to an insertion of the code contained in the included script to the place of the inclusion statement. Implementing the inclusions in this way is undesirable for the compiled language. The compiler processes the individual scripts separately, thus enabling reuse of the compiled modules without the need of repeated processing.

An inclusion whose argument can be determined at compile-time is resolved immediately (*static inclusion*) otherwise the inclusion is deferred to run-time (*dynamic inclusion*). The script included dynamically is bound with the including one at run-time which is, of course, slower than compile-time linking. Unfortunately, many PHP scripts use inclusion expressions that cannot be evaluated at compile-time. The algorithm used by the PHP interpreter for resolving the inclusions makes it even more difficult for the compiler to make the decision at compile-time even if the target is specified by a string literal. By inspecting many existing PHP applications and libraries, we observed that the vast majority of them use only a handful of patterns for specifying the inclusion target. For example, a common pattern is

```
include($AppRoot . "path/to/file.php");
```

where $AppRoot is a PHP variable containing the application root path computed by the previous code and the dot operator performs a string concatenation. Although the expression cannot be evaluated at compile time, the inclusion can be made static. The trick inheres in configuring the compilation of the application so that one or more regular expression patterns are matched against the source code of each inclusion argument to replace the recognized patterns with associated literal constants – the paths relative to the application source root, which is already known to the compiler.

Declarations contained in dynamically included scripts are unknown to the compiler at the time when the including script is being compiled, thus their uses must be compiled as uses of unknown functions, classes or interfaces. Obviously, this presents a problem when declaring a class that inherits from class (or implements an interface) not known at compile-time. In such case, the derived class is treated as unknown despite the fact that its declaration is known to the compiler. This is because the changes in the superclass or implemented interface (which can take place at run-time) can totally change the behavior of any method of the class. In the current version of the Phalanger, such declaration is converted into an *eval* construct that evaluates the source code at run-time. This way, the compilation of the declaration is deferred to run-time at which point all super-classes and implemented interfaces are known. This approach is easy to implement yet is not optimal as the run-time compilation is expensive. The future versions of the Phalanger will emit the declaration in the form independent of the unknown base classes and interfaces where possible.

The behavior of the *eval* construct is similar to the dynamic inclusion. The difference is mostly in the persistence as the *eval*'ed code is compiled into an in-memory dynamic assembly and is not persisted.

Apart from the *eval* construct, there are other constructs and functions that utilize run-time code compilation. Those include the *assert* construct, which evaluates a string containing a PHP expression, the *create_function* library function, which enables the user to define an anonymous (lambda) function with a specified body, and some others. Even though the source code passed to these routines can be created at run-time, it is often not the case and the parameters are usually literal strings. In that case, the compiler processes the literals as if they were regular source codes and immediately generates the IL code during the initial compilation; the compilation at run-time is no longer necessary.

Variables

Global variables are stored in a hash table held by the script context object. Both direct and indirect accesses are thus performed similarly to the original PHP interpreter – using a hash table lookup. There is not much opportunity for optimization here since the global variables can be changed anytime from any function or any script that may be even unknown at compile-time.

On the contrary, the local variables are accessible only within the scope of the function that declares them. Therefore, it is often possible to represent them by the CLR local variables allocated on the stack. This is an important optimization as it is applicable to the vast majority of functions and the creation of the hash table in the function's prologue and the following look-ups are expensive. Nonetheless, in some rare cases the list of local variables and their values needs to be available at run-time. This only happens when a function contains an *eval* construct, a run-time evaluated *assert* construct, an inclusion, a call to a function working with the variable list (e.g. *extract* function), or an indirect function call, which can target the latter. In such cases, a hash table of local variables, which is similar to that of the global ones, has to be created in the function prologue and all uses of the local variables become look-ups in the hash table.

Note that an indirect variable access (access by name) is usually not an obstacle to the optimization of local variables unless there are too many variables used in the function. An indirect access is compiled into a switch over the variable names known at compile-time. Only when the indirectly accessed variable is unknown at compile-time (the default case in the switch is reached) the hash table for the local variables unknown at compile-time is created if it didn't already exist and the local variable is looked up. Therefore, a dynamic access to the variable doesn't necessarily degrade the performance by creating and accessing the hash table.

So far, the compiler doesn't perform any type analysis. Gains of such analysis are very limited due to the nature of the PHP language and are usually not worth the increased complexity of the compiler. Reasoning about the types of the global variables is completely useless as their estimated types can be changed by the code unknown at compile-time. On the other hand, the type inference for local variables might be considered. For example, a local variable controlling the *for*-loop holds usually an integer in the scope of the loop. Nonetheless, effects of such optimizations might be negligible when compared to the expensiveness of run-time code evaluation and other features.

Therefore, each variable is currently either of type *Object* (common super-type of all CLR types) or a special type called *PhpReference*. The latter type is used for variables with aliases, i.e. for those variables that may potentially be used with &-modified assignment operator (by-reference assignment) or that can be passed to a function using by-reference semantics. All global variables are of type *PhpReference* as it is unknown whether they are aliased or not.

In order to cope with PHP references in the way they are used in the language, the *PhpReference* type introduces an additional level of indirection. The type comprises of a single field of type *Object* containing the actual value of the variable.

For example, if two variables are assigned by reference, say $x =& $y, subsequent assignments by value to any of them modifies the other as well. Hence, the assignment $x = 1 changes values of both $x and $y to 1. In compiled code, these variables will be of the type *PhpReference*. The assignment by reference makes them refer to the same instance of the *PhpReference* (the one of $y). The assignment by value assigns to the field of the *PhpReference* instance, so all variables sharing this instance get the same value.

Functions and Methods

User functions are compiled as public static methods of the script type representing the source file that declares the functions. User methods are compiled as methods of the CLR type representing the corresponding user class. Two overloads are generated for each user routine: an *argument-full* implementation and an *argument-less* stub.

The argument-full overload is used by calls whose target is known at compile-time. Its signature includes all user-defined formal arguments. The body contains the compiled code of the routine preceded by a prologue processing arguments and initializing local variables (populating local variables table, checking type hints, etc.).

Contrary to argument-full overloads, all argument-less stubs have the same signature. In many cases a call to a compile-time unknown function needs to be made. Signature uniformity allows delegates of a single type to be used for such calls. The caller pushes the arguments onto an *internal stack* and calls the argument-less stub via the delegate. The task of the stub is to move the actual arguments from the internal stack to the evaluation stack, and call the argument-full implementation. The internal stack is a pre-allocated resizable array residing in the script context.

Object Oriented Features

The PHP language is a class-based object-oriented language supporting run-time modification of the instance fields and some other unusual features. The Phalanger compiler supports the entire object model proposed by PHP5.

PHP classes and interfaces are represented directly by CLR classes and interfaces, respectively, preserving the inheritance hierarchy. The common base class for PHP classes implements much of the PHP specific behavior such as by-name field access and method invocation, instance serialization, dumping, comparison, etc. Compiled PHP classes can be easily reused by other .NET languages. The role of the Phalanger as a consumer and extender of classes produced by other .NET languages is currently limited to cases where such class has been designed for the Phalanger by following several rules related to method signatures, field types and helper methods. These requirements stem from the dynamic and loosely typed nature of the PHP language making late-binding a very frequent phenomenon that should be highly optimized. Being able to directly consume and extend classes produced by other .NET languages would be a great improvement as the whole .NET Class Library and many other libraries would become immediately available to PHP programmers. The solution that features .NET Framework 2.0 Lightweight Code Generation [10] is currently being designed and will be implemented in the next versions of Phalanger.

In the PHP language, instance field declarations are optional. The declared fields are compiled as instance fields of the resulting CLR class and a method giving fast indirect access to these fields is emitted to each class with at least one instance field declared. Instance fields created at run-time are stored in a hash table associated with the instance. Although the compiler is able to discover what fields might possibly be created at run-time, it is incorrect to treat them as if they were declared so, because the semantics of accessing these fields is generally unknown at compile time (for example, a subclass can overload field access by declaring the *__get* and *__set* methods, which consequently turns some undeclared field access operations in its base class to *__get* and *__set* invocations).

When a field is accessed within a method using the *$this* pseudo-variable and the corresponding field is found at compile-time, an IL instruction is emitted to accesses it directly. Otherwise, the lookup has to be deferred to run-time and a call to the run-time operator method is emitted instead. A field access via an ordinary variable is always deferred to run-time because the current version of the compiler doesn't perform any type analysis. Either way, there will always be cases when such field access has to be dynamic.

Method declarations are compiled in a similar way to the functions. There are two ways of invoking methods in the PHP language – virtual and non-virtual. Virtual invocation is denoted by the *$instance→method*(*arguments*) operator, whereas *class*::*method*(*arguments*) operator performs a non-virtual invocation. Both operators can be used to invoke instance as well as static methods. When invoking a static method in the virtual manner, *$instance* is used merely to lookup the method implementation. On the other hand, when an instance method is invoked statically, it is given the call site's *$this* as the instance (if the caller's *$this* is not assignable to the callee's one or the caller has no *$this* at all, a dummy instance is created). Due to the lack of the type analysis, virtual invocation is currently always resolved at run-time via an operator. Non-virtual invocations can be compiled as direct invocations, provided that the class is known at the compile-time.

Some more unusual object features found in the PHP language include the possibility to declare abstract static and final static methods and the possibility to change a member visibility from protected to public by the subclass. In most cases, the Phalanger uses custom attributes to map such features to the CLR.

3. LANGUAGE RUN-TIME

The PHP interpreter provides hundreds of functions to the programmers. These functions can be divided into two main categories:

- *built-in functions* – the most commonly used functions implemented directly by the interpreter
- *external functions* – additional functions implemented in dynamic libraries (.dlls) provided often by third parties.

The Class Library – Built-in Function Set

The *Phalanger Class Library* provides the implementations of the built-in functions and classes. This library is designed to be simply extensible and language independent. The current library functions are implemented in the C# language as public static methods logically grouped to the encapsulating CLR static classes. The semantics of by the PHP functions and classes, required for the use from a PHP code, is added via metadata associated with the respective methods and types. These metadata drive the compiler when it emits calls to the library functions and operations on the library classes.

The Extensions – External Function Set

The external PHP functions are implemented in dynamically linked libraries. These libraries are loaded to the PHP interpreter's address space and communicate with PHP via *Zend API* – a predefined set of functions.

Virtually all PHP extension libraries working against Zend API of PHP 4.3.* are now available to .NET applications via Phalanger's *Extension Manager*. The Extension Manager emulates the original PHP interpreter environment, provides the necessary API to the extensions and bridges the gap between the unmanaged world of the PHP extensions and the managed world of Phalanger in both directions. This solution enables access to the functionality of any PHP extension not only to the PHP scripts, but also to any other .NET language.

The original dynamic libraries are encapsulated by the *managed wrappers*. A managed wrapper is a tool-generated assembly comprising of stubs representing functions and methods provided by the corresponding extension. Each stub marshals its arguments to native PHP structures, performs the call to the PHP extension and unmarshals the results back to the managed form.

Because the PHP extension dynamic libraries do not contain type information, additional hand-written XML files describing function and method signatures are used by the wrapper generator. The generator analyses the dynamic library, adds the type information and emits managed stubs into the resulting assembly. Both versions of the stubs (argument-full and argument-less) are generated to allow indirect calls from the compiled PHP code.

Using the managed wrappers, the native implementations of external functions are completely hidden to the outside managed world so the caller doesn't need to care about the fact that the functionality is actually implemented in the native dynamic library. Hence, the library implementation can be transparently replaced by a managed one anytime without modification of the calling code.

There are two modes of loading PHP extensions using the Extension Manager: *collocated* and *isolated*. The web server administrator may configure individual extensions depending on their reliability preferring either performance or safety.

Trusted extensions may be collocated in the address space of the PHP application process, in the same application domain as the compiled PHP code, leading to much better performance. In this scenario the stubs only convert the managed data to the native PHP structures and back.

Untrustworthy extensions may be loaded into an isolated process. The main process, which executes the compiled PHP code, is then protected from being damaged or even crashed by the code of the unmanaged extension. The two processes communicate via .NET Remoting using the shared memory channel or any other channel type.

ASP.NET Cooperation

Since the PHP scripts usually constitute web applications, the run-time support for the web environment is essential. A PHP web application comprises of the set of scripts and data files stored in a virtual directory on the web server. This directory needs to be configured as an ASP.NET application in order to be managed by the Phalanger. The Phalanger provides a module serving web requests and configures the ASP.NET to use it. The integration with ASP.NET server allows the Phalanger to take advantage of such features as monitoring source code and configuration changes, hierarchical per directory configuration, and sophisticated session handling.

When the request is issued to the Phalanger web application, an object called *request handler* is created to process it. The handler first checks the compilation cache – a directory in which the compiled script assemblies are stored. If the compiled assembly that corresponds to the requested script is found in the cache, it is loaded (unless already in the memory) and executed. Otherwise, the compiler is executed to compile the script and store it in the cache. The response is always generated by the compiled script. If the script is requested frequently, it resides in the memory in a form of just-in-time-compiled native code and the execution is thus really fast as the benchmark results below demonstrate.

The Phalanger also provides an option to pre-compile the entire web application to a single assembly. The request handler then searches the pre-compiled assembly for the requested script's type. By utilizing this scenario, the application source code is not needed any more unless the user requires the Phalanger to watch for its changes. Hence, the web application can be deployed in the compiled form in order to protect the intellectual property in the source code.

The pre-compilation is also essential for large applications comprising of thousands of scripts. That many scripts consume enormous amount of memory if compiled into separate assemblies and then all loaded. Compiling the application to a single assembly makes it more compact and saves the memory.

In cases when an application is pre-compiled, yet the source code still undergoes changes, the Phalanger enables to mark the script types with timestamps so that it can detect changes to the source file during the application execution. The Phalanger maintains the table of invalidated scripts at run-time and recompiles the script into a separate assembly stored in the cache if the script is invalidated.

4. RELATED PRODUCTS

The Phalanger system is one of the few alternatives to the PHP interpreter. The majority of existing PHP web applications is powered by the PHP interpreter alone. If the performance is not sufficient due to high server load, an accelerator is usually added to cache the preprocessed script files. There are many accelerators available today, including the Zend Optimizer [23], the *Turck MMCache* [22] and the *eAccelerator* [3].

Apart from the Phalanger, only two other systems take the approach of compilation. The first is the Roadsend Compiler [19] which compiles the PHP code into the native binaries using the C language as an intermediary. The second is the Resin Quercus [4] targeting Java Virtual Machine by translating PHP source codes into the Java language. The major disadvantage of both is a lack of support for all PHP extensions, which makes these systems currently almost unusable in practice. Additionally, the Roadsend Compiler doesn't currently support the latest versions of the PHP language. The very first beta version of the Quercus has been released several months ago. The system is still under development, and it hasn't been tested on a real-world application yet.

5. BENCHMARKS

The benchmark presented below compares the Phalanger with the versions 5.0.4 and 4.3.11 of the PHP interpreter optionally accelerated by the Zend Optimizer. The benchmark measures the overall performance of the *phpBB message board system* [17] version 2.0.14 by issuing a series of requests that exercise the common operations performed by the message board system users. Since all tested PHP engines use the same database server and the requests are sent sequentially, the benchmark measures the relative differences in the speed of request processing. To measure the results, the benchmark uses the *Microsoft Web Application Stress Tool* [14]. The configuration used for the benchmark was Intel Pentium M 1.4 GHz with 1 GB RAM running Windows XP Professional SP2, IIS 5.1 web server [11] and MSDE 2000 SP3 database engine [12].

Figure 1 visualizes the results of the benchmark. The first three columns show the performance of various Phalanger configurations. The first measurement, the managed MSSQL extension, shows the best results. This extension is a C# reimplementation of the PHP MSSQL extension using Microsoft SQL driver available with the .NET Framework. The second and the third Phalanger configurations exercise the native MSSQL extension shipped with the PHP 4.3.11 interpreter encapsulated in the managed wrapper. The poor result of the third test is caused by isolating the extension into a separate process. Performance degradation is expected in this case since all data transferred between the application and the SQL server has to be passed through .NET Remoting channel connecting the two processes. Therefore, the extension isolation is not appropriate for extensions transferring large amount of data.

The remaining four tests are performed on the PHP interpreter with and without use of the Zend Optimizer. The conclusion of the benchmark is that the most powerful Phalanger configuration improves the performance of the phpBB application by the factor of 2.3 when compared with the best configuration of the PHP interpreter.

Of course, the absolute numbers of the benchmark are not relevant. Series of other benchmarks which varied in the used database server (*Microsoft SQL Server* [12], *MySQL Server* [15]), web server (*Apache* [20], *IIS 6* [11]), particular operations performed on the application as well as benchmarks performed on different applications showed that the version 1.0 of the Phalanger in the configuration with managed extensions makes the request processing about two times faster than the PHP interpreter accelerated by the Zend Optimizer.

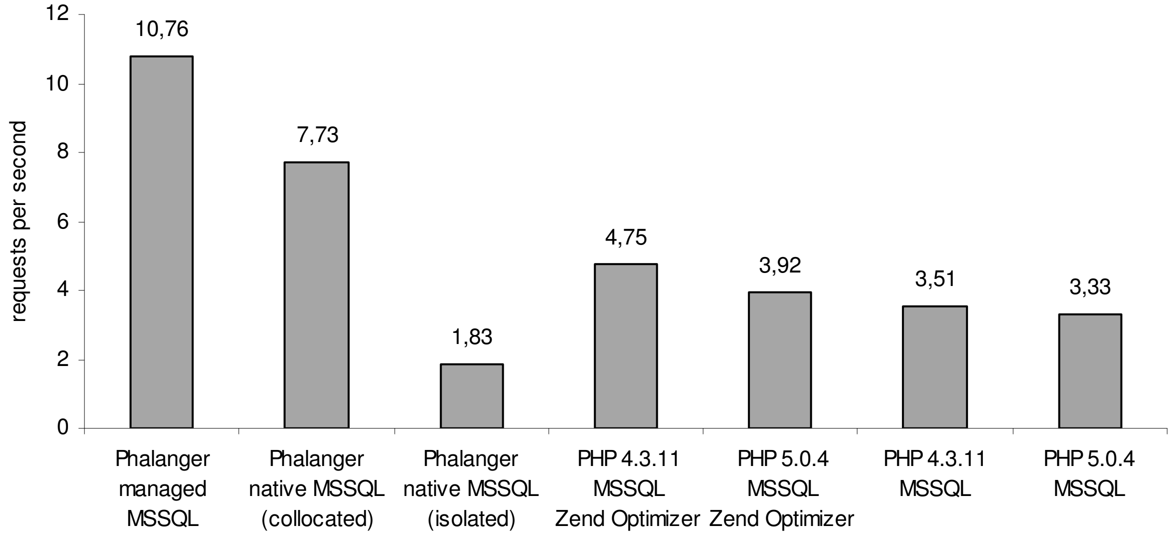


Figure 1. Performance comparison of the phpBB web application running on the Phalanger and the PHP interpreter (not) being accelerated by the Zend Optimizer.

6. CONCLUSION & FUTURE WORK

The Phalanger is a functional tool which allows to deploy existing PHP applications without significant modifications on an ASP.NET web server, increasing the throughput significantly compared to the original PHP interpreter. Phalanger proves that the PHP language compilation targeting the .NET Framework is not only feasible, but even advantageous.

Apart from the demonstrated performance improvements, the Phalanger provides the means for migration of existing PHP applications to the modern web environment of ASP.NET, allows the .NET programmers to utilize useful functionality implemented in the numerous PHP libraries and gives the PHP application developers the ability to access .NET Framework libraries as well as develop their PHP applications inside Microsoft Visual Studio .NET [13].

Another advantage of targeting the .NET Framework over compiling to the native code or to some kind of specific byte-code stems from the amount of work that Microsoft invested to improve the .NET execution engine itself. In general, the performance of applications targeting .NET Framework gets better with the new versions of the .NET run-time. For example, the .NET implementation of the Python scripting language, IronPython, gained a significant increase in performance when migrated from .NET Framework version 1.1 to version 2.0 without any changes to the IronPython scripting engine itself [7]. Further improvements were achieved by utilizing new features of the platform. Phalanger is likely to get the same benefits when ported to the new version of .NET.

The first final version of the Phalanger system has been released recently and dozens of widely used PHP applications and frameworks, including a huge application comprising of about 2000 script files, have been successfully tested on it. The first goal of the Phalanger system, to be able to run the existing PHP4 and PHP5 applications, has been, to a large degree, achieved. However, as the development of new PHP libraries and features (such as Reflection API, Standard PHP Library and features proposed by PHP6) continues, it is necessary to include them in the Phalanger so that the newest versions of the PHP applications continue to run on Phalanger.

