Embedding Mono code in unmanaged applications on GNU/Linux

LUNDS UNIVERSITET
Lunds Tekniska Högskola

LTH School of Engineering at Campus Helsingborg
Department of Computer Science

Bachelor thesis:
Martin Arvidsson
Viktor Hermansson

LTH School of Engineering
Lund University
Box 882
SE-251 08 Helsingborg
Sweden

LTH Ingenjörshögskolan vid Campus Helsingborg
Lunds universitet
Box 882
251 08 Helsingborg

Printed in Sweden
Media-Tryck
Biblioteksdirektionen
Lunds universitet
Lund 2011

Abstract

In today's society more and more work is carried out with the help of different computer systems. To benefit from the data, integration between the systems is needed. Saab has developed a solution to the problem, by the name WISE. With a modular design costs can be minimized, because a new integration does not necessarily require new software, but can be achieved with configuration of an existing module. (a so-called driver).

By supporting languages on a higher level than C++, development of new drivers can be speeded up to further decrease the costs. As a first step C# support was implemented with the help of C++/CLI. Such a solution is constrained to the Windows platform. To be able to meet the customers need for Linux compatibility this project was initiated, to create a wrapper driver with the help of Mono.

In the report it is shown that it is fully possible to create a working embedding of C# with the Mono runtime. The documentation of the limited embedding-API is however inadequate, this resulted in us having to investigate the functionality by creating small test cases and read the source code to see how function calls behaved. We have implemented a working wrapper driver which with the help of Mono enables WISE to start up and call C# applications. To verify the functionality in the wrapper driver, we have implemented a test driver which creates all data types known by WISE and sends them between C# and WISE. To test our wrapper driver in a real scenario, we have developed a C# chat driver.

Keywords: C#, Mono-Project, Marshalling, Embedding, WISE

Sammanfattning

I dagens samhälle utförs allt mer av arbete via olika datasystem. För att kunna dra nytta av all data behövs integration mellan systemen. Saab har tillverkat en lösning, vid namn WISE på detta problem. Genom en modulär design kan kostnaderna hållas nere, då en ny integration inte nödvändigvis behöver ny programvara utan löses via en konfiguration av befintlig modul (s.k. driver).

Genom att ge stöd för språk på högre nivå än C++, kan utvecklingen av drivers snabbas upp och på så sätt ytterliggare krympa kostnaderna. Som ett första steg implementerades stöd för C# m.h.a. C++/CLI. En sådan lösning medför att koden är låst till Windows-plattformen. För att kunna tillgodose kundernas behov av Linux-kompatibilitet påbörjades det här projektet, att skapa en wrapper-driver m.h.a. Mono istället.

I rapporten kan man läsa att det är fullt möjligt att skapa en fungerande inbäddning av Mono-runtimen. Dock är dokumentationen till, det begränsade embedding-API:et bristfällig, vilket resulterade i att vi fick med hjälp av enkla testfall undersöka hur funktioner betedde sig. Vi har implementerat en fungerande wrapper-driver som med hjälp av Mono-biblioteket möjliggör att WISE kan starta upp och kalla på C#-applikationer. För att verifiera att wrapper-drivern fungerar som den ska har vi implementerat en test-driver som skapar upp alla de datatyper WISE har stöd för och skickar dem mellan C# och WISE. Vi har också utvecklat en C# chat-driver för att testa ett verkligt scenario med hur wrapper-drivern är tänkt att användas i framtiden.

Nyckelord: C#, Mono-projektet, Marshalling, Inbäddning, WISE

Contents

1 Introduction

1.1 Background

Saab AB is a Swedish company with headquaters in Stockholm. The company's main focus is military defence and civil security. Saab was founded in 1937 and was originally an aeroplane manufacturer. Today Saab has 12,500 employees located on all continets with sales amount to around SEK 24 billion, where development and research acounts for 20 percent of sales.[1]

Saab is divided into five bussiness areas: Aeronautics, Dynamics, Electronic Defence Systems, Security & Defence Solutions and Support & Services. Aeronautics researches and builds airbourne crafts. Dynamics offers combat weapons as missiles and torpedoes. Electronic Defence Systems offers systems for surveilance and threat detection. Security & Defence Solutions develops technology to protect propety and individuals. Support & Services provides support for Saab products.[1]

The office in Helsingborg where this thesis has been written is in the area of Security & Defence Solutions. The subdivision located in Helsingborg is called Saab Training Systems and works with training for both military and civil purposes. They develop software which improves training scenarios, both virtual and real life. Both Saab Training Systems and Saab will further be refered to as Saab in this thesis.