The great challenge and the major goal for the next version of the Phalanger is to make the PHP language the first class language of the .NET Framework, i.e. to make all .NET classes accessible directly from the PHP language. The next version of the Phalanger will run on .NET Framework 2.0 which will allow it to use the new features of the .NET engine and make the compiled PHP applications even faster. The Mono platform [16] will also be supported.

REFERENCES

[1] *.NET Languages*: www.dotnetlanguages.net /DNL/Resources.aspx

[2] Aho, A. V., Sethi, R., Ullman, J. D.: *Compilers*, Addison-Wesley, 1986

[3] Alcantara F., Vanbrabrant B., Tabary, F.: *eAccelerator*, eaccelerator.net

[4] Caucho Technology, Inc.: *Resin Quercus,* www.caucho.com/resin-3.0

[5] ECMA: *Common Language Infrastructure*, msdn.microsoft.com/net/ecma

[6] Gough, J.: *Compiling for the .NET Common Language Runtime*, Prentice Hall, 2001

[7] Hugunin, J.: *IronPython: A fast Python implementation for .NET and Mono*, PyCON 2004, python.org/pycon/dc2004/papers/9

[8] Lidin, S.: *Inside Microsoft .NET IL Assembler*, Microsoft Press, 2002

[9] Matousek, T., Prosek, L., Novak, V., Novak, P., Benda, J., Maly, M.: *Phalanger*, www.php-compiler.net

[10] Microsoft: *.NET Framework Platform* www.microsoft.com/net

[11] Microsoft: *Internet Information Services*, www.microsoft.com/iis

[12] Microsoft: *SQL Server*, www.microsoft.com/sql

[13] Microsoft: *Visual Studio .NET*, www.microsoft.com/vstudio

[14] Microsoft: *Web Application Stress Tool*, www.microsoft.com/technet/archive /itsolutions/intranet/downloads/webstres.mspx

[15] MySQL AB: *MySQL Server*, www.mysql.com

[16] Novell and contributors: *Mono Project*, www.mono-project.com

[17] phpBB Group: *phpBB*, www.phpbb.com

[18] Richter, J.: *Applied Microsoft .NET Framework Programming*, Microsoft Press, 2002

[19] Roadsend, Inc.: *Roadsend Compiler*, www.roadsend.com

[20] The Apache Software Foundation: *Apache HTTP Server Project*, httpd.apache.org

[21] The PHP Documentation Group: *PHP Manual*, www.php.net/manual

[22] Turck Software: *Turck MMCache*, turck-mmcache.sourceforge.net

[23] Zend, Inc.: *Zend Platform*: www.zend.com/products/zend_platform

MVE-2 Applied in Education Process

Milan FRANK
University of West Bohemia
Univerzitní 8, BOX 314
Pilsen, Czech Republic
mfrank@kiv.zcu.cz

Libor VÁŠA
University of West Bohemia
Univerzitní 8, BOX 314
Pilsen, Czech Republic
lvasa@kiv.zcu.cz

Václav SKALA
University of West Bohemia
Univerzitní 8, BOX 314
Pilsen, Czech Republic
skala@kiv.zcu.cz

ABSTRACT

For years we have been developing a our project on MVE. MVE stands for Modular Visualization Environment. It is a user friendly modular environment using data flow paradigm for communication between user-created modules. The core of the system is based on pure .NET technology.

We find this environment useful in several application areas. This paper focuses on successful employment within the education process. We believe that MVE-2 can be a good entry point for programmers to learn how to develop plugin style software components and to cooperation between them.

This paper also discusses advantages of modern programming techniques available in .NET. We have found several .NET features very useful during design and development of the environment.

Keywords
.NET, MVE, Visualization, Plugin, Data Flow, Pipeline

1. INTRODUCTION

MVE-2 is our grass root effort to create a general and easy to use modular environment. It uses pipeline approach for problem decomposition. This paradigm is useful for both theoretical and practical purposes. Engaging our system in education leads our students naturally to perform clear and well defined problem decomposition as well as to follow good programming habits.

We have used the MVE-2 in the frame of subjects taught at University of West Bohemia (UWB), in separate student projects and as a tool for real research projects. The environment proved itself useful in all the previously mentioned application areas.

The fact that we started the project from scratch allowed us to choose whatever technology available. Our choice of .NET as a core technology provided us with numerous features, which allowed a solid and elegant design of the system.

.NET Technologies 2006

Copyright UNION Agency – Science Press,
Plzen, Czech Republic

The rest of the paper is organized as follows: Brief overview of the history of our visualization environment development is given in the following subsection. Section 2 gives basic description of the MVE-2 architecture and its differences from similar systems. Section 3 focuses on the ways students have contributed to the development and expansion of the system. Section 4 describes how MVE-2 is useful for our scientific effort.

1.1 MVE History

The idea of developing a new modular environment at UWB started in 1996 as a diploma thesis of Martin Roušal. This original MVE system was based on the Win32 API. The primary focus on visualization tasks and the choice of development environment lead to several drawbacks of the original design, such as fixed set of datatypes, simple pipeline execution, explicit memory management and problems with components created with different programming tools. This environment was used for several years in both education and research applications.

In year 2004 the Center of Computer Graphics and Data Visualization (CCGDV) has decided to

Project supported by Microsoft Research 2003-178
6FP EU 3DTV No 511568

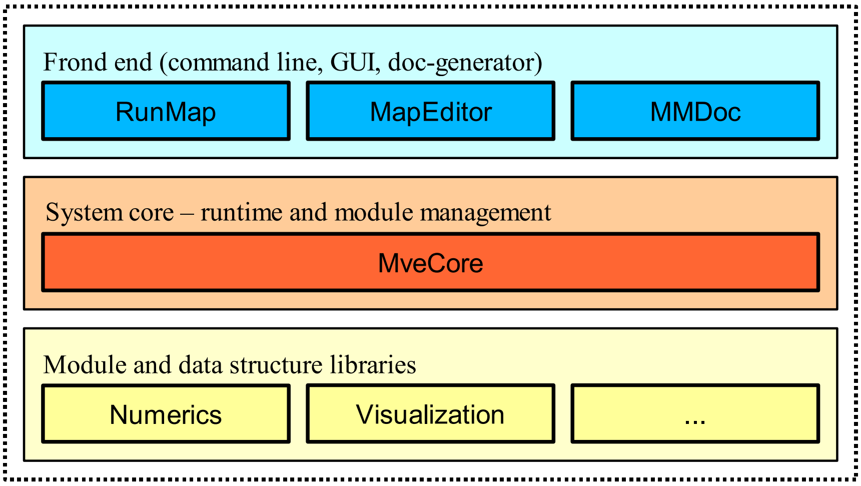

Frond end (command line, GUI, doc-generator)
RunMap
MapEditor
MMDoc
System core – runtime and module management
MveCore
Module and data structure libraries
Numerics
Visualization
...

Figure 1: This figure illustrates a general structure of MVE-project. Each sub-block represent one .NET assembly.

implement a new redesigned version of MVE. Based on extensive experience with the previous version, a new set of requirements has been set. The core of the system was designed and implemented by Milan Frank and his small team of MSc. students with supervision of prof. Václav Skala. The common set of data structures and basic modules for data visualization and computer graphics has been subsequently implemented by the team.

Furthermore, the MVE-2 development spreads beyond the boundaries of the core team as the system was used in subjects provided by the CCGDV for research purposes and in other areas.

2. SPECIFICATION

MVE-2 offers easy-to-use, data-flow based modular environment. Its primary users are researchers and students together with their projects. Our environment makes these projects compatible with each other with minimum additional effort. API (Application Program Interface) of a module is enforcing good programming habits, such as clear problem decomposition, precise comment writing and cooperation with other programmers. Using MVE-2 leads to less routine programming due to compatibility and reusability of existing modules. Therefore, users can concentrate on their particular problem and employ existing modules for marginal tasks.

When we were designing the MVE-2 system, we have attempted to achieve several goals. We wanted to create a well defined and understandable module API with following important features of the whole system:

- general core, ready for modules and data types from different application areas,
- module-maps with support for cycles and sub-branches,
- intuitive and friendly API for modules and data structures,
- automatic generation of basic GUI of a module,
- automatic generation of documentation for module libraries,
- built in XML export/import of all data types and module-maps. (very efficient way to check and modify data manually).

2.1 Core

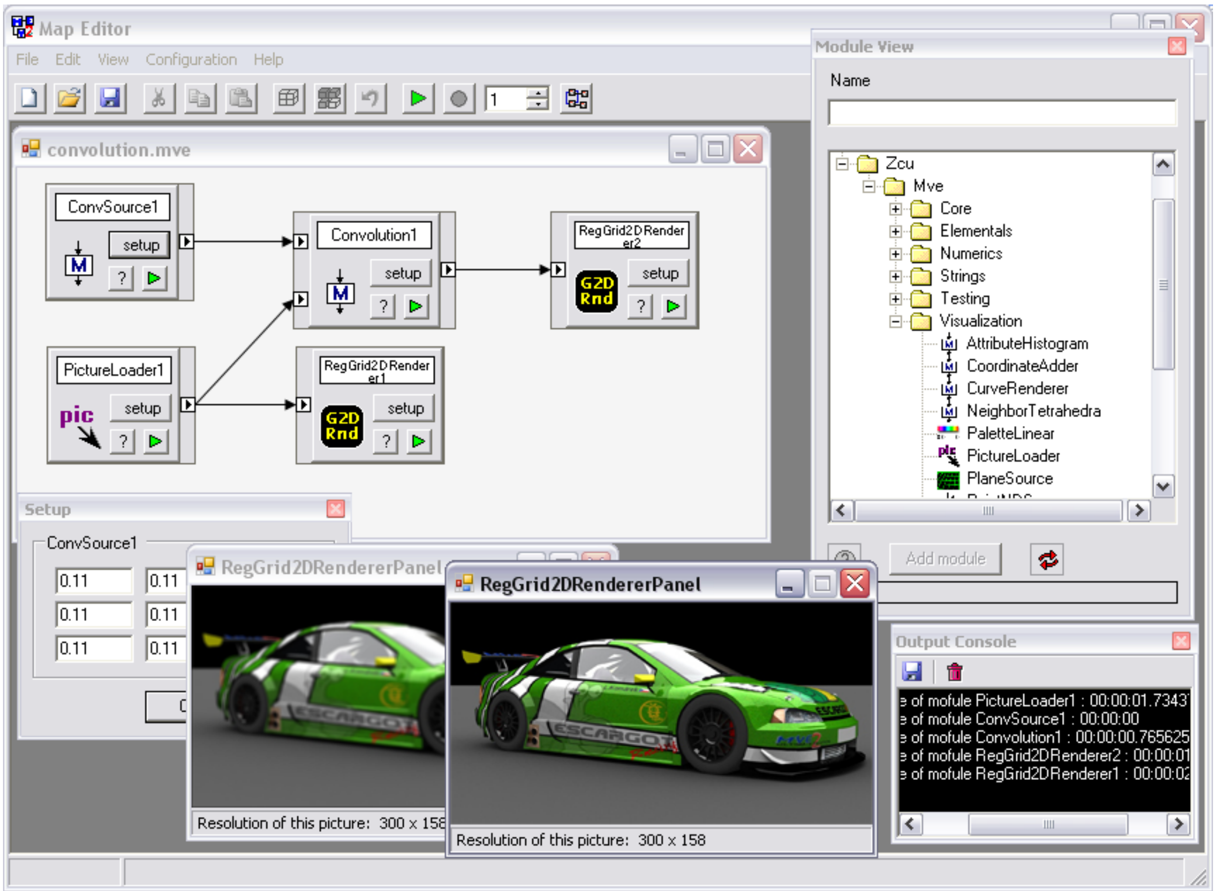
Main part of the environment is MveCore. It provides runtime and module management functionality. Capabilities of the core are accessed by two front-ends, a graphical interface that allows module map composition and execution, Map Editor, and a command line tool for executing existing maps stored in XML files.

One of the main advantages of the system is very simple implementation of a plain module. As well as this, power and high flexibility is available if required. This advantage is especially important for programmers at the beginning of their career. Another specialty of MVE-2 is execution mechanism of module network. Possibilities of module map topology are far beyond simple pipeline and reach closely towards complete visual programming. Many other advantages arise from use of pure managed environment (.NET).

2.2 MVE Front-end

MapEditor is a GUI front end of MVE-2. It allows intuitive module map editing, module configuration and execution. Figure 2 shows a screenshot of the GUI with a simple convolution application.

Located in the upper left corner is a module-map edit window with a simple pipeline using two sources (PictureLoader, ConvolutionMask) and two sinks (RegGrid2DRenderer). In the upper right corner there



ConvSource1

0.11 0.11
0.11 0.11
0.11 0.11

```
e of mofule PictureLoader1 : 00:00:01.7343
e of mofule ConvSource1 : 00:00:00
e of mofule Convolution1 : 00:00:00.765625
e of mofule RegGrid2DRenderer2 : 00:00:01
e of mofule RegGrid2DRenderer1 : 00:00:02
```

Figure 2: Screenshot of MapEditor. The GUI front end of MVE2. It shows a simple convolution scheme.

is a ModuleView dialog that contains list of available modules that are ordered according to namespaces. The standard output is redirected to the output console, which usually displays important messages from modules and core. In the example it displays the running times of modules. The two green cars in the center are the original and the filtered image rendered by renderers. In the bottom left corner a setup dialog of convolution source module is shown. User can define the convolution mask via this dialog.

2.3 Module libraries

The core and the front-end only create an empty space for modules. Without modules there is no useful functionality. Modules can be added into the MVE-2 system very easily. It is sufficient to copy an assembly (a .NET DLL) into a particular directory. All public subclasses of the MveCore.Module class are interpreted as modules.

Each assembly can also provide subclasses of MveCore.DataObject. It allows everyone to define their own custom data types that can be passed between modules via connections.

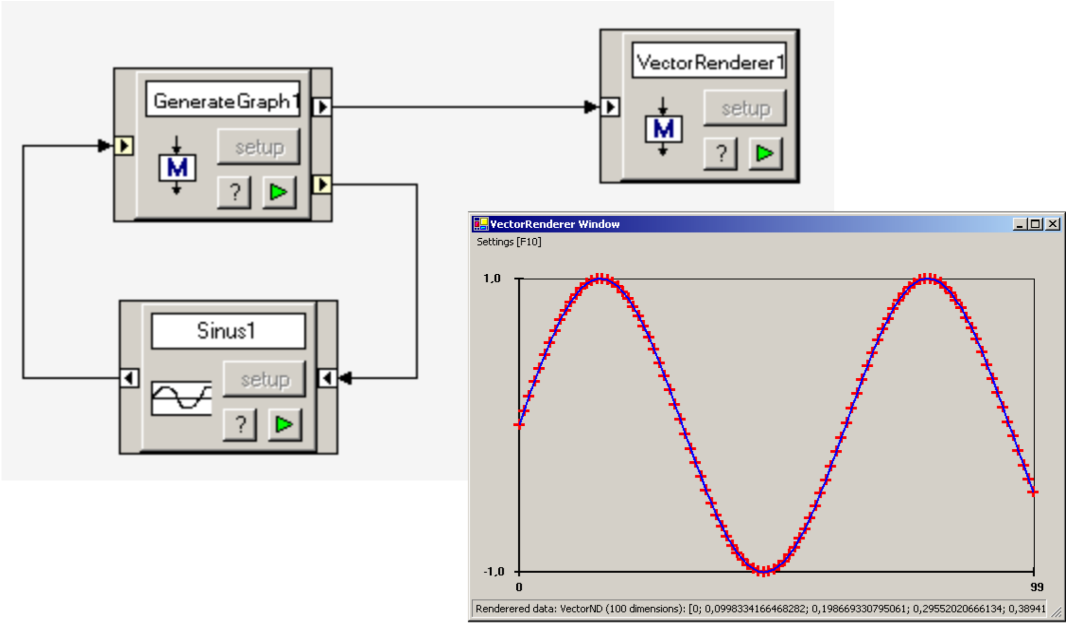
The assembly can contain (or call) any other code allowed by the .NET standard. If one has a project already written in the .NET environment, then it is usually very easy to provide a set of MVE-2 modules as wrappers for functionality of the code. These modules can serve as a "standard" interface and thus can be easily reused by many other researchers and developers.

2.4 Advanced pipeline examples

Execution mechanism implemented in the core can do more than simple pipeline execution. We support repeated execution, module driven execution and cycles. Any map can be run N-times. Module can run a subbranch to obtain its input data multiple times. It is also possible to create cycles in module map graph. See following examples:

Sinus (See Figure 3) is an example of sub-branch construction. Execution of Sinus module is controlled by GenerateGraph module. In this particular case the whole module map runs only once, while the Sinus module runs 100 times.

Counter (See Figure 4) is an example of DelayModule usage. The DelayModule acts as a single place memory with initialization. It returns data form previous (N-1) step. In the first step it returns data from initialization port. Thus it allows cycles in



```
Renderered data: VectorND (100 dimensions): [0; 0,0998334166468282; 0,198669330795061; 0,2955202066134; 0,38941
```

Figure 3: Simple example of module driven subbranch execution.

module-map graph. This example counts from zero to number of runs minus one. The DelayModules can be chained.

2.5 Module creation

As mentioned in the begging of the text, creation of a module is very simple. It is based on inheritance mechanism. Every subclass of MveCore.Module is interpreted by the MVE-2 core as a module.

There are only two methods that have to be subject to override. The first one is the constructor, which creates input and output ports and defines their names and accepted data types. The second one is the Execute method that represents the activity of the module.

The standard .NET property mechanism allows the module authors to easily provide configurable parameters of their modules. Every public read/write property of a standard datatype is automatically saved into and restored from the module map XML file, and it is also shown in a module GUI that is automatically generated for each module. These features are provided by the Module superclass, and don't require the user to write a single line of code.

There is also a set of advanced methods that can be called and set of events that can be handled by a module. These additional methods provide a possibility to create a module with advanced features, such as immediate reaction to incoming data, advanced module GUI creation, execution of a subbranch etc. This means good flexibility for a

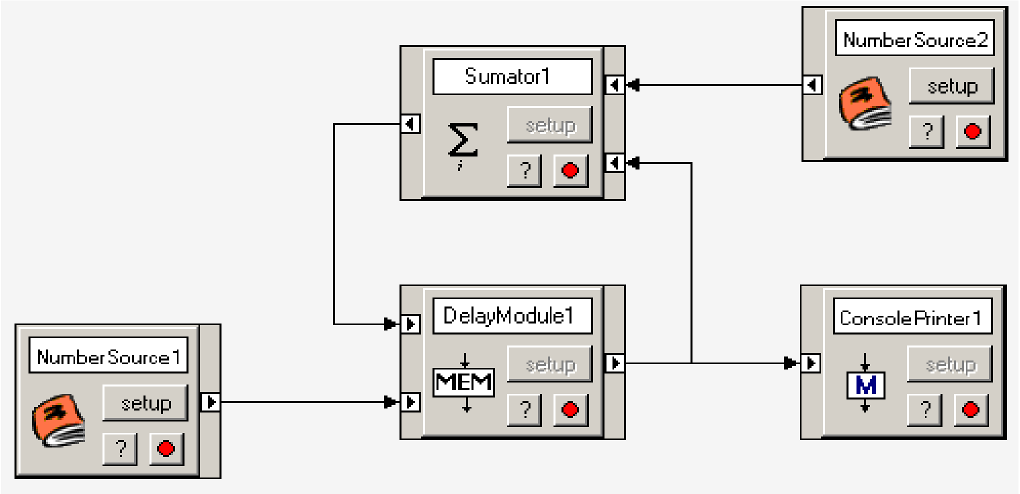


Figure 4: Simple example of loop with delay module.

```
public class Sinus : Zcu.Mve.Core.Module
{
  ScalarNumber y = new ScalarNumber();
  public Sinus()
  {
    AddInPort("in", typeof(ScalarNumber));
    AddOutPort("out", typeof(ScalarNumber));
  }

  public override void Execute()
  {
    ScalarNumber x = (ScalarNumber) GetInput("in");
    y.Val = Math.Sin(x.Val);
    SetOutput("out", y);
  }
}
```

Figure 5: Example of a simple module implementation. Note this is just a class derived from Core.Module class. It is than interpreted by core as a module.

module. Fortunately, in the beginning there is no need to even know about these methods.

Example of implementation of a simple module that calculates sinus of input value follows. Please note this is a complete C# source code one needs to create a MVE2 module. See Figure 5.

Creation of data type is similarly easy, only Core.DataObject is used instead Core.Module.

Documentation of a module library can be generated automatically by the MMDoc utility that is distributed with MVE-2 system. It uses the .NET attribute mechanism to obtain additional information from each module and data type, which describes the module behavior. This information includes description of module ports and configuration properties as well as general description of the task that is performed by the module. This information is also used by the GUI front-end to provide the user basic information about the modules in help dialogs and pop-ups.

3. MVE-2 IN EDUCATION PROCESS

There are two main ways how students get in touch with the system. Many students were asked to create modules to be integrated with the system and with one another. Small group of students was also involved at the development of the system itself and its peripheral tools, such as GUI frond end, automated documentation system, data structures and so on.

3.1 Student Contribution to the Core Development

Several volunteer MSc. and Bc. students were involved in the development of the core of the system. They were cooperating closely with a current project leader. Such involvement gave them feedback about their work and made them familiar with a developing model typical for small software companies. We believe it is a useful experience in a career of a programmer and will help them in seek for future employment.

Miroslav Fuksa was the first volunteer to be involved in the development. His contribution to the execution mechanism was very inspiring. For one developer it is not easy to keep in mind all the possibilities of such a complex algorithm as the module execution mechanism.

A huge contribution has been made by Zdeněk Češka. He is fully responsible for development of the GUI front end. Design of such complex subsystem gave him good practical experience about how to apply theoretical knowledge obtained in subjects of software engineering and knowledge of programming in .NET environment.

The whole of MMDoc subsystem was designed and implemented by Petr Dvořák. He also created a useful GUI front end of this subsystem. Such task made him familiar with several modern technologies such as .NET, XML, XSLT, CHM, etc. He proved himself to be able to apply such technologies in a real world application.

Very important task was to design and implement common data structures for data visualization. Miroslav Vavruška did significant contribution to this essential part of MVE.

Přemek Zítka was responsible for adding a useful feature. Thanks to his effort it is now possible to use automatically generated module GUI (setup dialog).

Using the standard .NET Framework PropertyGrid component it was possible to expose public properties of a module in a simple and elegant way.

3.2 Student Contribution to the Module Library Development

In the frame of Computer Graphics and Data Visualization subject taught at the UWB students were supposed to implement several modules that solve a particular task. These tasks included mesh displacement, elevation coloring, triangle mesh reduction, readers of several triangular formats, etc.

The results of their effort were interesting sets of modules. They were also supposed to provide a detailed documentation for their module libraries.

We believe such task give the student a basic idea about how to write useful pieces of code that can be integrated in some larger systems.

For example: Task chosen by student Jan Bárta was to create a reader and writer module for several standard geometry data files such as PLY, STL, TRI, CMX. Result of his work is clearly very useful and reusable by many other people.

Another nice task was to create a set of modules for generating a displacement mesh from a 2D picture. Jiří Skála took this work very seriously and the result of his effort is an example of what a module library should look like.

Mesh smooth and displace modules were designed and implemented by a team of Ondřej Kvasnička and Martin Pokorný. Their task was to create a set of modules to produce a mesh distorted by the intensity of applied texture. It was necessary to divide this task into several modules. It was interesting to see their feedback about how MVE helped them with the problem decomposition (and following composition) and programming cooperation.

Most of these modules are freely available at the MVE-2 website.

4. MVE-2 IN RESEARCH

As MVE-2 became stable it was employed in a number of real research projects that are carried by CCGDV PhD. students. This lead to benefits for all involved parties, MVE-2 has gained some useful modules, core developers received feedback about the performance of the core and researchers benefited from a easy to use and powerful tool for their projects, which allowed easier collaboration and code sharing.

So far three research topics were addressed using MVE-2: stripification of triangular models by Petr Vaněček, artificial hologram rendering and reconstruction by Martin Janda and Ivo Hanák, and space-time metric for dynamic mesh comparison by Libor Váša.

The first project carried by Petr Vaneček showed some performance drawbacks of the original data structures that were fixed in subsequent versions of MVE-2 by optimization of the visualization structures. Thanks to the efforts of Dr. Vaneček there is a fully functional and thoroughly tested support for triangle stripes and fans in the visualisation library provided with MVE-2.

The second project was the spatio-temporal metric implementation for dynamic mesh comparison by Libor Váša. This effort has benefited greatly from the available range of modules, while some more common functions were added to the visualization library. This allowed wide testing of the proposed algorithms on several kinds of source data, using loaders for various data types, and also the result was visualized using the modules provided by MVE-2. This project will continue using MVE-2 in the future in order to allow international cooperation with foreign universities that participate in providing source data.

The most recent application of MVE-2 in the research field is in the artificial holography research that is carried by Martin Janda and Ivo Hanak. They are developing modules for rendering scenes into artificial holograms and computer reconstruction of holograms. The environment allows this elementary team to cooperate easily, as each researcher works on his own module, while having a clearly defined interface to each other. They have also contributed to the core of the MVE-2 with some minor changes that improved the usability of the system for their specific purposes.

5. FUTURE DEVELOPMENT PLANS

Although the core of the system is not being actively developed anymore, the project is still growing by additions of modules and features into the MapEditor GUI. One of CCGDV MSc. students is currently working on a general rendering module that will utilise the D3DUT [4] for rendering. This rendering library developed also by CCGDV will allow platform independent rendering, which will enable full visualization pipelines on all platforms that allow compilation of the system and D3DUT.

In the near future, we would like to investigate the rewriting an area of the visualisation library so that it will utilize the new features of .NET 2.0. The generic data types of .NET 2.0 can be very useful and can simplify some algorithms quite significantly.

The system will be most likely be used in the data visualisation subject taught at the UWB, where students will contribute, develop and test modules as a part of their coursework. The system will also be used as a target platform for computer graphics related diploma theses.

The environment will be used by the holography and dynamic mesh researchers, who will contribute feedback and new modules to the system. This can give the users of the system the advantage of availability of state of the art algorithms within the environment.

The further in the future, our development plans include allowing parallel and distributed execution of module maps. Module libraries for volume data rendering and computational geometry tasks would also greatly improve the practical usability of the system, and we are currently looking for contributors or student leaders to develop such functionality for MVE-2.

6. CONCLUSION

We have described a modular system that is developed at UWB. Students at all levels of education have contributed to the system, which allowed them to learn a valuable lesson about modular programming.

The system is currently used in number of student and research projects, where the structure of the environment helps to clearly formulate and thus easier solve various kinds of problems.

The system uses .NET environment at its best. It enabled the system designers to implement desirable features, such as editable module properties, in a way that is not matched by any similar system in its elegance and simplicity.

7. ACKNOWLEDGMENTS

The authors of the paper would like to thank all previously mentioned contributors for their effort to improve MVE-2. We also thank to Angharad Savage for her careful spell checking of this paper.

8. REFERENCES

1. Schreder, W., Avila, L., Martin, K., Hoffman, W., Law, C.: *The VTK User's Guide.* Prentice Hall, New Jersey, 2001.
2. Váša, L., Skala, V.: A spatio-temporal metrics for dynamic mesh comparison. Subbmitted to AMDO 2006
3. Frank, M., Váša, L., Skala, V.: Pipeline approach used for recognition of dynamic meshes. Submitted to 3IA Limoges 2006
4. Home pages of D3DUT http://herakles.zcu.cz/research/d3dut/
5. *Home pages of VTK.* http://public.kitware.com/vtk/
6. *Home pages of MVE-2.* http://herakles.zcu.cz/research/mve2/

Using the .NET Profiler API to Collect Object Instances for Constraint Evaluation

Dave Arnold
School of Computer Science
Carleton University
1125 Colonel By Drive
Canada K1S 5B6, Ottawa, ON
darnold@scs.carleton.ca

Jean-Pierre Corriveau
School of Computer Science
Carleton University
1125 Colonel By Drive
Canada K1S 5B6, Ottawa, ON
jeanpier@scs.carleton.ca

ABSTRACT

Evaluating software based constraints at runtime is an important task for both the validation and verification of software. It is not uncommon to encounter constraints that require obtaining the set of all active object instances for a given classifier. When the application under test is being executed on a virtual machine or a managed runtime, it is often difficult, if not impossible, to obtain such a set. We will examine Microsoft's .NET common language runtime, and through the use of the profiler API, provide a concrete mechanism for obtaining the set of live object instances for a given classifier. We will then leverage this set to provide an extension to an existing C# and Object Constraint Language compiler to support the OclAny::allInstances operation.

Keywords

C#, Constraints, Profiler, OCL

1. INTRODUCTION

Software based constraints provide a mechanism for testing software. Such constraints can be expressed using a formal language such as the Object Management Group's Object Constraint Language (OCL) [Omg03a]. Constraints are expressed at the model level in the form of preconditions, postconditions and class invariants [Fra03a, War03a]. In our work, when a model is used to generate source code, the constraints are translated from the model level to the code level. The source code is then compiled through the use of a specialized compiler [Arn04a] to generate executable code. In the case of the C# programming language [Hew02a], this code is executed by Microsoft's Common Language Runtime (CLR) [Hew02b]. The CLR is not a virtual machine, but rather an execution engine. The CLR provides memory management for both allocation and garbage collection. As the CLR abstracts memory management away from the programmer, it is difficult to determine which object instances are allocated and active. The CLR does not provide any feedback to the application being executed about the state of the application's memory. That is, there is no way to determine the object instance information for a given classifier within the containing application.

.NET Technologies 2006
Copyright UNION Agency – Science Press,
Plzen, Czech Republic.

Context

Our paper will present an approach for accessing memory management information from the CLR via the .NET Profiler API [Mic02a]. Our approach will track each object instance of a given classifier from allocation through to garbage collection. We will demonstrate that our profiler, on request from the application being executed, can return the set of all object instances for a given classifier. The object instance set can then be used for various activities including the evaluation of software based constraints.

Organization

The remainder of this paper is organized as follows: Section 2 provides a brief background on the CLR’s memory management and garbage collection algorithms. Section 3 presents an unmanaged Component Object Model (COM) component that implements the .NET Profiler API to interface with the CLR and respond to memory allocation events. Section 4 examines how the unmanaged COM component can exchange information with the managed application being profiled. Section 5 provides a concrete use of our method by implementing the previously missing OclAny::allInstances operation in an existing C#/OCL compiler [Arn04a]. Finally, Section 6 presents concluding remarks and areas for future work.

2. CLR MEMORY MANAGEMENT

Instances of classifiers in .NET are allocated from a section of memory known as the managed heap [Stu03a]. The heap is managed because after an application requests memory, the garbage collector takes care of the cleanup. Object instances can be small, containing a few integers, or larger, for example holding a database connection with an extensive amount of state information. Object instances can be self-contained or reference other object instances. The role of the garbage collector is to determine when objects should be collected to free memory for other allocations. The garbage collector fills its role by selecting the object instances that can be deleted. Garbage collection is performed when an application attempts to allocate memory from the managed heap, and the managed heap is too full to complete the request. Managed heaps in the CLR are periodically renewed by identifying dead objects and then fusing contiguous runs of dead objects into blocks of memory to be reallocated. The method used for discovering dead objects is called tracing. Tracing is accomplished by following live references to objects in the managed heap. Once a live reference is encountered it is marked. The garbage collector can then easily determine that any object instance that is not marked can be reclaimed. Live objects are located by looking for heap pointers on all the stacks, in all statically allocated memory, within all object instances, and in a few other CLR data structures [Stu03a]. When a live object is located, the memory that the object points to is examined for additional references (pointers). If more are found they are likewise followed until the entire set of live objects is known. The action of determining the live object set is called "tracing the roots", and results in the transitive closure of the set of live objects.

The approach to garbage collection described in the previous paragraph is known as "mark and sweep" collection [Stu03a]. The problem with pure mark and sweep collection is that over time the managed heap becomes fragmented. To avoid heap fragmentation, "compacting collection" is used. Compacting collection removes dead objects and pockets of unallocated memory by sliding live objects down towards the low-address end of each heap segment, and then repairing any dangling pointers with updated values. Such compaction also has the positive side effect of maintaining object creation order, which improves locality of reference.

The expenses of all the object movement can be reduced drastically via an enhancement used by the CLR's garbage collector known as "generational collection" [Stu03a]. When a generational approach is used, object instances are initially allocated in the youngest generation. If they survive past a garbage collection cycle, they are promoted to an elder generation by copying. The refinement of this method over compacting collection is that object instances that are located in the younger generation generally have a shorter survival rate, while objects in the elder generation have a higher survival rate. As object instances are split into different managed heaps, different techniques are used to reclaim memory. The CLR uses a non-copying, non-compacting collector for the elder generations. In the youngest generation, a copying approach is used. The CLR garbage collector is triggered by allocation volume or memory scarcity; when heap resources run low, the roots are traced, and either one or both generations are scavenged for memory. For details on how the CLR garbage collector is implemented see [Stu03a].

Garbage collection is well worth the complexity and the effort [Hil03a]. Garbage collection provides additional application reliability and programmer productivity. However, since garbage collection can be triggered without notice and because the managed application is not notified when garbage collection takes place, it is hard to determine which object instances are active at a given point during execution. We will now examine a method for accessing the managed heaps directly to extract the necessary object instance information.

3. THE PROFILER API

In order to obtain the set of all live object instances, it is obvious that we require a way to get inside the CLR and examine the managed heaps. Unfortunately, the only way to implement such functionality would require modifying the CLR itself. But, modifying the CLR is not a practical solution. Fortunately, for our purposes Microsoft has provided a back door into the inner workings of the CLR. This back door is the profiler API [Hil03a]. The profiler API allows for an external COM component to monitor the execution and memory usage of an application running under the CLR. Normally, the profiler monitors the running application and does not interfere with it. In our approach, we will leverage the profiler API to monitor object instance allocation and garbage collection, and we will return this information to the managed application.

The profiling APIs are implemented via two COM interfaces. One of the interfaces is implemented by the CLR (ICorProfilerInfo), and the other is implemented by the profiler itself (ICorProfilerCallback). The ICoreProfilerCallback interface receives notification from the CLR regarding various events that occur during a managed application's execution. The ICorProfilerInfo interface extracts additional information from the CLR itself.

Initialization

The CLR connects with one profiler at most during its initialization phase [Hil03a]. The profiler must use the Initialize method defined in the ICorProfilerCallback interface to save the ICorProfilerInfo interface pointer so that it can be used to retrieve additional information from the CLR during actual profiling activities. The Initialize method must also register for CLR events that the profiler is interested in. The ICorProfilerCallback interface supports approximately sixty CLR events. To reduce the amount of overhead introduced by the profiler, the profiler specifies which events it is interested in. For our task, we wish to be notified when a new object instance is allocated, when the garbage collector is invoked, and finally we need to be notified when an object reference (pointer) is moved. The last event is required because we will be maintaining a set of pointers to the actual object instance memory locations. Table 1 illustrates the profiler event bit masks we are using. For our purposes of object instance collection and tracking, we only need to implement four of the sixty ICorProfilerCallback events. The following sections will describe each of the methods, and their corresponding implementation details.

Event Mask	Meaning
COR_PRF_MONITOR_ SUSPENDS	GC Notification
COR_PRF_MONITOR_GC	GC Calls
COR_PRF_ENABLE_OBJECT_ ALLOCATED	Object Allocation
COR_PRF_MONITOR_OBJECT_ ALLOCATED	Object Allocation

Table 1. Select Profiling Events

ObjectAllocated

The ObjectAllocated method is invoked by the CLR each and every time memory in the managed heap has been allocated for an object [Hil03a]. The method provides two parameters; the first parameter is a pointer to the managed heap location where the newly allocated object instance is being stored: objectId. The second parameter is a pointer to the class descriptor for the objectId: classId.

Our implementation is fairly straightforward: the class descriptor is used along with the previously saved ICorProfilerInfo interface pointer to determine the classifier name. The classifier name is then compared against the set of given classifier names that we are "interested" in. A classifier becomes interesting when the application being profiled notifies us that we will be asked for the classifier's object instance set. Details of how this notification works will be provided in Section 4; for now it suffices that we are only interested in a subset of the list of classifiers. In an effort to reduce the resources needed by the profiler, rather than store all of the object instance information, only instance information for classifiers that are deemed to be interesting is stored.

MovedReferences

The MovedReferences method is invoked by the CLR to notify the profiler that the garbage collector has moved one or more object instance locations [Hil03a]. When this occurs, the objectIds provided by the ObjectAllocated method are no longer valid, as they may no longer point to the correct location within the managed heap. It should be noted, that the object's internal state does not change, just its location within the managed heap. In the context of our profiler, all we are doing is updating our internal arrays to reflect the movements.

ObjectReferences

The ObjectReferences method is called by the CLR once for each object instance that remains in the managed heap after a garbage collection operation has completed [Hil03a]. We use the ObjectReferences method to mark the object instances as un-collected, and the object instances are still kept inside our array of objectIds.

RuntimeSuspendFinished

The CLR calls RuntimeSuspendFinished to notify the profiler that the CLR has suspended all of the threads needed for execution suspension [Hil03a]. One of the reasons for runtime suspension is garbage collection. As the ObjectReferences method will be called for each object instance that survives when the runtime is suspended, we will mark each of the tracked object instances as collected. When the ObjectReferences method calls are complete, the object instances that have survived the garbage collection will be un-collected. Our array will then only contain the objectIds of object instances that are still live.

Intuitively, this may seem like a bad idea because there will be a delay between when we mark all the object instances as collected, and when we realize that a given object instance is live, and needs to be un-collected. The delay is not a problem because the

CLR guarantees all of the ObjectReferences calls will be performed before the CLR's execution threads are restarted.

Summary

By providing a specialized implementation for the preceding four ICorProfilerCallback methods our profiler is able to maintain an internal array of pointers for each instance of the interesting classifiers. Each pointer is a reference to a live object instance on the managed heap. During garbage collection, the runtime is suspended and each object instance we are tracking is marked as collected. Before the runtime is restarted, the CLR provides notification for each object instance that has survived garbage collection. We are then able to un-collect the object instance pointers stored in the array. Finally, should the garbage collector compact the managed heap, our profiler will receive notification so that the heap pointers can be updated accordingly.

We have now explained a COM component that interfaces with the CLR. The component registers for object allocation and garbage collection events. The events are used to maintain an array of currently live object instances for interesting classifiers. The next task, presented in Section 4, is to provide a managed interface into the COM component so that the managed application, which is being profiled, can register its classifiers as being interesting and access the object instance pointer array.

4. GETTING OBJECT INSTANCES

As our COM component is loaded and initialized by the CLR running in an unmanaged memory space, the COM component is unable to call methods that are located inside the managed application. However, managed applications can invoke methods that are exported by a COM component. To allow a managed application to query the array of live object instances for a given classifier, this COM component will have to be able to register a given classifier, request the number of live object instances allocated, and finally be able to move the allocated object instances from the unmanaged COM component into the managed application for inspection. These tasks are implemented via the provision of five methods exported by our COM component. The following sections will discuss each of the five exported methods in detail.

IsOCLProfilerAttached

The first exported method determines if the CLR running the managed application has loaded our profiler. The method name contains the abbreviation OCL, for the Object Constraint Language as our implementation of the described profiler is for use with the OCL. More details of our implementation will be provided in Section 5.

Implementation of this method consists of determining if a global reference to the ICorProfilerCallback interface contains a valid pointer. If a valid pointer is located then the profiler has been loaded correctly, otherwise the profiler is not running. The IsOCLProfilerAttached method is not required, but is used as a safety mechanism to determine if the required profiler functionality exists before the managed application calls the remaining four exported methods.

RegisterObject

RegisterObject is used to inform the attached profiler that the managed application would like to keep track of object instances for the given classifier. Classifiers are provided via the single string parameter to the RegisterObject method. The string should contain a fully qualified classifier name. For example, suppose the class Customer existed in the DaveArnold.Data namespace. The call to RegisterObject would take the following form: *RegisterObject ("DaveArnold. Data.Customer").*

Each call to the RegisterObject method adds the given classifier name to the list of interesting classifiers. Profiling only begins following the RegisterObject call. In order to achieve accurate object instance information, the RegisterObject calls should be made immediately after the managed application starts.

GetInstanceCount

The GetInstanceCount method takes a single string parameter, and returns an integer value. The parameter is the fully qualified name of the classifier for which the number of live object instances is requested. GetInstanceCount will invoke the garbage collector to determine which object instances are live at the current time. GetInstanceCount will also wait until the thread processing the queue of finalizers has finished. A finalization method can be viewed as a destructor. The finalization queue is the set of instances that have been marked for deletion, but the runtime has yet to execute the finalization method. Depending on the number and complexity of the finalizers to be executed, GetInstanceCount may be computationally intensive. However, the strategy of forcing garbage collection and waiting for finalizer execution, ensures that the return value is always accurate.

StartInstanceCopy & StopInstanceCopy

StartInstanceCopy is used to inform the profiler that the managed application has requested the list of all object instances for a given classifier. StopInstanceCopy informs the profiler that the managed application has received the requested object instance list. The process of transferring the list of object instances from our COM component to a managed application is a non-trivial operation. The following sections will provide rationale for the operation's complexity, and present the technical details of how the transfer process is accomplished.