Saab has developed an integration platform called WISE (Widely Integrated Systems Environment) which enables systems to exchange information without modification of the communicating systems. WISE can be used to have many different virtual training simulators work together, eg. a tank simulator can communicate with a helicopter simulator. A system communicate with WISE via a driver. The driver translates messages to and from WISE. For convenience it is desirable to be able to write this driver in any chosen language. WISE on Windows have C# driver compatibility but customers increasingly inquire for C# Linux compatibility and to meet the customers needs, Saab need to develop new wrapper drivers to enable drivers to be written in C# for the Linux platform.

1.2 Scope

1.2.1 Goals

The main goal is to develop a wrapper driver to WISE which enables further driver development in C# as well as documenting the compatibility between .NET and, the open source implementation, Mono. To test if the wrapper works properly a chat and test driver is designed.

Main goals for the wrapper driver are (listed in priority):

1. Correctness (Load and convert data to C# drivers)
2. Easily managed code
3. Driver compatibility with .NET
4. Performance (speed of execution)

Goals for the test driver:

- Test functionality in the wrapper to assure the first goal of the wrapper driver.

Goals for the chat driver:

- Get an understanding of how customers interact with the WISE API.
- Test the wrapper with a real application.

Goals for the compatibility documentation:

- Give Saab an insight in the current and future status of Mono.

The purpose of the project is to lower the cost when developing new drivers for WISE.

1.2.2 Scope Details

The project has a target machine and support for other systems is not required. The target machine is a 32 bit Ubuntu 10.04 system with GNU Compiler Collection (GCC) version 4.3.2. The GCC version is a requirement from WISE. What Mono version to use is a result from the testing in this project but the latest long-term supported version (2.6.7) was initially chosen. Even though Mono supports other languages than C# no other languages have been considered, and all references to managed code refers to C# code in this document.

Despite that the project has a target machine, the wrapper driver is frequently tested in Windows XP and reflections about this system is also considered in this document.

1.2.3 Acceptance requirements

The requirements from Saab are:

- Create a wrapper driver that can load C# drivers.
- Create a chat driver in C#.
- Create a test driver in C#.
- Document differences between Mono and .NET.

1.3 Method

The workload is divided into three different phases. The work will not be isolated to the specific phase and probably alternate between a couple of phases during the project's timespan.

1.3.1 Phase 1

This is an information gathering phase. Information is gathered in two forms by reading documentation and experimenting with short program snippets. The focus lays on the different possibilities to communicate between managed and unmanaged code. We investigate both extending C# code with unmanaged libraries and unmanaged programs with a C# extension.

1.3.2 Phase 2

In this phase the focus is on WISE, how WISE works and how to embed Mono in this environment. In the beginning, Saab is hosting a course in driver development to get a kick-start on the phase. This is followed by the actual wrapper-driver implementation. When this is done two C#-drivers are constructed, one test-driver and a chat-driver. The test-driver is for extra validation of functionality and the chat-driver is a proof-of-concept for the customer, Saab. The development model will be an iterative model where we will alternate between pair programming and single programming depending on the task.

1.3.3 Phase 3

The last phase is partially focused on quality assurance and partial documentation. More testing will be done than in the second phase, some newly discovered bugs will be squashed. The experiences from earlier stages are documented in this report for further development and may be of assistance in similar projects. Another goal in this phase is to document the differences between Mono and Microsoft .NET.

1.3.4 Source criticism

For the information in this thesis we have tried to find as reliable sources as possible. Information about Saab, Microsoft and .NET is used from respective companies own web pages and we assume this information is correct. Most information about Mono is taken from Mono's official web page (`http://mono-project.org`) as well as the older page (`http://go-mono.com`). The information on the older site could be outdated but another more reliable source about Mono is not available. Two of our sources ([11], [20]) are from a forum on the Internet, we have carfully chosen who to cite, the former is from a person who we do not know about and can not prove his knowledge but he seems to know what he's talking about. The latter is from Miguel de Icaza, one of the main developers of Mono, we assume he is a reliable source.

2 WISE Connectivity

WISE is a software suite and a generic integration platform that allows connection of systems or applications into a common environment and creates an information flow between them.[2]

2.1 WISE vs. conventional integration

Integration between systems can be very problematic and time consuming. To get two systems to communicate with each other, one of them has to be modified. For every system joining the connection the complexity of the integration drasticly increases.