4.1.1 Direct Access via the Array Pointer

Intuitively, the easiest implementation would have been to pass the fully qualified classifier name to an exported method, and have the method return a pointer to the corresponding array. The managed application could access the array of pointers and de-reference each one for evaluation. The experienced .NET programmer will quickly realize that a managed application cannot take the address or size of a managed type. The reason for this is if the garbage collector is executed and moves the managed object instance, the object instance array and all pointers to the array will become invalid. As we cannot prevent the garbage collector from executing, nor keep a reference to a managed object, another method is required to get the object instance pointers out of the profiler and into the managed application.

4.1.2 Memory Copy

As each array element in the profiler stores a pointer to the managed heap location where the object instance is being stored, the actual bits can be copied to a new location, which is accessible from the managed application. To allow the managed application access to the memory, we will use the managed application to create a new object instance for the given classifier. We will then use the profiler to copy the memory from the existing live object instance to the newly created object instance. The result is that the managed application will create a new object instance for each element in the array stored in the profiler, and then upon creation, the profiler copies the original element's state information to the new object instance. The new object instances can then be used as needed in the managed application. If the new object instances are modified in the managed application, the original instances are not modified. The following code listing presents a C# method that returns an ArrayList of live object instances for the given classifier type, using the previously described operation.

```
1) public static ArrayList GetInstancesFor(string value,
      Type t) {
2)    VerifyProfiler();
3)    lock(typeof(OCLProfilerControl)) {
4)      int count = GetInstanceCount(value);
5)      ArrayList result = new ArrayList();
6)      StartInstanceCopy();
7)      for(int i=0;i<count;i++) {
8)        Object obj = t.InvokeMember(null,
           BindingFlags.DeclaredOnly | BindingFlags.Public |
            BindingFlags.NonPublic | BindingFlags.Instance |
            BindingFlags.CreateInstance, null, null, null);
9)        result.Add(obj);  }
10)     StopInstanceCopy();
11)     return result; } }
```

Line 2 begins by ensuring that the profiler has been loaded and attached. If the profiler is not available an exception will be thrown. Line 3 creates a mutual-exclusion lock on the remainder of the method. Such a lock is required along with various critical sections in the profiler to prevent new object instances from being allocated during the copy process. In addition, a critical section in the profiler prevents the garbage collector from executing until the copy process has completed. For implementation details regarding threading see [Arn04a]. Line 4 uses the previously defined GetInstanceCount method to determine how many object instances will need to be copied. GetInstanceCount also triggers the garbage collector and finalization process so that the object instance array contains accurate information. Line 6 informs the profiler that the next object instances that we create will be copies of existing ones, and not regular object instances. Lines 7 through 9 create a new object instance for each existing instance. The creation process invokes the ObjectAllocated method in the profiler. Instead of executing the normal behaviour of adding another object instance to the given classifier’s array as previously described, the following behaviour is executed. Based on the number of instances copied since the call to StartInstanceCopy, the profiler is able to determine which element of the array to copy. The profiler then uses the saved ICorProfilerInfo interface pointer to determine the size of the object instance via the GetObjectSize method. Finally, the profiler copies the bits used to store the object instance located at the previously determined array index to the location where the newly created objectId has been located in the managed heap. Once the copy process is completed, the profiler increments an internal counter so that the next array index is used on subsequent calls to ObjectAllocated. Line 9 adds the newly created object instance, which is now a copy of the original one, to the result list. Finally, line 10 informs the profiler that any newly created objects are no longer copies of existing ones.

Summary

Calling the GetInstancesFor method shown above will return an ArrayList that contains a copy of every live object instance matching the fully qualified classifier name provided to the method call. The object instance copies can be used for any activity without affecting the original live object instances.

We have now defined a CLR profiler to track object instance allocation and garbage collection, and have created a connection to the managed application being profiled so that the live object instance set can be accessed. The following section will examine a concrete example of how the profiler and corresponding connection can be used to aid in the execution of software based contracts.

5. EVALUATING THE OCL IN C#

The Object Constraint Language (OCL) [Omg03a] is a constraint specification language with precise semantics [War03a]. More specifically, OCL expressions evaluate without side effects. This means, the state of a system can never change due to the evaluation of an OCL expression. The OCL is not a programming language. It is not possible to write logic or flow control statements in the OCL. A process or thread cannot be created, and only query operations may be called. A query operation is an operation that does not produce side effects. As the OCL is a modeling language by definition, its expressions are not directly executable. The OCL is a strongly typed language. Each OCL expression has a type. To be well formed: every OCL expression must conform to the OCL type rules [Omg03a].

We have integrated OCL version 2.0 assertions into the C# programming language. To support this addition, a specialized compiler has been developed that compiles C# source code along with OCL assertions to provide software based constraint evaluation [Arn04a].

OCL Integration

Keeping with C#'s design goal of simplicity [Tru02a], we used a straightforward notation that allows for maximum flexibility. Our experience has indicated that some developers prefer to inline the OCL in close proximity to the corresponding C# element. Other developers prefer to keep the OCL in a separate repository that is linked to the corresponding C# elements at compile-time. We support both approaches.

OCL blocks are denoted by the "OCL" keyword. The keyword is immediately followed by an opening square bracket. Following the opening bracket, a series of C# style literal strings specify the OCL constraint. A closing square bracket is required to denote the end of the OCL block.

Class invariants can be assigned to any C# structure or class. Preconditions and postconditions can be applied to any C# method, constructor, destructor, delegate, property, or indexer. Our specialized compiler can be configured to enforce, use, or ignore the OCL contracts.

Compilation

Our specialized OCL/C# compiler is based on the Mono C# compiler [Xim04a]. The Mono C# compiler is an open source C# compiler, which allowed us to directly integrate OCL constraints into the core of the compiler. In order to allow for the OCL blocks as defined in the previous section to be processed by the C# compiler, we need to augment the C# grammar accordingly. The grammar modification is straightforward. C# defines the notion of attributes [Hew02a]. Attributes can be placed on any programming element in any order and represent additional metadata for the given programming element. Grammatically, our OCL blocks behave like C# attributes. Our C# grammar modification consists of adding a rule that states that wherever attributes can be specified, zero or more OCL blocks can be specified immediately before the attributes. OCL blocks are defined separately from attributes to allow enforcement of their usage by our compiler.

We run the C# compiler until mid-way through the semantic pass. The C# compiler is then stopped so that each of the OCL constraints can be processed. We now need to convert each OCL constraint into a C# assertion. To accomplish this, we go through each operation attached to each structure or class. When a method, delegate, property, or indexer is encountered the following steps are executed[1].

1. Create a C# parse tree for each of the invariants assigned to the class that contains the operation.

2. Create a C# parse tree for each of the preconditions and postconditions assigned to the operation.

3. If a postcondition uses an element that contains the @pre modifier, a local variable is added to the beginning of the operation and is assigned the value of the requested element. The local variable is then used in the postcondition to represent the requested element's value before the operation is executed. The mini C# parse tree for the postcondition is modified to use the local variable, instead of the actual element.

4. If the operation contains either invariants or postconditions, a local variable (result) is added to the beginning of the operation to represent the return value of the operation. If the return type is void, no variable is added. The C# parse tree for the operation is modified so that all return statements are replaced

[1] The following steps do not take into consideration inheritance in order to preserve understandability in the context of this paper.

by an assignment to the local variable and then a jump to the end of the operation. As we need to check invariants and postconditions at the end of the operation, we change the structure of the method to include an area for making the checks, and enforce that all code paths travel through our new area.

5. The OCL specification dictates that the end result of each of the mini C# parse trees is a Boolean constraint. The Boolean constraints are added to the method's parse tree as follows:

(a) Each invariant constraint is placed into the condition section of an if statement and negated. If the if statement evaluates to true, then the invariant has failed. The body of the if statement will generate an assertion. The invariant statements are placed at both the beginning and the end of the operation.

(b) Step (a) is repeated for the preconditions and the if statements are placed immediately after the beginning invariant if statement.

(c) A Boolean constraint is generated to yield the result of the postconditions.

(d) An if statement is created to determine if the postconditions have failed. The body of the if statement will generate an assertion.

6. After the final if statement in the operation, a new return statement is added to return the result variable. If the result variable does not exist, no return statement is added.

Once each operation's C# parse tree has been updated to include the OCL constraints, the C# compiler is restarted. The rest of the semantic analysis is completed on the main C# parse tree, which includes the additions made by the OCL integration. Upon successful completion of the semantic pass, the C# code generator completes the compilation.

allInstances

The OCL defines an allInstances operation on each classifier. The allInstances operation is defined to return a collection of all the object instances defined using the classifier [Omg03a]. The original version of our C#/OCL compiler did not support allInstances because, as already discussed, C# maintains an automatic garbage collector, so it was difficult to determine when an object instance had actually gone out of scope. In addition, there was no mechanism in C# to get the live object instance list.

With our previously discussed method, we can modify the compiler to support the allInstances operation and allow the user more flexibility when defining software contracts. We will discuss the modifications made to the compiler in order to provide this functionality. As the implementation of the allInstances method will require invoking the profiler and incur additional overhead during application execution, we have created a compiler option to enable allInstances support. If an application that uses the allInstances operation is compiled, and the corresponding option is turned off the compiler will issue an error. If the allInstances compiler operation is enabled, but the application being compiled does not make use of the allInstances operation, the compiler will not add profiling code to the application.

The original compiler already has the allInstances operation defined in the lexical analysis and parsing phases. The semantic analyzer has a skeleton method that emits a compiler error, stating that the allInstances operation is not supported. We have replaced the existing method in the semantic analyzer with one that performs the following tasks. The first step is to ensure that the required compiler option has been enabled, if not the compiler issues an error message. Once it has been determined that the allInstances operation is supported, the compiler uses the OCL expression resolution method [Arn05a] to resolve the front part of the expression. Consider the following allInstances expression.

```
Customer.allInstances()->forAll(c : Customer | c.age >= 18)
```

The compiler resolves the front part of the expression, which should result in a classifier. Once the classifier is resolved, the compiler ensures that the classifier is not a primitive type. According to the OCL specification [Omg03a], the allInstances operation is not defined on primitive types. This makes sense because some primitive types are stored as literals or on the stack, and not in the managed heap. In addition, the set of all integers is not really useful from the software constraint point of view. If the classifier is in fact a primitive type, the compiler will issue an error.

Once the previously defined classifier resolution and primitive type check are complete, the compiler begins to generate C# code to implement the allInstances operation. The first step is to register the classifier with the profiler upon application startup. This is accomplished by inserting a call to the RegisterObject method at the beginning of the application’s entry point. With classifier registration complete, the compiler then generates code to implement actual retrieval of object instances. The OCL expression is translated into the following C# code.

```
bool result = true;
foreach(Customer c in
   (Set)OCLProfilerControl.GetInstancesFor(“Customer”,
   System.Type.GetType(“Customer”))) {
      result = result & (c.age >= 18);
}
```

The expression above, results in a Boolean value specifying if the OCL constraint is valid or not. The GetInstancesFor method is used to return an ArrayList containing the active object instances of type Customer. The first parameter to the method call is a string literal representing the classifier name, the second parameter is a System.Type object representing our Customer. The type object will be used to dynamically create the Customer copies as previously discussed. The GetInstancesFor method returns an ArrayList, the OCL specification indicates that the allInstances operation returns a Set. The Set type does not exist in the .NET Framework Class Library (FCL). The compiler includes an OCL type library [Arn04a], which defines the OCL Set type [Omg03a]. The Set type contains a conversion operator to convert an ArrayList to a Set. Finally, the foreach C# primitive is used to iterate through each Customer in the Set and determine if the age constraint holds. With the OCL allInstances expression converted to a C# Boolean expression, the C# code can be inserted into the application being compiled as discussed in the previous section.

6. CONCLUDING REMARKS

The following section will look at some areas for future work and recapitulate our approach by discussing how the addition to our existing C#/OCL compiler provides the constraint developer with additional resources for writing accurate and detailed constraints.

Future Work

We have only illustrated how this method can be used to implement software based constraints via the OclAny::allInstances method. It would be interesting to explore other uses for the complete set of live object instances. We are currently exploring how our method can be used in the verification and validation of non-functional requirements.

Conclusion

We have seen how a specialized COM component can be written using the Microsoft .NET Profiler API. The profiler API provides our component with notifications when object instances are being allocated on the managed heap, when the object instances are being moved, and finally when they are collected. Using these notifications we are able to maintain a list of live object instances sorted by the creating classifier. As the COM component runs outside of the managed runtime provided by the CLR, a series of exported methods are required to provide an interface for accessing the live object instance list under the CLR. Using the COM component together with the connecting bridge we are able to extend our existing C#/OCL compiler to provide support for the OclAny::allInstances operation. Such support empowers the software constraint designer with additional resources form which more detailed and accurate software constraints can be devised. Ultimately, allowing the constraint designer to create constraints that are not limited by technical aspects, leads to a more complete and accurate software verification and validation process.

7. ACKNOWLEDGEMENTS

Funding for this work has been generously provided by the Natural Sciences and Engineering Research Council of Canada.

8. REFERENCES

[Arn05a] Arnold, D. Constraints in C# using the OCL 2.0. In proceedings of the 23rd IASTED International Conference on Software Engineering, Innsbruck, February, 2005.

[Arn04a] Arnold, D. C#/OCL Compiler at http://www.ewebsimplex.ca/csocl.

[Fra03a] Frankel, D. Model Driven Architecture: Applying MDA to Enterprise Computing. Wiley, New York, 2003.

[Hew02a] Hewlett-Packard, Intel, Microsoft, C# Language Specification. Technical report, ECMA, 2002.

[Hew02b] Hewlett-Packard, Intel, Microsoft, Common Language Infrastructure (CLI). Technical report, ECMA, 2002.

[Hil03a] Hilyard, J. Inspect and Optimize Your Program's Memory Usage with the .NET Profiler API. MSDN Magazine, January, 2003.

[Mic02a] Microsoft, .NET Framework Tool Developer's Guide: Profiling Specification. Technical report, Microsoft, 2002.

[Omg03a] OMG: Response to the UML 2.0 RfP. Technical report, OMG document ad2003-01-16, 2003.

[Stu03a] Stutz, D., Neward, T., and Shilling, G. Shared Source CLI Essentials. O'Reilly & Associates, Sebastopol, 2003.

[Tru02a] Trupin, J. Sharp New Language: C# Offers the Power of C++ and Simplicity of Visual Basic. MSDN Magazine, September, 2002.

[War03a] Warmer, J., Kleppe, A. The Object Constraint Language: Getting Your Models Ready for MDA. Addison-Wesley, Boston, MA, 2003.

[Xim04a] Ximian Mono Project at http://developer.ximian.com/projects/mono

Building a framework for the consistency management of distributed applications

Vilmos Bilicki
University of Szeged
Department of Software Engineering
Hungary, 6720, Szeged
bilickiv@inf.u-szeged.hu

József Dániel Dombi
University of Szeged
Department of Software Engineering
Hungary, 6720, Szeged
dombijd@inf.u-szeged.hu

ABSTRACT

Distributed computing is leaving the laboratory and research lab environment and is now playing a significant role in the infrastructure of different companies and institutions. The requirements of running 7x24 without any noticeable failure can be effectively achieved only with a distributed architecture. The computing power and storage capacity of desktop machines have also become attractive as the basic building blocks of a distributed resource-sharing network.

Along with the useful properties of a distributed environment we get some challenges as well. A crucial question is that of consistent global knowledge among the distributed components. During the building and testing phases of our distributed software package called LanStore it turned out that currently there is no framework for .NET that offers group communication and consistency maintenance. There is the Peer-to-Peer API for unmanaged code that can be used in managed code, but this API was intended to be used in a WAN environment and it does not provide strong guarantees for consistency.

Hence we decided to design and build a framework that supports consistency management. One design criterion we applied was to support a highly changeable environment like that in a student computer laboratory. Our framework does not depend on any underlying communication infrastructure. It can provide the same set of services regardless of whether it is a peer-to-peer network or an IP level multicast network is used as the platform.

Keywords

Keywords: distributed system, consistency, group communication, peer-to-peer

1. INTRODUCTION

The number of the users with broadband Internet access is skyrocketing. According to estimates the number of users with broadband access in the U.S. increased by 36% in 2004. Now almost 70% of all U.S. home users have broadband connections. On a global scale, the number of the users in the world with Internet access grew by 182% during the period 2000-2005. 15.7% of the total world population now has Internet access. This penetration means that more than one billion users (one-sixth of the planet's human population) are connected to the Internet, which is probably the largest community on earth. The value of this community from the business perspective is constantly growing as well. The total Internet spending hit $143.2 billion in 2005[Eni05]. Yet the demands of this market differ from the conventional ones in several respects. The most important difference arises from the fact that, on the Internet, bank holidays and the different parts of the day lose their meaning. Business life should be run in a 7x24 way. But when this point is combined with the fact that the number of users who use a service is rather unpredictable, it is becomes clear that it is no easy task to develop such a system, one that is efficient, reliable and cost effective.

With the current high speed LAN and WAN network infrastructures the distributed paradigm is a reasonable solution for these problems. Such a service is provided by a group of processes that are operating and distributed throughout the network. The user should, however, see this system as a monolithic service and not notice its distributed nature. But using the network as a communication medium among processes introduces new problems. Current data networks - like IP networks - do not give guarantees for the correct delivery of the sent data. A developer has to take into account the variable aspects of the communication channel.

.NET Technologies 2006
Copyright UNION Agency – Science Press,
Plzen, Czech Republic.

One solution that has become more attractive is to use desktop machines as the basic building blocks of a distributed system. These PCs are less reliable than dedicated servers or they may run in environments where continuous operation is not guaranteed (as in a student laboratory for instance). Therefore a reliable distributed system should be able to tolerate the failing of one or more of its serving nodes. Depending on the type of tolerated failure, the system can become quite complex and costly to implement.

To overcome this complexity a common method is to use a framework that hides the failures of the system from the higher layers. Probably the biggest question for a distributed system is that of consistency. To be able to act as one virtual service the distributed system should have a consistent knowledge base. The message-oriented Group Communication Service (GCS) [Vit99] may provide the consistency for a distributed system.

There are well-known frameworks for providing the above-mentioned services, but we found just the Peer-to-Peer API [Win03] was available for the .NET environment. Our experience showed during the building and testing of the LanStore [Bil05] system that a well-tested, general, easily extendable consistency framework removes most of the burdens associated with testing and developing. Hence we decided to build the DCon framework to provide this functionality.

First we will introduce our new contribution, then we will outline the most common services available for group communication. One interesting approach is the Paxos algorithm, which will be evaluated in the next section, followed by a discussion of several well-know frameworks. As one of our goals was to build a framework for a student lab environment, in the next section we present the results of measurements that were conducted in our laboratories. Based on our measurements we designed a framework that is described in the implementation section. In the final section we draw some conclusions and suggest several possible directions for future study.

2. Our contribution

We carried out a set of a measurement to test the reliability of a typical campus computer laboratory. In the literature we found only the [Bol00] study about the reliability of the desktop machines, but this measurement was conducted on desktop machines used mainly by dedicated persons. In contrast, our measurements were conducted in a public student laboratory.

We decided to implement a distributed consistency management framework, we know this is the only distributed consistency management framework for the .NET environment. Our system can use the services of a peer-to-peer network and native IP level multicast too. We implemented the Paxos algorithm [Lam00, Lam01] in a way that is optimal for frequently changing networks (see measurements). Our Paxos implementation is able to handle the membership changes. We ported the Paxos algorithm to a Peer-to-Peer environment where the group members are not on a central list.

3. Distributed systems

A general distributed system may have an arbitrary number of components and each of these components may have a different task and a different state space but to the service user it behaves like a centralized monolithic system. These components may communicate in an arbitrary way. The fault tolerance of these components is usually solved by replication. The replicated components execute the same algorithm and each of them should have the same state. One popular approach is to model this system with state machines [Sch90]. A metric of a distributed system is the safety it provides. Here safety means the number and types of failures it survives without losing consistency. Another important metric is called liveness. This means that with different types and numbers of failures the distributed system can still progress. A widely used solution for the above mentioned issues is the view-oriented group communication service (GCS). Here service reliability is provided at the message level. The following basic services are defined:

1. Membership service
2. Reliable multicast

A view is a state of the system consisting of a set of active nodes. If this set changes, the view changes as well. The most important property provided by a GCS is called “Virtual Synchrony”. If two processes participate in the same two consecutive views the same set of message will be delivered. For further details the interested reader may peruse the article [Vit99].

4. Paxos

The “Virtual Synchrony” property provides the global ordering of the messages and a reliable message delivery in a distributed system. The price we pay for this solution is that it is not scalable. As was shown in the Spinglass article [Ken01], the systems providing “Virtual Synchrony” can scale effectively only up to several tens of nodes.

The classic Paxos [Lam00, Lam01] protocol solves the consensus problem for an asynchronous replicated system. It guarantees consistency in the case of benign failures. Hence this algorithm has better scalability properties than systems with the “Virtual Synchrony” property. The drawback is that the progress of the system is not guaranteed, and the total order of messages is not fully controlled by the clients.

The algorithm solves the following problem. Let P be a set of processes and let V be the set of values. Every process in P can choose one value from the set V, the goal of the Paxos algorithm being to guarantee that only one from these selected values is accepted. The network can delay and multiply the messages arbitrarily; the participating nodes can crash and restart randomly but the Byzantine failures [Lam82] are not tolerated. In other cases system consistency is guaranteed. The progress of the system is guaranteed only in stable periods.

The functionality of Paxos is provided by two basic primitives: the quorum and a global order provider. The task of the quorum is to select at most one value from the available values. There are distributed solutions for preserving the global order of the messages (e.g. GCS), but sometimes a single decider can handle it more effectively. Paxos may be regarded as a special case of the view membership protocols [Lamps01]

5. Recent solutions

For handling the issues of a distributed system in the .NET environment one can use the P2P API [Win03] and the System.Transactions [Win06] namespace. P2P API provides a basic IP overlay infrastructure. As the consistency of the given reliable storage is based on timestamps and serials, and it does not give appropriate feedback about the success or failure of a transaction, it cannot be used in several critical services. The System.Transactions namespace in .Net 2.0 offers only classical transaction services. It is unsuitable for a consensus-based data consistency.

Group Communication Systems-based frameworks have a long history, and they are now in their fourth generation. Here we mention only the most well known frameworks.

Isis [Bir94] was the first and best-known primary component membership service. Among other services it defined and provided the "Virtual Synchrony" property for the first time.

Transis [Dal96] was the first GCS that utilised the native IP level multicast services. It was the first partitionable membership service. The system contains multicast clusters that are interconnected. It has a multicast flow control mechanism that controls the traffic at the network level. It also supports group communication. The messages can be unordered, causally ordered, and totally ordered and safely delivered.

Totem [Mos96] utilises the native IP multicast capabilities of the underlying network too. It provides a system-wide total ordering of the messages even in the case of network partition and remerge ("Extended Virtual Synchrony"). This goal is achieved with a logical ring where only the token holder may speak. In larger networks there are hierarchical ring topologies.

The goal of the Ensemble [Ken00] project was to improve the quality of the software used in the Isis project. Instead of the monolithic approach the system was implemented using modules and well-defined interfaces. The micro-protocol stack further improves the flexibility of the system. The code was implemented in the ML language, which is an O'Calm variant language. With this approach they were able to define and perform transformations on the code in a mathematically proven way.

Spinglass [Ken01] uses a revolutionary new approach. The currently used GCS's cannot be scaled up to a really large number of nodes. The Spinglass project addresses this problem and it uses "gossip-based" protocols to provide a highly scalable, secure and reliable Group Communication System. The gossip protocols emulate the spread of an infection in a crowded population. It employs a NNTP like protocol [Kan86] (Bimodal multicast) as the basic infrastructure provider. This protocol gives a steady data delivery rate with predictable, low variability in throughput. It provides only probabilistic guarantees of virtual synchrony.

6. Feasibility study

Our university has a computer science laboratory with 204 PCs. Students can either use the Windows or Linux operating systems from 8 am. to 8 pm., and they can switch between the operating systems whenever they want. We measured machine availability by pinging these machines every minute for 3 weeks between February 6 and February 25 in 2006. Based on the TTL value of the response we were able to detect not only the failures but the type of the operating system too.

We measured that a week the mean number of the online Windows workstations was always above the critical 50%.

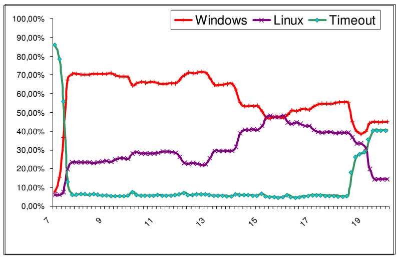


Figure 1: Operating system percentage / hours (2006.02.20)

The first figure shows the same statistics but now for a particular day. We notice that during the day except for a short period the number of online windows machines was above the critical level. The difference was about 10%. In the next figure the number of restarts is shown for another day. We notice that there are situations where more than 10% of the machines are restarted. In such cases it may happen that during a transaction more than 50% of the windows machines are online but the ones that are running may vary.

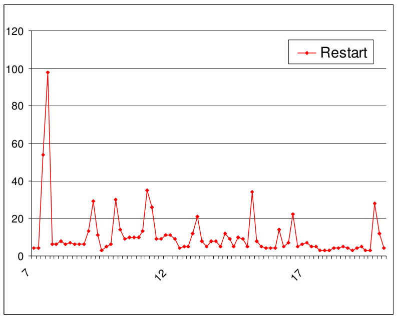


Figure 2: Number of restarts every 10 minutes (2006.02.14)

From these measurements we may conclude that for a reliable and liveness system we have to take into consideration these special time periods.

7. The DCon framework

The goal of the framework is to provide a distributed replicated data storage service with strong safety guarantees and weaker liveness properties. It can tolerate any arbitrary number of non-Byzantine failures. The liveness property is guaranteed only when more than the half of the nodes are active, but these nodes can change from time to time.

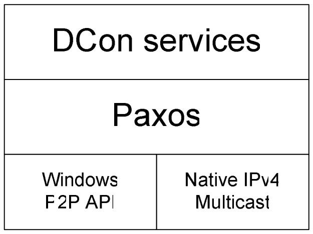


Figure 3.

We could have followed the approach of the above-mentioned frameworks and implemented a message-level GCS. But as our framework will provide only consistency services and not group communication services, we constructed it so that it would handle the issue of consistency more effectively. We selected the famous Paxos algorithm, which is ideally suited for these purposes. The reliability of this algorithm is mathematically proven. It can tolerate an arbitrary number of non-Byzantine failures without losing consistency. To be able to use it in a WAN environment and to be effective in a LAN environment we implemented it on the top of the Windows Peer-to-Peer API and the native IP level multicast services.

The DCon framework has three layers. These layers are shown in Figure 3. The first layer hides the distributed nature of the system from the user. It provides basic data manipulation and configuration services for the user. A data item can be added to the system, and existing data items can retrieved by a slow or fast query (see the next section). There are several methods available for reconfiguring the system.

The second layer implements the Paxos algorithm in a network independent way. At the bottom are the network dependent modules. Currently there are two modules: the native IP level multicast module and the module based on the services provided by the Windows Peer-to-Peer API.

In the following section we will describe our implementation of the Paxos algorithm in native multicast and P2P environments.

8. Our Paxos implementation

Functionality is provided by three abstractions: Leader, Consensus algorithm, Learner.

From a higher point of view the system works as follows. The clients send instructions to a leader. This leader carries out a three-phase transaction on the participating nodes and sends the results to the client.

Now we will describe the algorithm and a detailed description of our implementation (please consult Figure 4 for details).

Firstly, during the implementation phase of the classic Paxos algorithm we had to solve the following problems:

Message ordering: The purpose of the leader abstraction is to serialise the incoming requests. As we have seen this task can be done in a distributed manner (with logical timestamps and so on), but these solutions are more costly and are less reliable than the single leader solution. One could argue that the single leader incorporates a single point of failure into system. This is true, but as the leader does not have persistent data it can be easily replaced by a live substitute.

Leader election: As a communication medium between the Leader and the participating nodes, the *Instructions* multicast channel is used. During idle periods, the Leader periodically multicasts a beacon packet that contains the number label of the latest instruction. Based on our experience in other fields we chose to set this period to 10 seconds. During active periods these packets contain Paxos instructions (*Propose, Accept, Decide*). Failure detection is achieved by timeouts. If there is no traffic on this channel for three times the beacon period (30 seconds), the clients will submit a *LeaderSelect* frame that contains their stability properties (the greatest message serial known by this node, the number of restarts, the duration of the longest stable period). Each node compares the received values with its values and if it discovers that its values were better (in the case of equal values the greater IP number is chosen) it will wait for a random period between 0 and 15 seconds and start sending beacon packets. If a node thinks that it has the best values but receives beacon packets it will accept the new leader. With these settings a Leader change will last at most 45 seconds.

Learning the actual leader: There is a *Leader* channel where the leader submits the beacon every 30 seconds. This channel is intended for clients for them to determine the actual leader. The clients send the data items to be stored to the leader using a TCP connection.

Monotony maintenance: The leader node retransmits the messages from the clients to the *Instructions* multicast channel and these values assigns a global number *G* and local number *N* to the messages. Local numbers are interesting only when there are two or more leaders. These numbers should be unique among the leaders so it is constructed as follows: IP address+ $N*2^{32}$. For every submitted message *G* is increased and *N* is reset to 0. *G* and *N* are included in the beacon packets as well.

Every node in the distributed system is subscribed to the *Instructions* multicast channel. For every different global number there will be a separate "Synod" protocol that guarantees consistency among the nodes. It works as follows:

Phase 1. The leader selects a global *G* and a local number *N* for the instruction and sends it as a proposal for the nodes subscribed to the *Instructions* channel. This is the so-called *Prepare* request. If a receiving node receives a *Prepare* request it checks whether it is able to accept it. If the last accepted request has a global number which equals the received global number and the local number is less than that of the current request then it responds with a reject answer, otherwise it will send a prepare accept response. Both of the responses contain the last accepted request and the also the number of this request.

Phase 2. If the leader receives a response for its propose request from the majority of the nodes, then it selects the latest accepted request, or if there was no request previously then it uses its own request and sends an accept request to the *Instructions* channel. In the case of insufficient responses or a reject answer it will increase the local number and submit the prepare request again. If there are insufficient responses after the fifth unsuccessful round it will stop the process and send an unsuccessful message to the clients. If it gets one or more reject answers it will increase the *N* value and send the message again. After five unsuccessful turns it will increase the value of *G* to the maximal value reported by the clients plus one received in the reject messages. If it is unsuccessful then it will report this to the client. This situation can happen only when there are several leaders and all are functioning for a longer period of time. But this may happen only in very special circumstances. It is quite rare.

The node receiving an accept request checks the local number of the request, and if it is greater than the last accepted one or there was no such *G* then it accepts the request and sends an accepted message to the leader. Otherwise a reject response is sent with the *N* value and maximal known *G* value.

Phase 3. After receiving sufficient accepted messages the Leader sends a *Decided* message to the *Instructions* multicast channel. The node that receives the *Decided* message will insert the Decided values into its Decided values storage. The timeout for each phase is 20 seconds. If the number of received accept messages was less than the previously defined majority value it will try sending the accept request again. If it fails five times it will send this result to the client and stop the process. In the case of a reject message it will follow the process described in Phase 2 and restart Phase 1. If the chosen value was not the value originally sent by client, then the Leader will repeat the whole process until the decided value and the accepted value coincide. This situation may occur if the *G* known by the leader is less than the greatest *G* in the whole system.

A detailed description of this algorithm can be found in [Lam00, Lam01]. We implemented the Paxos algorithm using several optimisations to achieve better response times:

For the system to progress we need the majority of nodes to be live. It may happen that in a fluctuating system, the majority of nodes are always present but are constantly changing. For example the prepare request is received by node A, then node A restarts and node B finishes its restarting process. So node B will only receive an accept request. The classic Paxos algorithm recommends rejecting this message. But with this solution it can happen that we have to replay the whole propose/accept procedure. Instead of this we suggest the following. If a node receives an accept request without previously receiving a propose request it shall answer this request. If it disagrees with the value suggested by the accept request it shall handle the accept request as a propose request; if it agrees with the received value then it shall handle the accept request as a propose and accept request. With this modification we did not change the durability of the algorithm, but in some cases we reduced the required number of message exchange from six to two. This algorithm is described in [Lam01].

```
Phase1:
Server:
            Var ReceivedRequest([G[N,V]], Iteration=0
            SendPropose(Nx$2^{32}$,G)
```

```
Node:
        Var ReceivedProposes [G[S,V]]
        ReceivePropose(S,G){
        IF(G not known)
                SendAcceptPropose()
        ELSE IF ($S_{max}$ <S)
                SendAcceptPropose($S_{loc}$,$S_{value}$)
        ELSE
                SendRejectPropose(G,$S_{max}$,$G_{max}$)
        }
Phase2:
Server:
        IF (ReceivedAcceptPropose > Memb/2)
                IF(MAX(S) != 0)
                        SendAcceptReq(G,$S_{loc}$,$S_{value}$)
                ELSE
                        SendAcceptReq(G,$S_{loc}$,V)
        ELSE
                IF(N<5)
                        N=N+1
                        GOTO Phase1.
                ELSE
                        REPORT ERROR
Node:
        IF(G not known)
                SendAcceptReq()
        ELSE IF (Smax <S)
                SendAcceptReq ()
        ELSE
                SendRejectReq(G,$S_{max}$,$G_{max}$)
Phase3:
Server:
        IF NUM(ReceivedAcceptReq > Memb/2)
                SendDecide(G,V)
        ELSE
                IF(N<5)
                        N = N+1
                        GOTO Phase 1
        IF(Sv != V)
                G = G+1
                GOTO Phase 1
        ELSE
                SendSuccess()
```

Figure 4. Algorithm

Change of membership: The participating nodes maintain two lists of instructions. In the "Client list" are stored the data items submitted by the clients, while the "System list" contains the instructions for system maintenance. The handling change of membership is solved by these special instructions, which are treated the same way as instructions from clients.

Message optimization: A *Leader* may incorporate an arbitrary number of Paxos messages with different *G* values into one submitted packet. The Decide packets may be piggybacked to Accept packets. The prepare packets are only needed during the start of a longer stable period. With these optimisations we then need only one message per transaction during stable periods. The details of these optimisations were mentioned in part in two papers [Lam00, Lam01].

Slow/Fast query: A client may learn the chosen values in a fast or slow way. The fast way is to query the adjacent node about its list of decided values. The slow way is to perform a distributed query of the missing values. This query is submitted to the *Instructions* multicast channel. The distributed query contains the number label of the last known decision. The nodes receiving the query will respond to and return the accepted values. The client will summarise the answers and in the case of unknown new decisions it will send a decide message to the *Instructions* multicast channel to help the progress of the whole system.

9. Measurement

We tested our implementation in different circumstances to prove that the single leader role does not affect its stability.

To be able to simulate different network conditions we developed a simulation framework where every machine was simulated with separate thread. With the help of this solution we were able to fine tune the machine restart probabilities.

In the following we will present our results about the stability of the leader election process. During the experiment we simulated 200 PCs with the restarting probability of 10% to 50% . On the Figure 5 we can notice, that the system converged in a very fast manner in the case of low restarting probability. If we raise the restarting probability the system also converged, but in this case the convergence is slower, and it contains more peaks.

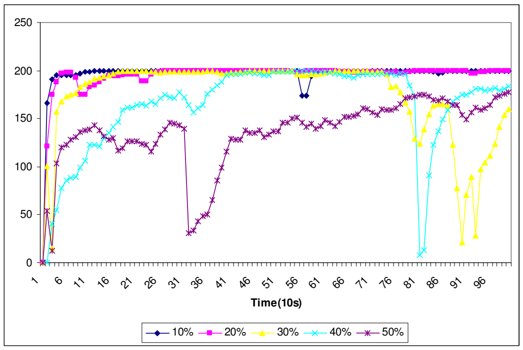


Figure 5. Number of threads, which are know the good leader at the same time

10. Paxos in a peer-to-peer environment

The services of the Windows Peer-to-Peer API were described in the recent solutions section. We can if we wish use it as a basic infrastructure to build an IP overlay multicast service. The communication service will be less efficient than in the native case, but in some situations we cannot use native multicast services anyway. The reliable storage service does not guarantee safety properties comparable to those of Paxos.

In our system we solved the following problems:

Group membership: To be able to implement a Paxos-like algorithm with guaranteed safety properties we have to know something about the success of the spread of the information. For this we need some membership details. As this framework assumes that there will be a high number of nodes there is no central information about the membership. To overcome this, we chose to measure the total weight of the network and this value will be refined from time to time, but we will save only the maximal value while the system is running. Based on the maximal and the current value, the algorithm will be able to decide whether a partitioning has occurred and if this partition is capable of acting as a reliable storage medium. In the case of partitioning only the partition with weight more than the half of the whole weight will be suitable to act as a reliable storage. This solution works when, after the partitioning, there is a will on the user's part to merge the graph. If the partitioned sections start their lives separately, one can initiate a separate instance of the consistency algorithm on each. After doing so, it will be impossible to unify the network, however as the algorithm is intended to preserve global consistency there is no easy way of merging systems with a different history. With this membership view the nodes in the Peer-to-Peer network act as Paxos nodes.

Global order: This can be handled in a distributed or centralised way. The decentralised solution may be a more suitable solution for a peer-to-peer network, but as the Windows Peer-to-Peer API uses a central point of the network for graph maintenance we opted for this solution. A step toward the fully decentralised solution could be the use of per client root nodes. In this case an additional iteration is needed to evaluate the global order of the values. This could be done with the help of the weight of the groups which accepted a value with the same serial number.

After this high-level overview we will describe how our solution works:

The Peer-to-Peer network or segment has a central point- the node with the smallest ID (the same node being used for graph maintenance). This node sends a beacon signal every T seconds to each of its neighbours. The main task of this beacon is to measure the weight of the network.

Loop free message transfer: The graph constructed by the P2P system is a redundant one, hence we need an algorithm to avoid the situation of message loops. The P2PDatabase article [Awe02] advocates using spanning trees, but in a dynamic network it would be a costly solution. So we decided to use the well-known "Path Vector" algorithm [BGP06][Win03] (the same idea being used for name queries in MS P2P API). Every beacon packet has a path vector attribute that contains the sequence of nodes it traversed during its trip. If a node receives a beacon packet it first checks whether it is present in this attribute. If it finds its ID then the packet will be discarded. It then inserts its ID at the end of the path vector attribute and submits the packet to each of its neighbours except the neighbours which are present in the path vector. With this solution we have a multicast communication infrastructure.

Aggregated feedback: To measure the current weight of the network, each node will send a feedback to each beacon packet with the aggregate number of feedback packets received. Every non-leaf node (i.e. one which transmitted a beacon packet) has to wait for an answer for each submitted beacon packet. As we use the services of the Windows Peer-to-Peer API, theoretically the neighbours are always online (if not, the graph maintenance algorithm will correct this), but to avoid a potentially long delay of 5 minutes, every node has to maintain a timer for each submitted beacon frame. The timeout value will be inversely proportional to the number of nodes in the path vector attribute. In the case of a timeout it will send back a packet with a weight value of one. If there are redundant paths it may send the same feedback back several times. To avoid this, the synchronising packets contain a timestamp. A node will answer with an aggregate weight only for the first packet, and for the remaining packets with the same timestamp it will respond with a feedback containing a zero weight. Finally the root node will aggregate the feedbacks and this number will be the weight of the current network. The maximal value during this time will be the membership weight of the network. To ensure that this value is common knowledge, it will be attached to each beacon frame and stored at every node.

The root node acts as the leader in our Paxos algorithm. The algorithm is the same as in the case of native multicast, the only difference being that the nodes aggregate the answers they receive and send this answer as feedback values. The *Propose*, *Accept*, and *Decide* packets can act as beacon packets too. The root node will send beacon packets only after a defined idle time. To minimise the network traffic a submitted packet may contain several Paxos packets for several instructions and types. The root node will receive an aggregated feedback from participating nodes. The weight of the response should have a value greater than maxweight/2.

Slow/Fast query: The fast query option is the same as in the native multicast case. The slow query contains the last known decision number. The algorithm is the same as in the case of beacon packets. The feedback packets will contain the decisions known by the traversing nodes. Every transmitting node will check the feedback values for unknown decisions and then store them. If a node discovers that one or more of its accepted values are not present among the decided values, it will attach these values to the voted values. If it finds its accepted values among voted values, then it will increase the counter for these values. Thus the client will be able to learn the decided values and also the values accepted by the majority of nodes. The detection of root node failure is handled by the underlying framework. Each node also checks whether it is the root node of the new graph. If it finds that it is, then it will initiate a query to learn the last synchronising number of the decided and proposed values. The result of this query will be the weight value of the current network. If it finds that it is larger than the half of the previous one, then it will start acting as the leader.

11. Conclusions and future work

In this article we described a solution which provides consistency services in a distributed environment. We implemented the well-known Paxos algorithm and solved several associated problems. As our framework handles only the consistency problem and it provides no group communication services ours should not really be compared to recent systems like Isis and Transis.