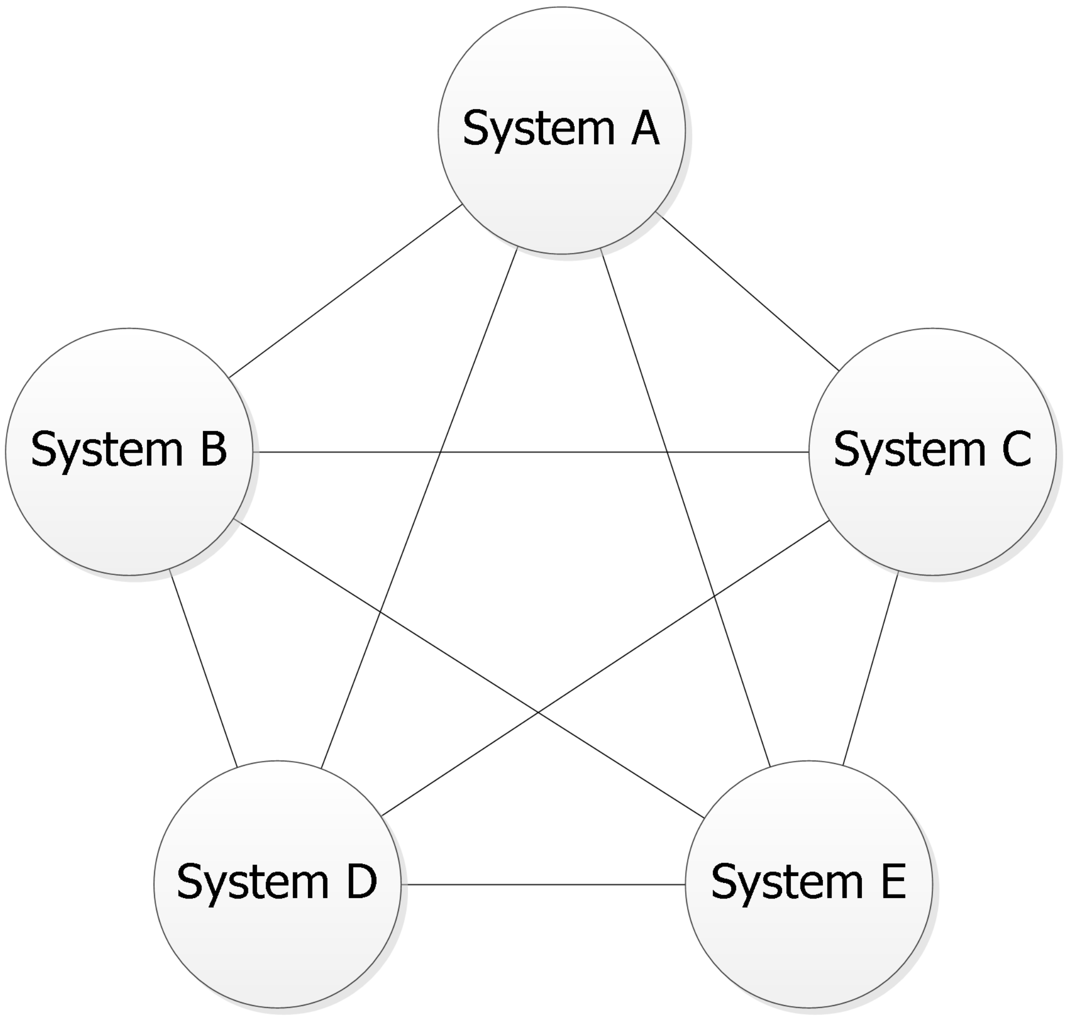

Figure 1: Conventional integration.

In conventional integration between systems the integration takes place in the integrating systems. This means they have to be modified to be able to communicate with each other. With WISE the integration point is inside the WISE runtime with a WISE driver for each system. The WISE driver translates the application specific messages to a common information model, allowing the system to communicate with WISE in its native language thus eliminating the need for modifying the system. Once a driver is written for a specific application protocol, it can be reused in other systems using this application.