Our goal was to provide a simple and reliable API for consistency handling. Currently we also provide the same set of services on the P2P framework and on native IP level multicast.

Our software package is now in the development stage. Timing can be critical in a distributed system. The current values are based on our experience in the field of IP routing where the neighbour maintenance solves the same failure detection issue [OSPF96]. The tuning of the timeout values should be done in a real environment and software package should be tested under a variety of conditions.

In the future we would like to add a gossip-based module that can be deployed in the Windows P2P API. With this module the framework will not just be effective in LAN, but will be scalable in WAN as well.

12. Acknowledgement

We would like to thank David P. Curley for checking this article from a linguistic point of view.

REFERENCES

[Awe02] B. Awerbuch and C. Tutu. Maintaining Database Consistency in Peer to Peer Networks. Technical Report, CNDS-2002-2. 2002

[Bil05] V. Bilicki. LanStore: a highly distributed reliable file storage system, .NET Technologies'2005 conference proceedings, ISBN 80-86943-01-1, pp. 47-57, 2005

[Bir94] Birman K., van Renesse R. (editors) - Reliable Distributed Computing with the Isis Toolkit, IEEE Computer Society Press

[Bol00] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In Proceedings of SIGMETRICS, Santa Clara, CA, June 2000.

[BGP06] Y Rekhter, T. Li, S. Hares. A Border Gateway Protocol 4 (BGP-4). http://www.ietf.org/rfc/rfc4271.txt

[Dal96] D. Malki and Y. Amir and D. Dolev and S. Kramer. he Transis Approach to High Availability Cluster Communication. Communications of the ACM, 39(4):63--70, April 1996.

[Eni05] Enid Burns. Broadband: Online Retail Sales Grew in 2005. http://www.clickz.com/stats/sectors/retailing/article.php/3575456 , January 2006

[Kan86] B. Kantor, P. Lampsley. RFC 977 - Network News Transfer Protocol. 1986 http://www.faqs.org/rfcs/rfc977.html

[Ken00] Ken Birman, Robert Constable, Mark Hayden, Christopher Kreitz, Ohad Rodeh, Robbert van Renesse, Werner Vogels. Proc. of the DARPA Information Survivability Conference & Exposition (DISCEX '00), January 25-27 2000 in Hilton Head, South Carolina.

[Ken01] R. Shostack, and M. Pease. Kenneth P. Birman, Robbert van Renesse and Werner Vogels. Spinglass: Secure and Scalable Communication Tools for Mission-Critical Computing. International Survivability Conference and Exposition. DARPA DISCEX-2001, Anaheim, California, June 2001.

[Lam00] L. Lamport. Part time parliament. ACM Trans. on Computer Systems, 16(2), May 1998.

[Lam01]L. Lamport. Paxos made simple. ACM SIGACT News Distributed Computing Column, 32(4), December 2001.

[Lam82]L. Lamport, R. Shostack, and M. Pease. The Byzantine Generals Problem. ACM Transactions on Programming Languages and Systems, 4(3):382-401, 1982.

[Lamps01] B. W. Lampson. The ABCDs of Paxos. Principles of Distributed Computing, 2001.

[Mos96] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. LingleyPapadopoulos. Totem: A fault-tolerant multicast group communication system. Communications of the ACM, 39(4):54--63, April 1996.

[OSPF96] Y. Moy. OSPF Version 2. http://www.ietf.org/rfc/rfc2328.txt

[Sch90] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: {A} Tutorial. ACM Computing Surveys, 22(4):299-314, 1990.

[Vit99] R. Vitenberg and I. Keidar and G. Chockler and D. Dolev. Group Communication Specifications: A Comprehensive Study. Tech. report CS99-31, Comp. Sci. Inst., The Hebrew University of Jerusalem and MIT Technical Report MIT-LCS-TR-790, Sep. 1999.

[Win03] Microsoft. Introduction to Windows Peer-to-Peer Networking. November 2005. http://www.microsoft.com/technet/prodtechnol/winxppro/deploy/p2pintro.mspx

[Win06] Microsoft. System.Transactions Namespace. 2006. http://msdn2.microsoft.com/en-us/library/system.transactions.aspx

Implementing Unified Access to Scientific Data from .NET Platform

Sergey B. Berezin
Assistant Professor,
Moscow State University,
Computational Mathematics and
Cybernetics, Leninskie gory,
1 MSU, 119992, Moscow, Russia.
s_berezin@cs.msu.su

Dmitriy V. Voitsekhovskiy
Postgraduate,
Moscow State University,
Computational Mathematics and
Cybernetics, Leninskie gory,
1 MSU, 119992, Moscow, Russia.
idmitry@inbox.ru

Vilen M. Paskonov
Professor,
Moscow State University,
Computational Mathematics and
Cybernetics, Leninskie gory,
1 MSU, 119992, Moscow, Russia.
paskonov@cs.msu.su

ABSTRACT

Scientific data differ from common relational data in many aspects: scientific data may have a very complex structure, they are usually stored in files of various formats and individual data items can be very large. In this paper we present an extensible and efficient client-server system for accessing scientific data and its metadata. The architecture and major capabilities of our system will be described in the paper. The core of our approach is an extensible XML-based structure that annotates scientific data with rich metadata and maps every file or part of a file to a named strongly typed entity.

We do not introduce any new file formats and file transfer techniques, thus our approach doesn't require major changes to existing computational software. SOAP protocol and Web Services are used for accessing data sets and performing data requests. Filtering and caching enables an efficient access to large portions of data over network. Example of implemented filters are cropping and thinning of 2D and 3D arrays.

Our system is fully extensible and allows adding new data types, new file formats and new filtering algorithms without changing its core algorithms. Now it is used for accessing results of computational fluid dynamics simulations, but we hope that it can be adapted to many branches of science. The client is implemented on the .NET platform; the server-side is currently running on the IBM Regatta SMP mainframe on AIX

Keywords

Scientific data access, data management, visualization, web services, SOAP.

1. INTRODUCTION

Scientists are overwhelmed today by amounts of data generated by experiments and simulations. According to the Scientific Data Management Center at the Lawrence Berkeley National Laboratory [Sdm05w] up to 80 percents of a scientist's time is spent on data manipulation and only 20 percents – on actual analysis. That's why there is an emerging need of more convenient tools for scientific data access and analysis. Tendencies of scientific data management in near future are listed in [Jim05a] along with vision of the next generation data analysis tool called "smart notebook". In this paper we make a small step to such a tool by introducing our approach which consists of two parts: a scientific data access system and a data visualization tool.

A lot of systems for scientific data management and analysis were developed for many branches of science, from astronomy [Jim01a] to computational fluid dynamics problems on irregular meshes [No01a]. Our system origins from the field of computational fluid dynamics but we believe that it appears to be useful in other branches of science.

Most important features of scientific data management systems can be found in the survey [Rea00a]. In our approach we focus on following aspects:

Logical data management – a data management system abstracts from the physical data layout. The resulting view of the data is a uniform collection of data items.

Physical data management – a request for logical data items results in a transparent physical files access, filtering and caching.

.NET Technologies 2006
FULL papers conference proceedings
ISBN 80-86943-06-2
Copyright UNION Agency – Science Press,
Plzen, Czech Republic.

Metadata management – metadata describes data themselves [Jef02a]. Metadata is an important part of scientific data set, because it helps a scientist to understand data better and it helps various tools to perform a data analysis and visualization more efficiently.

2. RELATED WORKS

The evolution of Web technologies along with cheaper and more powerful hardware and increased networks bandwidth has brought to life new approaches to scientific data management. The huge number of online repositories and data centers allows scientists to publish, to search, to display and to download data.

The NCSA's Scientific Data Service (SDS) [Sds97w] provides Web access to a wide range of scientific data, facilitating data sharing between science teams and the general public. The SDS is a CGI program that provides scientific data in the several well known file formats. SDS is extensible and modular, but it is a fairly time consuming task to make SDS understand a new file format.

The metadata in SDS contains the fixed number of attributes to search by: spatial, temporal, dataset name, archive center, parameter name, platform name, sensor name, etc. Users can interactively examine the contents of a file with their Web browser, view a thumbnail image of the data, and retrieve the file, or a desired subset of the file, in its original file format or in ASCII. SDS has no object-oriented features and lacks support for client-side data management and caching.

The OpenGIS scientific data server [Ogs97w] is created by joint efforts of NCSA and the USDAC Consortium. It provides geospatial data according to object model described by the OpenGIS Abstract Specification. This model hides format details for three different types of geospatial data. Access to the scientific data objects is performed through the OpenGIS API. The objects returned to client can be visualized or saved as files. But object became a isolated entity after it has been obtained and it holds no reference to source data set.

The Distributed Oceanographic Data System (OPeNDAP) [Dap04w] is intended to give researchers a transparent access to oceanographic data across the Internet. Communication model in OPeNDAP works with URL addresses of web servers that deliver data to the researcher. In fact, researcher's data analysis software acts as a sophisticated web browser. Each data set is accessed via URL. Calls of API functions are forwarded to referenced web servers. Depending on the request type, the server returns a textual description of the data set contents or the actual values of data variables in a binary form. Textual descriptions provide a client library with metadata information concerning the operations that can be applied to data and the way binary data is to be decoded. The OPeNDAP incorporates a data translation facility, so that data may be stored in formats defined by the data provider, yet may be accessed by the user in a manner identical to the access of local files. Thus, the system provides transparent access to scientific data, but still there is no support for client-side data management.

Originality of our approach is based on following features: (1) integrity of data sets during their entire lifecycle, (2) efficient client-side data management and (3) common object-oriented API based on SOAP and XML. Another feature of proposed system is its high extensibility resulted from .NET Framework dynamic nature.

3. ABOUT DATASET

3.1. Common Features of Scientific Data

Long time passed since the single standard and SQL have been developed for the relational data model. However, scientific data strongly differ from common relational data in several aspects. This makes existing data management paradigms unsuitable for scientific data [Jim05a]. There is still no unified model for accessing scientific data. In this paper we introduce a new approach to the scientific data access that seems to be pretty general.

Our logical data model was designed to reflect following common features of all scientific data:

- Scientific data may have a very complex structure and are usually stored in files of various specific formats; individual data items can be very large.
- Scientific data often depend on parameters (for example, on time) or can be viewed as a collection of parameter *slices*.
- Practically all results of scientific researches contain both data and metadata.

Metadata can be of two types. The first type of metadata is designated for human reading and contains information about simulation parameters, about authors and the origin of data and so on. This type of metadata allows associative search, categorization and better understanding of scientific data by external researchers.

The second type of metadata describes the type and the format of scientific data. It is most useful for different automated tools for data retrieval, filtering and analysis. For example, the information about the type helps the visualization system to suggest the most suitable visualization method and its parameters.

3.2. DataSet Object Model

DataSet is a key notion of our approach to the scientific data management. It can be thought as a self-describing entity containing references to actual data annotated with rich metadata. DataSet Object Model is shown on Fig.1.

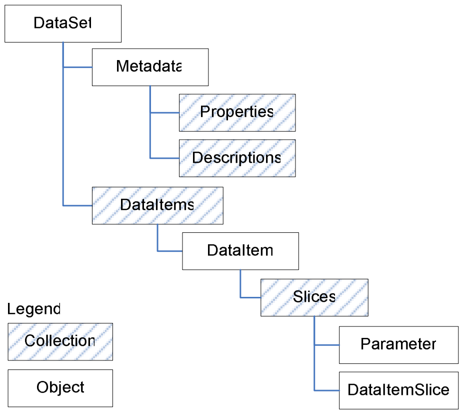


Figure. 1. DataSet Object Model.

The Metadata section holds descriptive information about DataSet and its data in a human readable form. The Properties collection contains information about simulations parameters, units of measurements, affiliations and authors of data. The Descriptions collection contains descriptions and annotations of any object in a DataSet. These collections can be used for searching and arranging DataSets.

The Metadata section also contains an address of a data source server (to which a data request should be sent) and the origin of a DataSet. The former allows copying and distributing the DataSet, keeping its functionality, and the latter allows checking for possible updates to the DataSet.

Every logical part of data set is represented by DataItem, which maps a portion of real data to a named strongly typed object. A DataItem can depend on one or more named parameters. Thus, a DataItem is a collection of so-called slices, which correspond to data for specific parameters values. The value of a DataItem for specified parameters is represented by an individual DataItemSlice object.

Each parameter has a name and a strongly defined type such as double, string etc. The example of parameters in computation fluid dynamics is time or the Reynolds number, in geophysics – coordinates of a data capture.

A DataItem can be either simple or composite. Simple DataItems hold references to a data piece that can be retrieved from a single location. We do not introduce new file formats, but instead we rely on existing well known formats such as netCDF or HDF [Fmt06w]. The usage of existing file formats has following advantages:

- We can easily assembly existing data in DataSets;
- We can use existing libraries to write or read DataItems of DataSets;
- We can extract parts of DataSet for processing with existing tools and utilities.

Composite DataItems are built by on the basis of one or more components (see Fig.2). Each component is the pair of a DataItem (possibly also composite) and an optional class name of the component. Class names help to distinguish components.

The following example introduces a constructor for computation fluid dynamics problems. Let's assume that the DataSet contains two DataItems: `uvw-values`, as a three-dimensional array of vectors, and `channel`, as a spatial grid. Combination of 3D vector array and data grid is a vector field. The constructor named `DataField` is used to create a composite DataItem `velocity`, representing the vector field. In such a way, the composite DataItem should be declared in the DataSet as an output of the `DataField` constructor depending on two components: the `uvw-values` with the class "values" and the `channel` with the class "grid" (see listing 1).

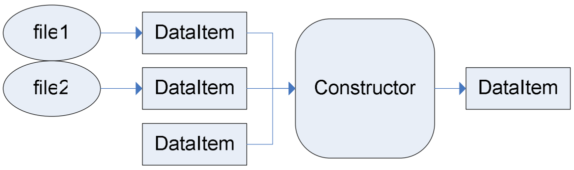


Figure 2. Composite DataItem construction.

Another example of important constructors is the constructor named `CompositeVectorArray` that allows creating new arrays by combining several arrays with smaller dimensions of items (see example listing 1), and vice versa.

A DataSet aggregates different data sources transparently for applications and makes it possible to view scientific data as a single collection of typed objects. This allows both logical and physical data independence.

The common standard for XML metadata descriptors and the DataSet XML schema definition were developed and now they are used in data repository for simulations in the field of CFD. We believe that this structure will be suitable for many fields of science, from computational fluid dynamics to biological systems modeling.

3.3. DataSet Example

The following DataSet XML document describes results of the numerical modeling of an unsteady flow of viscous incompressible fluid in a flat channel. The results of the modeling consist of four files with a scalar array (three for the fluid velocity components and one for the pressure) for each moment of time.

```
<dataset id="…" dataSource="http://..."
   origin="http://..." type="CFD" …>
  <metadata>
    <property name="Re"
        description="Reynolds number"
        type="double" value="140.0" />
    …
    <description>Incompressible viscous flow
in a 3D channel</description>
    <description id="velocity">Velocity
vector field</description>
  </metadata>
  <structure>
    <dataItem id="uvw-values"
              type="Vector3dArray3d" >
     <composite
        constructor="CompositeVectorArray">
      <component id="u-values" />
      <component id="v-values" />
      <component id="w-values" />
     </composite>
    </dataItem>
    <dataItem id="velocity"
              type="VectorField3d" >
     <composite
        constructor="DataField">
      <component id="uvw-values"
           class="values"/>
      <component id="channel" class="grid"/>
     </composite>
    </dataItem>
    <dataItemTemplate id="u-values"
      type="ScalarArray2d"
      sourceType="netCDF" />
  </structure>
  <data>
    <dataItem id="channel"
     type="NonUniformGrid3d"
     sourceName="grid.dat"
     sourceType="plain text" />
    <parameter name="time" type="double">
      <slice value="0.00000">
        <dataItem id="u-values"
             sourceName="u_0000.cdf" />
        …
      </slice>
      …
    </parameter>
  </data>
</dataset>
```

Listing 1. Example of DataSet XML document.

The `structure` section specifies composite DataItems and templates for simple DataItems. The `dataItemTemplate` element is used to simplify DataItems declarations, especially parameterized. If the template is defined for certain `id` then attributes of the DataItem with the same `id` in `data` section will be considered as defined by default and may be either omitted or redefined with new values.

In the `data` section there are simple DataItems defined and arranged in slices by parameters values. In our example DataItems are defined for every moment of time and correspond to each component of the velocity vector and pressure. The DataItem `channel` represents the mentioned above spatial grid, that does not depend on time.

4. IMPLEMENTATION DETAILS

4.1. Architecture Overview

The server-side is currently running on the SMP mainframe IBM Regatta on AIX and implements two Web services. The first Web service performs administrative functions and provides access to DataSets. Search by metadata values is possible. The second Web service serves requests for DataItems and performs filtering. Complementary data request caching is used to maximize the speed of the service. The client-side of the system is implemented on the .NET platform as class libraries. Global view of our system is shown on Fig. 3.

The central class of the libraries is a DataSet. It is developed according to the DataSet object model (see Fig.1) and can be constructed on the basis of an XML document that represents DataSet entity. The DataSet class contains metadata and a collection of named objects of the DataItem class.

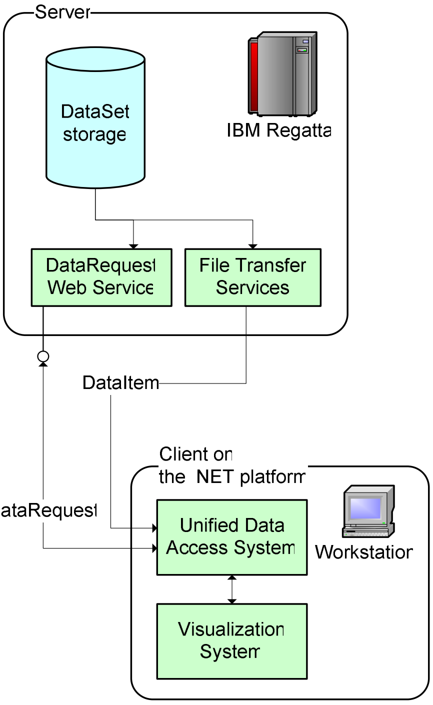


Figure 3. Architecture overview.

The structure of the DataItem class is represented by a tree, the nodes of which correspond to parameters and leaves – to DataItemSlice objects. The DataItem class offers convenient methods for data indexing by a set of parameters, returning an object of the DataItemSlice class for specified parameters values.

The DataItemSlice methods provide a direct access to data and return typed data object.

When data object is requested the system automatically forms and sends a data request to a data source address that is declared in metadata of the DataSet. Thus, client applications work with data transparently and without caring where and how they are stored.

The following code accesses array of velocity vectors at the moment 0.0 from the DataSet described in section 2.3. It can be seen in the DataSet that the required array is located on a server as three different files, but the physical representation of the data does not matter for the client at all.

```
// Creating object of class DataSet using
// XML-representation of DataSet
DataSet dataset = new DataSet(xmlDoc);
// Fetching DataItem by its name
DataItem velocity =
    dataset.DataItems["uvw-values"];
// Creating parameter corresponding to time
CompositeParameter param =
    new CompositeParameter(
    new ParameterValue("time", 0.0d) );
// Fetching DataItemSlice for the parameter.
// It is an instance of DataItem for
// specified parameter value.
DataItemSlice dataVelocity =
    velocity[param];
// Getting required data
Vector3dArray2d data =
  dataVelocity.GetData() as Vector3dArray2d;
```

Listing 2. Getting required data in C#.

4.2. Data Filtering

In most cases an application may request filtering of the data, i.e. their additional processing. For instance, a visualization program does not need such detailed grid data as they are usually computed in numerical experiments. Therefore thinning filter will be useful in this situation, because the resulting data after filtering will have exactly as many points as necessary for its correct visualization.

Another example of filtering is cropping. Let us assume that a scientist wants to study part of the data in detail. There is no need for a full local copy of existing data in that case, therefore cropping filter will return only required data.

Data filtering can be performed either by the client-side of the system or by the server-side. It occurs absolutely transparently for applications that work with the system: the decision where filtering will take place is taken by the system itself.

Thus, besides specific data handling for specific problem field, filters increase the efficiency of the system and reduce network traffic.

The following example expands the previous one given in listing 2 and illustrates how an application may request data with additional filtering. If there is no need for such a detailed velocity vectors array as it stored in files, a thinning filter may be applied to the data. The “Thinner” filter has parameters PercentageX, PercentageY and PercentageZ – those are fractions of points for each axis, which shall remain after filtering, and we make them equal to 5%. Code in C# is shown below:

```
// Creating object of class DataSet using
// XML-representation of DataSet
DataSet dataset = new DataSet(xmlDoc);
// Fetching DataItem by its name
DataItem velocity =
    dataset.DataItems["uvw-values"];
// Creating parameter corresponding to time
CompositeParameter param =
    new CompositeParameter(
    new ParameterValue("time", 0.0d) );
// Fetching DataItemSlice for the parameter.
// It is an instance of DataItem for
// the specified parameter value
DataItemSlice dataVelocity =
    velocity[param];

// Creating filter "Thinner" for required
// data type and setting up its parameters
Filter filter = FilterFactory.GetFilter(
       "Thinner", // filter class name
       dataVelocity.TypeDescriptor);
FilterServices.SetFilterParameters(filter,
   new FilterParameter[] {
   new FilterParameter("PercentageX", 0.05),
   new FilterParameter("PercentageY", 0.05),
   new FilterParameter("PercentageZ", 0.05)
        } );

// Getting required data
Vector3dArray2d data =
    dataVelocity.GetData(filter)
    as Vector3dArray2d;
```

Listing 3. Getting filtered data in C#.

Here an application gets the required filter, requesting it from the `FilterFactory` object by the filter’s class name and the type of data, to which it shall be applied. Use of class factories is one of the keys which enable the system’s high extensibility.

4.3. Performing DataRequest

DataRequest contains DataItem reference and filters, which shall be applied to this DataItem. DataRequest’s content provides all information required to load the data. Any DataItem reference belongs to one of the three types. The first type, named dataSource, is designed for server’s handling, which can locally (for the server) load requested data according to the reference. The second type, named dataRef, is used for remote loading of data that already are available on server as one file or directory. Besides the data type, dataRef contains the transfer protocol type and URL. The third type of a DataItem assumes that data are stored inline in

DataRequest. It is designated for transferring small pieces of data and improves the overall efficiency of request processing.

When an application requests data, the client libraries form DataRequest for specified DataItem from DataSet. DataRequest is passed by SOAP protocol to DRS Web service (see Fig.3), which loads requested data and tries to apply specified filters.

Only part of the filters might be applied, because some of filters may be either absent on server or inapplicable for particular data types. After filtering is completed, the server makes filtered data shared for the client, removes applied filters from DataRequest, and replaces all dataSource elements with dataRefs referring to the data or with inline data (see Fig.4).

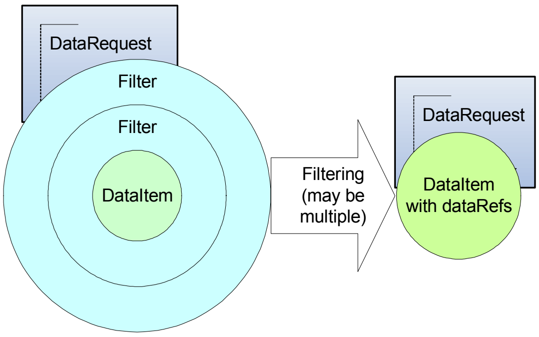


Figure 4. DataRequest lifetime.

The final DataRequest is sent back to the client. In this stage it contains either inline data or dataRefs, i.e. what and how a client must load, and a list of unapplied filters (which may be empty). As the data are downloading to a local computer, remote dataRefs become local references. After that, the client-side of the system parses the data and applies the filters which have been failed at the server. Data parsing is performed by special data source objects. The system can use various data source objects for each pair of a data source type and a type of data, which shall be loaded. All information that is necessary for a data loader is contained in dataRef.

We neither introduce a new file transfer technique nor restrict the choice of the existing one. The data transfer type is specified in dataRef by the server depending on its capabilities or any other term (for example, a security policy). Currently this is a transfer by FTP that is used, i.e. the server returns an address of FTP endpoint and a path to the needed file. One more file transfer possibility is the usage of WS-Attachments extension. This option is simple and interoperable, but it requires an extra bandwidth and may not be applicable due to a security policy on some systems.

In all suitable cases both the server and the client make caching of the request's result to decrease request handling time. Data request processing diagram is shown below.

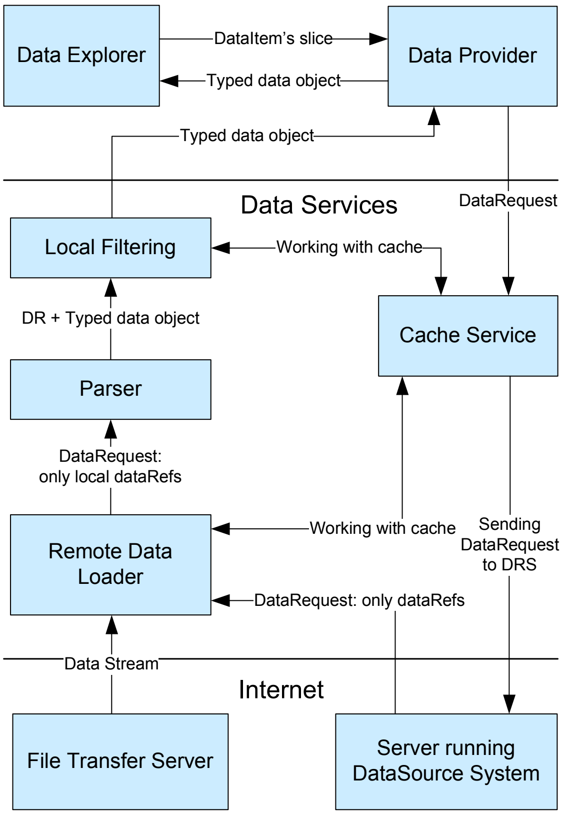


Figure 5. DataRequest performing schema.

Contents of DataRequests, which are generated by code on listing 3, are shown on listing 4:

```
<soap:Envelope … >
 <soap:Body>
  <dataRequest
       dataSource="…" dataSet="guid" … >
   <filter name="Thinner">
    <parameters> … </parameters>
    <dataItem type="Vector3dArray3d">
     <composite
       constructor="CompositeVectorArray">
      <component> <!-- u-values -->
       <dataItem type="ScalarArray3d">
        <dataSource sourceName="u0000.cdf"
sourceType="netCDF" sourceParameters="u" />
       </dataItem>
      </component>
      <component> … </component> <!-- v -->
      <component> … </component> <!-- w -->
     </composite>
    </dataItem>
   </filter>
  </dataRequest>
 </soap:Body>
</soap:Envelope>
```

Listing 4. DataRequest that is sent to the server.

```
<soap:Envelope …>
 <soap:Body>
  <dataRequest
       dataSource="…" dataSet="guid" …>
```

```
    <dataItem type="Vector3dArray2d">
      <dataRef sourceType="binary"
         sourceParameters="">
       <ftp url="ftp://..." />
      </dataRef>
    </dataItem>
  </dataRequest>
 </soap:Body>
</soap:Envelope>
```

Listing 5. DataRequest received from server

4.4. Extensibility of the System

One of our goals is not to design system for handling CFD-related data, but to create extensible and adaptable framework for managing scientific data sets. Our system can be extended by new data types, new data sources and new filters.

New data type is just a CLR class with no other requirements. Additional interfaces such as IScalarArray2d or INonUniformGrid3d whose names speak for them are implemented when needed. For each data type special type descriptor can be defined in configuration file.

Data sources are used for loading data objects from files or for composing new data objects from existing ones (example is constructing vector array from few scalar arrays). Thus, new data source has to be developed for each new file format or for new composite data type. Data sources are also listed in configuration file.

Data filters transform data objects according to filter's parameters. New data filters should implement two main functionalities: filter should be able to embed itself in XML data request for server processing and be able to perform actual client side filtering if it is not supported on server. Filters are also defined in configuration file.

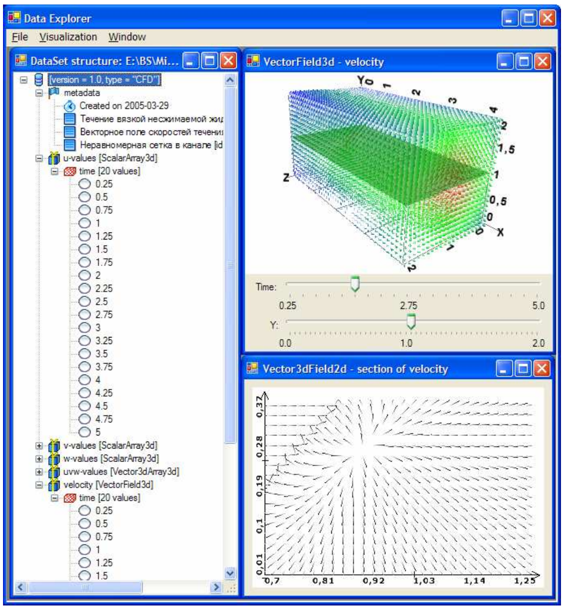
For each new type of objects CLR class name and strong assembly name is specified in configuration file. On system start-up configuration file is examined. Assemblies are loaded on demand and objects are tied together in runtime using reflection and dynamic type information.

The system's architecture also allows every module having special code optimizations. For example, a filter can be optimized for work with a certain data type (from any module) and vice versa.

5. VISUALIZATION

Atop the data access system described above we build a visualization system for graphical exploration and analysis of data. A sample screenshot is shown below.

Our visualization system is built around the concept of workspace – a combination of DataSets and DataViews. It is important to mention that Workspace contains only references to DataSets, so Workspace is a very compact data structure that can be easily transferred from the researcher's workstation to his or her notebook providing a familiar work environment at any location. The structure of DataSet is shown in the left window on the screenshot. There you can see a list of DataItems and their parameters.



[version = 1.0, type = "CFD"]
metadata
Created on 2005-03-29
Течение вязкой несжимаемой жид
Векторное поле скоростей течени
Неравномерная сетка в канале [id
u-values [ScalarArray3d]
time [20 values]
0.25
0.5
0.75
1
1.25
1.5
1.75
2
2.25
2.5
2.75
3
3.25
3.5
3.75
4
4.25
4.5
4.75
5
v-values [ScalarArray3d]
w-values [ScalarArray3d]
uvw-values [Vector3dArray3d]
velocity [VectorField3d]
time [20 values]
0.25
0.5
0.75
1
1.25
1.5

Figure 6. Visualization system screenshot

DataView is a visual object formed by a pair of a data object and a visualization algorithm. On the screenshot you can see one primary DataView (in the right-top window) and one dependent DataView (in the right-bottom window).

The primary DataViews take one of DataItems as its data object. The visualization algorithm can be chosen by the user from a list of options that is formed according to a DataItem type. Options could be sorted additionally according to the problem description found in metadata (i.e. physical oriented visualization algorithms will be on top for CFD problems).

If a DataItem depends on some parameters, the user is given a choice either to create a DataView for individual parameter slices or to display the entire DataItem with extra dimensions added by parameters. For example, a scalar 2D data field in coordinates (u,v) dependent on time can be displayed as an animation of a 2D surface in time or as a 3D scalar field in coordinated (u,v,t).

The secondary DataViews is created by applying one of visualization tools to an existing DataView, primary or secondary. The visualization tool is an object that can be applied to a specified type of DataView, has its own visual representation and results in a new data object. The green plane in the right-top window is a section tool that extracts 2D subset from a 3D vector. The section plane can be moved up or down using control below. On the screenshot values of the 2D vector field subset are shown as a 2D marker field.

So, the Workspace can be thought as hierarchy of DataViews with DataSets as roots. Interacting with controls changes data in the DataView and this change is propagated automatically to all dependent DataViews. Although the data flow paradigm is not new in the field of scientific visualization [Vis96a] our visualization system allows graphical constructions of new visualization tools instantly from a visual representation of data. We believe that this will enable scientists to get new insights into data on the fly.

6. SUMMARY

The scientific data access system presented in this paper has following advantages:

- It gives an object oriented view to scientific data, which means that the client can retrieve metadata and data as strongly typed objects with caching and filtering.
- It allows creating a single family of data analysis tools, because almost any set of scientific data can be represented as a DataSet.
- It provides an indexing and associative search of data by their attributes and parameters hiding their physical location.
- It is highly extensible and provides interfaces for adding new data types, new types of data storage and new filters. This makes our system applicable to almost every branch of science.
- It is designed to interact with existing data storage formats and there is no need to abandon the existing computational or simulation software.

7. FUTURE WORK

We plan to extent our approach in three ways:

- Implement data management abilities – currently our system is a data access system with no ability to modify DataItems or DataSets.
- Extend a set of visualization tools by extending our visualization software and providing interfaces for our data access system from the existing powerful visualization software such as AVS
- Implement in-memory cache on client computer. Weak references are not suitable for this task because when amount of data exceed hundreds of megabytes weak reference became invalid shortly despite that there are still a lot of free memory.
- Implement server-side software on .NET Platform with reusing significant part of client-side code for data filtering and parsing.
- Design and implement second version API using language integrated queries and features found in LINQ [Lnq06w].

8. ACKNOWLEGEMENTS

This project was supported by RFBR grant 05-07-90378, 04-01-00332 and by the Student Laboratory of Microsoft Technologies at the Moscow State University.

9. REFERENCES

[Vis96a] William J. Schroeder, Kenneth M. Martin, William E. Lorensen. The Design and Implementation of an Object-Oriented Toolkit for 3D Graphics and Visualization. IEEE Visualization '96

[Sdm05w] Scientific Data Management Center at Lawrence Berkeley National Laboratory at http://sdm.lbl.gov/sdmcenter

[Jim05a] Jim Gray; David T. Liu; Maria A. Nieto-Santisteban; Alexander S. Szalay; Gerd Heber; David DeWitt. Scientific Data Management in the Coming Decade. Microsoft Research Technical Report MSR-TR-2005-10

[Jim01a] Jim Gray; Alexander Szalay; Ani Thakar; Peter Z. Zunszt; Tanu Malik; Jordan Raddick.Christopher Stoughton; Jan van den Berg. The SDSS SkyServer - Public Access to the Sloan Digital Sky Server Data. Microsoft Research Technical Report MSR-TR-2001-104

[No01a] J. No, R. Thakur, D. Kaushik, L. Freitag, and A. Choudhary. "A Scientific Data Management System for Irregular Applications", in *Proc. of the Eighth International Workshop on Solving Irregular Problems in Parallel (Irregular 2001)*, April 2001

[Rea00a] Reagan Moore. Data Management Systems for Scientific Applications. IFIP Conference Proceedings; Vol. 188. pp. 273 – 284, 2000.

[Jef02a] K. G. Jeffery, A. Asserson, A. S. Lopatenko, Comparative Study of Metadata for Scientific Information: The place of CERIF in CRISs and Scientific Repositories. Gaining Insight from Research Information, 6th International Conference on Current Reseach Information Systems, August 29-31, 2002 in Kassel, German

[Sds97w] NCSA Scientific Data Server http://hdf.ncsa.uiuc.edu/horizon/DataServer/sds_design.html

[Ogs97w] Open Geospatial Consortium http://www.opengeospatial.org/

[Dap04w] OPeNDAP: Open Source Project for a Network Data Access Protocol. http://www.opendap.org

[Fmt06w] Scientific Data Format Information FAQ http://fits.cv.nrao.edu/traffic/scidataformats/faq.html

[Lnq06w] The LINQ Project. http://msdn.microsoft.com/netframework/future/linq/

MC# 2.0: a language for concurrent distributed programming based on .NET

Yury Serdyuk
Program Systems Institute of
Russian Academy of Sciences
Russia (152020), Pereslavl-
Zalessky

Yury@serdyuk.botik.ru

ABSTRACT

In this paper, we introduce a new version of MC# — a language for .NET-based concurrent distributed programming. This language is an adaptation of the basic idea of the Polyphonic C# language (Benton N., Cardelli L., Fournet C., Microsoft Research Laboratory, Cambridge, UK) for the case of distributed computations.

We present the background and goals of developing the language and introduce its novel constructs : movable methods, channels and handlers. We describe the specific features of MC# and formulate differences between its current and previous versions. Examples of programming in MC# are given: a program for finding prime numbers by Eratosthenes sieve, and a program named *all2all* which demonstrates interaction between distributed processes. In conclusion, we give a brief description of the current implementation along with the list of applications that have been developed, and identify directions for future work.

Keywords

Concurrent distributed programming, MC#, movable methods, channels, handlers, Runtime-system, .NET.

1. INTRODUCTION

The wide use of computer systems with massive parallelism, such as multicore processors, clusters and Grid-architectures, posed again the problem for developing high-level, powerful and convenient programming languages that would allow one to create complex and at the same time reliable software systems that efficiently use the possibilities of concurrent distributed computations and are easily scalable to a given number of processors, nodes or computers.

Currently available program interfaces and libraries for organizing parallel computations, such as OpenMP [OpenMP] (for systems with shared memory) and MPI (Message Passing Interface) [MPI] (for systems with message passing), have been implemented for C and Fortran languages, and hence are very low-level and inadequate for modern object-oriented programming languages like C++, C# and Java. Additionally, such interfaces rely on the use of libraries rather than on appropriate programming language constructs.

.NET Technologies 2006
FULL papers conference proceedings
ISBN 80-86943-06-2
Copyright UNION Agency – Science Press,
Plzen, Czech Republic.

In general, a modern high-level programming language consists of two parts:

1) basic constructs of the language itself, and

2) a collection of specialized libraries accessible through appropriate APIs (Application Programming Interfaces).

New demands on increased programmers productivity (achieved through a higher abstraction level of language constructs, among other things), as well as on reliability and security of programs they develop, account for a tendency to transfer key concepts of most important APIs into the corresponding native constructs of programming languages.

For example, the embedding of asynchronous methods and chords into Polyphonic C# [BCF04], which is an extension of the C# language, allows one to use it without the System.Threading library, which is normally required to implement multithreaded applications on top of .NET. On the other hand, the introduction of new data type constructors (for streams, anonymous structures, discriminated unions and others) along with appropriate query definition tools into Cω language [BMS05] renders obsolete the ADO.NET data subsystem (specifically, the traditional System.Data and System.XML libraries intended to handle relational and semistructured data).

We suggest that the next step in this direction be to introduce high-level constructs for creating concurrent distributed programs into the object-oriented language, and thus to free the programmer from the need to use the System.Remoting library (and, in many cases, also the System.Threading library), which is required to develop conventional distributed applications using C#.

From the practical point of view, the goal pursued by the developers of MC# was to design a language for industrial concurrent distributed programming which is going to involve more and more human resources, with the oncoming age of multicore computations. This language aims to replace C and Fortran languages in this area. It allows to create complex software systems that have satisfactory effectiveness when executed on parallel architectures. The choice C# as a basic language gives the possibility of using a modern object-oriented programming language equipped with rich libraries (like libraries for Web-application development, specifically, for dealing with Web-services, designing graphical applications, implementing systems with a high degree of security etc.), and, at the same time, to eliminate such low-level and unsafe features as C pointers which dramatically decrease programmer's productivity and the reliability of software systems. In this regard, our approach coincides with that taken in the development of the X10 language [SJ05], which is oriented towards "non-uniform cluster computing".

In MC# language, in contrast to using MPI interface, there is no need to distribute computational processes over cluster nodes explicitly (though such possibility also is provided by the language) – it is enough only to identify which functions (methods) can be executed concurrently. Moreover, in MC# language the new computational processes can be created and distributed over accessible nodes during program execution dynamically (X10 language also provides for that possibility for "activities"), that is impossible for MPI-programs. Similarly, there is no necessity to code by hand an object (data) serialization preparing moving them to remote node or machine — the Runtime-system performs an object serialization/deserialization automatically.

In fact, MC# language is an adaptation of the basic idea of the Polyphonic C# language (more precisely, of the basic idea of the join calculus [FG02]) for the case of concurrent **distributed** computations. As a matter of fact, the authors of the Polyphonic C# language presumed that asynchronous methods would be used either on a single computer or on a set of machines where they have been fixed and interact through the remote method call tools provided by the .NET Remoting library. In the case of MC#, the execution of an autonomous asynchronous method can be scheduled on a different machine selected either of two ways: by explicit indication by the programmer (which is not a typical case) or automatically (in this case, usually a cluster node or machine in the Grid network with the least workload is selected). Interaction of asynchronous methods that are executed on different machines is implemented through message passing using channels and channel message handlers. In MC#, channels and handlers are defined using chords in the Polyphonic C# style.

Channel message handlers are a new feature of MC# 2.0 as compared to the previous version of the language [GS03]. The second significant distinction consists in a different semantical treatment of channels and handlers (see the third key feature of MC# language in Section 2.1 and a forthcoming paper [S06]).

The paper is organized as follows. Section 2 describes the novel constructs of the MC# language — movable methods, channels and channel message handlers. In Section 3, we demonstrate how MC# constructs can be applied to develop two concurrent distributed programs — finding prime numbers by Eratosthenes sieve and *all2all* program demonstrating interaction of distributed processes. In Section 4, we give details about the current MC# implementation, which consists of a compiler and a Runtime-system. We provide conclusions and directions for the future work in Section 6.

2. NOVEL CONSTRUCTS OF MC#: MOVABLE METHODS, CHANNELS AND HANDLERS

In any sequential object-oriented language, conventional methods are synchronous: the caller always waits until the method called is completed, and only then continues its work.

The key feature of Polyphonic C# (which, in fact, became a proper part of the Cω language — and from now on we will refer only to the latter) is the introduction of so called "asynchronous" methods in addition to conventional synchronous methods. Indeed, such asynchronous methods are intended for playing two major roles in programs:

1) the role of autonomous methods implementing the concurrent parts of the basic algorithm and executed in separate threads, and

2) that of the methods intended for delivering data (possibly, with preliminary processing of it) to conventional, synchronous methods.

In the MC# language, these two kinds of methods form two special syntactic categories of:

1) movable methods and

2) channels

respectively.

In Cω, auxiliary asynchronous methods used for data delivery are usually declared together with synchronous methods. In MC#, the latter are represented as another special syntactic category that includes **channel message handlers** (**channel handlers** or even **handlers** for short).

2.1 Movable methods

Writing a parallel program in MC# language reduces to labeling with the special keyword **movable** the methods that may be transferred to other machines for execution:

modifiers **movable** method_name (arguments) {

< method body>

}

In MC#, movable methods are the only way to create and run the concurrent distributed processes. A consequence of the mentioned above properties of the movable methods is that

1) method call completes almost immediately (time is spent only on transferring the needed data to the remote machine),

2) movable methods never return a result (for interaction of movable methods among them and with other parts of the program, see Section 2.2 "Channels and handlers").

Correspondingly, by the rules of correct definition, movable methods:

- may not have a **static** modifier, and

- never use a **return** statement.

The movable method call has two syntactical forms:

1) object_name.method_name (arguments)

- in this case, the Runtime-system selects the execution location for a given movable method automatically, and

2) machine_name@object_name.method_name

(arguments)

- in this case, the execution location is indicated by the programmer explicitly.

Worth to note is that the objects created during an MC# program execution are **static** by their nature: once created, they don't move and remain bound to the place (machine) where they were created. In particular, it is on this machine that they are registered by the Runtime-system, which is necessary for delivering channel messages to those objects.

The first key feature of MC# language (or, more precisely, of its semantics) is that, in general, during a movable method call, all necessary data, namely

1) the object itself to which the given movable method belongs, and

2) arguments (both objects and scalar values) for the latter

are only **copied** (but not moved) to the remote machine (in **nonfunctional** mode – see below). As a consequence, changes made afterwards to the copy will not affect the original object.

In particular, if a copied object has channels or handlers, they also are copied to the remote machine — they become "proxy" tools for the original objects (see Section 2.2 for details).

There are two modes of parallelizing MC# programs: "functional" and "nonfunctional" (or objective), and the choice will, in the end, affect the efficiency of program execution. These modes are defined by the modifiers **functional** and **nonfunctional** in the movable method declaration (the default value is **functional**).

In the functional mode, an object for which a movable method is called, is not transferred to a remote machine (i.e., all needed data are passed to the movable method through its arguments). Conversely, by specifying the **nonfunctional** modifier, we force the object to be moved to the remote machine.

The use of MC# on cluster architectures, which typically consist of the frontend machine and the subordinate nodes, is specific in that the names for both the frontend and the node are to be specified if a movable method is being called under explicit indication of execution location:

machine_name : node_name@o.m (args)

Movable methods in MC# are similar to "activities" in X10. In the latter, asynchronous activities are created by a statement ***async** (P) S*, where *P* is a place expression and *S* is a statement. In contrast to MC# language with a "method level" concurrency, it is possible for multiple activities to be created in-line in a single method in X10.

2.1 Channels and handlers

Channels and channel message handlers are the tools to support the interaction of distributed objects.

Syntactically, channels and handlers are declared using chords in the Cω style. In the following example, the channel *sendInt* for transferring single integers is defined along with the corresponding handler *getInt*:

```
CHandler getInt int () & Channel sendInt ( int x )
 { return ( x );}
```

In such declarations, handlers have the following general format:

```
modifiers CHandler handler_name
                              return_type (args)
```

We can also declare a channel or a group of channels without a handler. In this case, we can use values being received by the channel through the global variables.

By the rules of correct definition, channels cannot have a **static** modifier, and so they are always bound to some object much in the same way as ordinary methods:

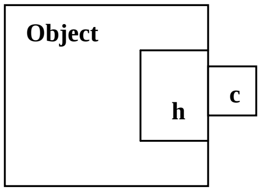


Figure 1. An object with channel *c* and handler *h*

Thus, we may send an integer *x* by the channel *sendInt* as

```
a.sendInt ( x ),
```

where *a* is an object for which the channel *sendInt* has been defined.

A handler is used to receive values from its jointly defined channel (or group of channels). For example, to receive a value from the channel *sendInt* we need to write

```
int  m = a.getInt ( )
```

If, by the time a handler is called, the channel is empty (i.e. if there have been no calls to this channel at all or all of the values sent through this channel before were selected during previous calls to the handler), then the call blocks. After receiving a value from the corresponding channel, the body of the chord (which may consist of arbitrary computations) runs and returns the result value to the handler.

Conversely, if a value is sent on a channel when there are no pending calls to the handler, the value is simply saved in the internal channel queue, where all the values coming with multiple sendings to this channel are accumulated.

It is worth to note that separate methods (handler or channels) from the chord are typically called from different threads of which the entire concurrent distributed program consists.

Similarly to Cω, it is possible to define several channels in a single chord. This is a major tool for synchronizing the concurrent processes in MC:

```
CHandler equals bool () & Channel c1 ( int  x )
                            & Channel c2 ( int y ) {
         if   ( x == y ) return  ( true );
         else             return ( false );
}
```

Thus, a general rule for chord triggering is the following: the body of a chord is executed only after **all** methods declared in the chord header have been called.

The above example illustrates the case of a single handler for multiple channels:

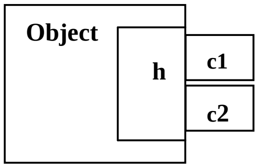


Figure 2. An object with a single handler for multiple channels

It is also possible to declare a channel shared by several handlers:

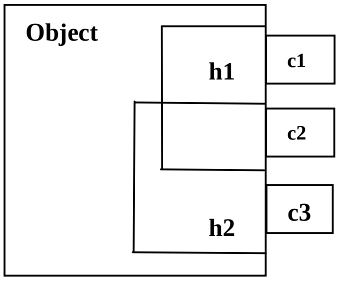


Figure 3. An object with a "shared" channel

So, once we have the values in both channels *c1* and *c2*, handler *h1* can be triggered. Similar is the case for channels *c2* and *c3* and handler *h2*. In general, all this together leads to non-determinism in program behaviour.

The second key feature of MC# language is that the channels and handlers can be passed as arguments to the methods (in particular, to the movable methods) **separately** from the object to which they belong (in this sense, they are similar to the pointers to methods or, in C# terms, to the **delegates**).

The third key feature of MC# language is that if channels or handlers were copied to a remote site (by which we mean a cluster node or a computer in the Grid-network) autonomously or as part of some object, then they become proxy objects, or intermediaries for the original channels and handlers. And the point here is that this replacement is hidden from the applied programmer — he can use the passed channels and handlers (in fact, their proxy objects) on the remote site as the original ones: as usual, all actions over the proxy objects are transferred to the original channels and handlers by the Runtime-system. In this sense, channels and handlers are different from ordinary objects: manipulations over the latter on a remote site are not transferred to the original objects (see the first key feature of MC# language).

Fig. 4 and Fig. 5 schematically demonstrate the passing and use of channels and handlers on a remote site. The subscripts in the channel and handler names denote the original site where they were created.

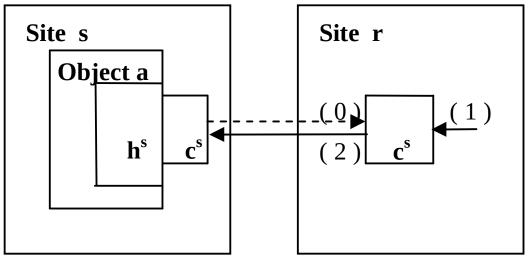


Figure 4. Message sending by remote channel:

(0) copying of the channel to remote site,

(1) message sending by (remote) channel,

(2) message redirection to the original site.

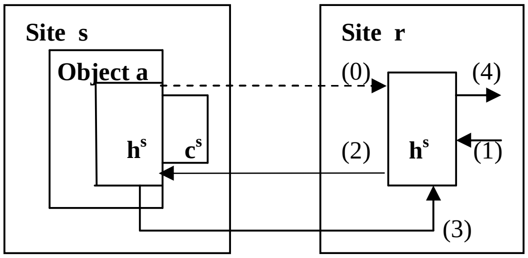


Figure 5. Message reading from remote handler:

(0) copying of the handler to remote site,

(1) message reading from (remote) handler,

(2) reading redirection to the original site,

(3) message return from the original site,

(4) result message return.

It turns out that these tools are enough to organize interaction of arbitrary complexity between the concurrent distributed processes.

In MC#, distributed processes can interchange arbitrary objects using channels and handlers. In X10, data interchange between places is realized through explicit spawning of asynchronous activities. So, if some thread wants to get a remote value *v*, it must create two activities:

```
final place origin = here;
finish async ( v ) = {
    final int x = v;
    async ( origin ) y = x;
}
```

In contrast to this, MC# Runtime-system hides from the programmer the spawning of auxiliary threads during message passing (see the example programs in the next Section).

3. PROGRAMMING IN MC#

In this Section, we will demonstrate the specific constructs of MC# language — movable methods, channels and handlers — and their semantic properties, on the example of two concurrent distributed programs.

First, we will build a parallel distributed program for finding prime numbers by the sieve method (also known as “Eratosthenes sieve”).

Given a natural number N, we need to enumerate all primes in the interval from 2 to N.

The sieving method is the following recursive procedure applied to the original list [2, … , N]:

1) select the head of the given list and output it to the resulting list of primes;

2) construct a new list by deleting from the given list all integers that are multiples of the head of this list;
3) apply the given procedure to the newly constructed list.

The main computational subroutine, which we called *Sieve* and the recursive calls to which will be distributed over a computer network, has two arguments: the handler *getList* to read the given list of numbers it will search for primes and the channel *sendPrime* to write the resulting list of primes. The end marker in both lists is -1.

An elementary step of unfolding the distributed computations (which consists of producing the next unit of the “conveyor” which sieves the integer stream) is sketched on Fig. 6.

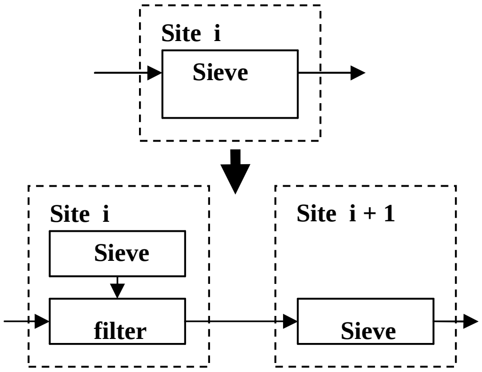


Figure 6. Unfolding step in the distributed sieve method

The full program text in MC# is given below. The original integer list [2, … , N] is sent on the channel *Nats* and the resulting list of primes is received from the channel *sendPrime* by the handler *getPrime*:

```
class Eratosthenes   {
 public static void Main (string[] args) {
  int N = System.ConvertToInt32 (args[0] );
  Eratosthenes E = new  Eratosthenes();
  new CSieve().Sieve ( E.getNat, E.sendPrime );
  for ( int n=2; n <= N; n++ )
    E.Nats ( n );
  E.Nats ( -1 );
  while  (  ( int p = E.getPrime() ) != -1 )
   Console.WriteLine ( p );
 }
 CHandler getNat  int()  & Channel Nats ( int n )
 {   return  ( n );  }
 CHandler getPrime  int()  & Channel sendPrime
   ( int  p )  {  return  ( n );  }
}
class  CSieve  {
 movable Sieve ( CHandler int() getList,
                 Channel (int) sendPrime ) {
  int  p = getList();
  sendPrime ( p );
  if  ( p != -1 )   {
   new CSieve().Sieve ( hin, sendPrime );
   filter ( p, getList, cout );
  }
 }
 CHandler hin int()  &  Channel cout ( int x )
{ return  ( x ); }
 void filter (int p, CHandler int() getList,
                  Channel ( int ) cfiltered )   {
  while ( ( int n = getList() ) != -1 )
    if ( n % p != 0 ) cfiltered ( n );
  cfiltered ( -1 );
 }
}
```

The second program, called *all2all*, demonstrates how we can provide for interaction inside a set of distributed processes in accordance with the “all to all” principle.

Below, each distributed process is an object of the *DistribProcess* class. It starts on a remote site selected by the Runtime-system, by calling the *Start* movable method of the mentioned class.

In turn, each distributed process creates *BDChannel* (Bidirectional channel) object containing the channel *Send* and the handler *Receive*, on its own site. By interchanging *BDChannel* objects, distributed processes can send or receive messages to and from one another regardless of their physical location. *BDChannel* object interchange is realized through the main process which is executed on the machine where the application was started.

Below we present the full program text in MC# where the number N of distributed processes is given as the input parameter.

```
class  All2all   {
 public static void Main (string[] args)   {
  int  i;
  int  N = System.Convert.ToInt32 ( args [ 0 ] );
```

```
          // N is number of distributed processes
  All2all a2a = new All2all();
  DistribProcess dproc = new DistribProcess();
  //   Launch distributed processes
  for ( i = 0; I < N; i++ )
    dproc.Start ( i, a2a.sendBDC, a2a,sendStop );
  //   Receive BDChannel objects from processes
  BDChannel[] bdchans = new BDChannel [ N ];
  for ( i = 0; I < N; i++ )
    a2a.getBDC ( bdchans );
  //   Send BDChannel array to each process
  for ( i = 0; i < N; i++ )
     bdchans [ i ].Send ( bdchans );
  //   Receive stop signals from processes
  for ( i = 0; i < N; i++ )
    a2a.getStop();
 }
 CHandler getBDC void(BDChannel[] bdchans)  &
      Channel sendBDC ( int i, BDChannel bdc )      {
    bdchans [ i ] = bdc;
 }
 CHandler  getStop void() & Channel  sendStop() {
   return;
 }
}
class   BDChannel   {
  CHandler  Receive object()
            & Channel Send (object obj )  {
             return  ( obj );
 }
}
class DistribProcess   {
 movable Start ( int  i, Channel  (int, BDChannel)
                          sendBDC, Channel () sendStop ) {
  //  i is a process  proper number
  int   j;
  BDChannel  bdc = new BDChannel();
  sendBDC ( i, bdc );
  BDChannel[]   bdchans  =
     (BDChannel[]) bdc.Receive();
  //   Send messages to other processes
  for  ( j = 0; j < bdchans.Size; j++ )
    if  ( j != i )
     bdchans[j].Send (“Message from process “ + i +
                            “ to process “ + j                   );
   //   Receive messages from other processes
   for  ( j = 0; j < bdchans.Size; j++ )
    if  ( j != i )
     Console.WriteLine ( “Process “ + i + “ : “ +
                  (string) bdchans [ j ].Receive() );
   //   Send stop signal to the main program
   sendStop();
  }
}
```

4. IMPLEMENTATION

All described above is the development and improvement of the ideas from [GS03]. Therein, the functions of the channel message handlers were shared by the synchronous methods in the chords and the special built-in objects, called “bidirectional channels”. Below, we describe the current implementation based on bidirectional channels.

The implementation of MC# language consists of

1) a compiler from MC# to C#, and
2) a Runtime-system.

The compiler’s main function is to replace movable methods calls by queries to the Runtime-system which schedules (selects a location of) execution for the methods. Translating the chords is conducted mainly in the same way as in Polyphonic C#, using bitmasks to mark the presence of received channel messages. Once a bitmask is filled up, received message content is extracted and the chord body execution starts. In this part of the compiler, the mechanism of monitors implemented in the .NET class *Monitor* is relied on heavily.

The MC# compiler performs two passes: at the first pass, it gathers information about channels declared by the chords and at the second pass, it emits C# code including, in particular, the needed objects and methods to deal with the channels. Specifically, the compiler is implemented using the ANTLR parser generation framework (http://www.antlr.org).

The main components of the Runtime-system are:

1) *Resource Manager* — a process implementing (currently, the simplest) centralized scheduling of resources (mainly, the cluster nodes) and running on the cluster frontend, and
2) *WorkNode* — a process running on each cluster work node.

Besides, there are *mcsboot* and *mcshalt* utilities to start and terminate the Runtime-system, correspondingly.

The main purpose of the *WorkNode* process is to accept the movable methods scheduled for execution on the given work node and to run them in separate threads. Before running, it deserializes the object associated with a movable method and the method's arguments. The *WorkNode* process has, as a component part, a *Communicator* process running in its own thread. *Communicator* is responsible for receiving and delivering the channel messages intended for objects located on the given node. For this purpose, all objects having channels (and handlers) are registered in a special table located on the node. Thus, a channel message has the following format to ensure proper message delivery:

< (IP-)address, *Communicator* port, object number,
channel name, message content >

The compiler and the Runtime-system run under both Windows and Linux. For the latter we use the Mono system (http://www.mono-project.com) — a free implementation of .NET framework for Unix-like systems.

By way of experiments, we have written a large series of parallel programs in MC#, such as calculation of Mandelbrot set (fractals), 3D rendering, Web search through the Google Web-service, radar-tracking signals processing, solving computational molecular dynamics tasks, etc. Running these tasks on the cluster, we used up to 96 processors. For all mentioned applications, we got an easy to read and compact code and satisfactory results in terms of the efficiency of parallelizing. The graph on Fig. 8 shows the relationship between the processing time (in sec.) for a 40 Mb input file and the number of processors in the radar-tracking signal processing task. The tests were conducted on the "SKIF K-1000" cluster (98th in Top500, November 2004) of the United Institute of Informatics Problems, National Academy of Sciences of Belarus.

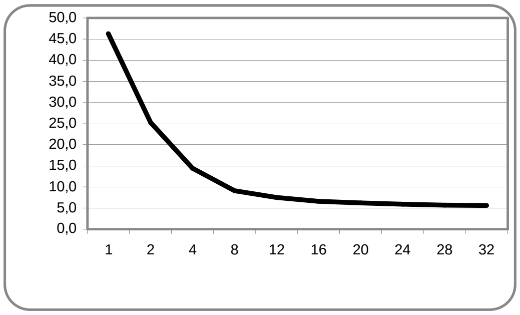


Figure 8. Processing time for 40 Mb radiohologram

5. CONCLUSIONS

This work presents an extension of C# language with the high-level features for concurrent, distributed programming based on the asynchronous programming model of Polyphonic C#. It can be considered as a general-purpose language for practical industrial programming, which oriented towards creating complex parallel software systems intended to run on cluster architectures.

We built a prototype implementation of MC# language for Linux cluster and a network of Windows machines. (The MC# project site is at: http://u.pereslavl.ru/~vadim/MCSharp)

Our future work will focus on implementing the MC# language in full accordance with the ideas put forward in the paper. Along with that, we are going to develop a more efficient Runtime-system by implementing a decentralized scheduling of movable methods calls and providing support for modern fast interconnects (Infiniband, QsNet II). A version of MC# programming system for metacluster computations is under development.

ACKNOWLEDGMENTS

The author wishes to thank Vadim Guzev and Alexei Molodchenkov for participating in the implementation of MC# programming system and applications for it.

REFERENCES

[BCF04] Benton, N., Cardelli L., Fournet C. Modern Concurrency Abstractions for C#. ACM Transactions on Programming Languages and Systems, Vol.26, No.5, 2004, pp. 769-804.

[BMS05] Bierman, G., Meijer, E., Schulte, W. The essence of data access in Cω. ECOOP 2005, LNCS 3586, Springer, 2005. pp. 287-311.

[FG02] Fournet, C., Gonthier, G. The join calculus: a language for distributed mobile programming. In Proc. Applied Semantics Summer School, 2000. LNCS, Vol.2395, Springer, pp. 268-332.

[GS03] Guzev, V., Serdyuk, Y. Asynchronous parallel programming language based on the Microsoft .NET platform. PaCT-2003, LNCS, 2763, Springer, pp. 236-243.

[S06] Serdyuk, Y. A formal basis for the MC# programming language (to appear).

[MPI] Message Passing Interface: http://www-unix.mcs.anl.gov/mpi/

[OpenMP] OpenMP specifications: http://www.openmp.org/specs.

[SJ05] Saraswat, V.A., Jagadeesan R. Concurrent Clustered Programming, CONCUR 2005, LNCS 3653, Springer, 2005, pp.353-367.