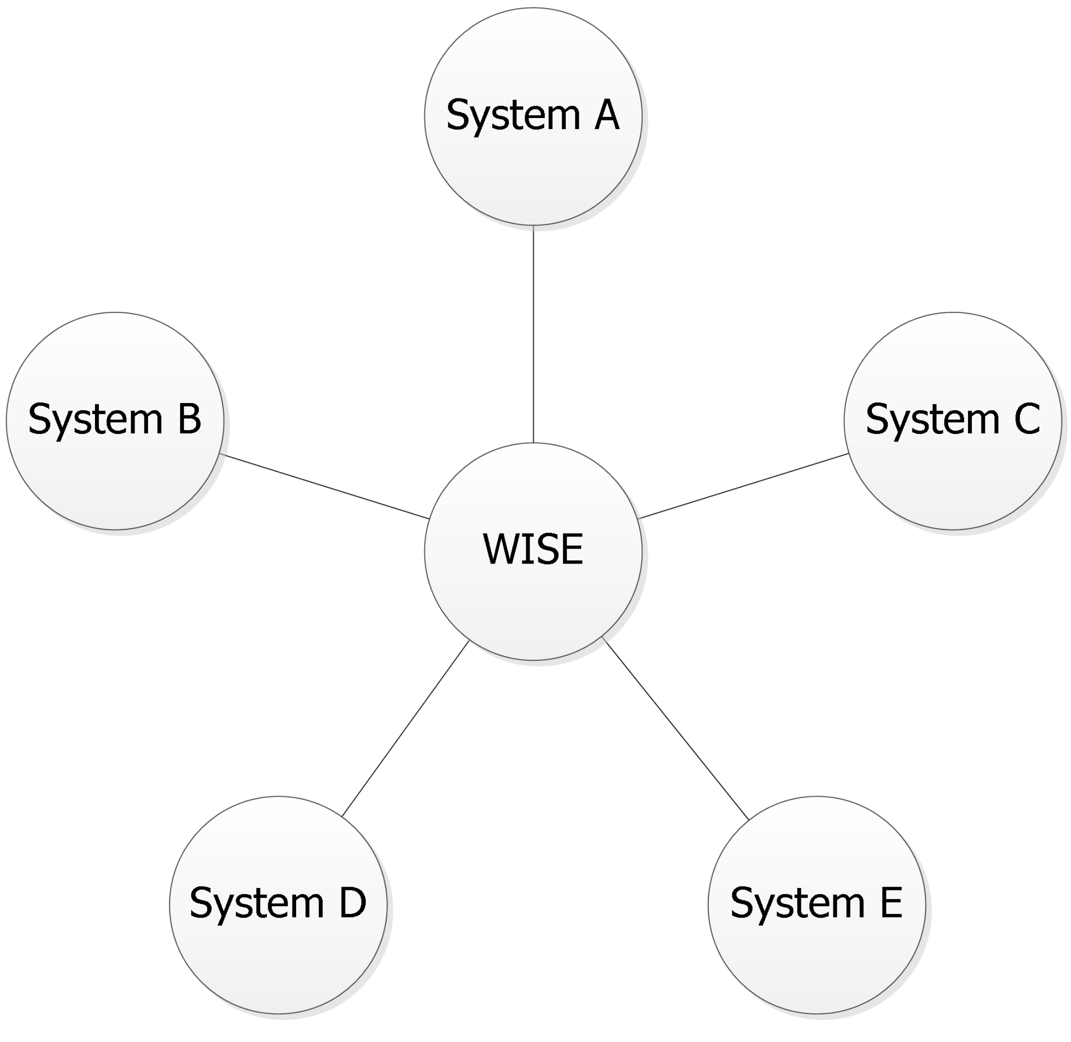

Figure 2: WISE integration.

2.2 WISE components

WISE is an environtment which consists of different applications and components, they are:

CoDE WISE Connectivity Designer Edition (CoDE) is the application where connections between systems are configured. A database to each system is created here as well as the connection between different databases.

CoRE WISE Connectivity Runtime Edition (CoRE) is the application used when running a connection between two or more systems in WISE.

Test Tool WISE Test Tool is used to verify the information flow between systems. Test Tool can be used to create new object and events in one system and check if they are correctly distributed to the other systems.

Connectivity SDK The WISE Connectivity SDK is used by programmers when creating new drivers for WISE. The SDK includes project templates for Microsoft Visual Studio so programmers gets a correctly configured project right away.

3 Microsoft .NET Framework

The .NET Framework is a Microsoft technology for Microsoft operating systems. The idea is to compile application code into a Common Intermediate Language (CIL) which is interpreted by the Common Language Runtime (CLR) during execution. Applications can be written in any of the .NET languages and because of the CIL link to each other without restrictions.[3]

The .NET Framework has two main components: the Common Language Runtime and the .NET Framework Class Library. The CLR is an execution environment where CIL code is executed in a similar way as Java is executed in the Java Virtual Machine. Code targeting this runtime is referred to as managed code. Native or unmanaged code refers to code compiled into architecture specific binary code.[4]

The Class Library is an object-oriented collection of reusable types, which speeds up and simplifies the development process of new applications.

3.1 Some components in .NET

ADO.NET
ADO.NET with its subcomponents Entity Framework (ET) and Language-Integrated Query (LINQ) serves as an abstraction layer between the application-code and the database. With this layer it is possible to change the underlaying database manager system without major rework of in the application layer.

Windows Communication Foundation (WCF)
In a Service-oriented architecture (SOA), where the functionality is split up to a number of different services, puts great demand on the integration. WCF is a framework to simplify and unify different kind of communication.

Windows Workflow Foundation (WF)
With WF the program is design in a flowchart, the different building blocks is called activities. Some examples of activities: ForEach, ReceiveMessage and Sequence. If these standard activities does not fit; there is a possibility to create custom activities. When the program is executed the WF-engine takes care of initializing, threading and holding

the different states of the program.

Windows Presentation Foundation (WPF)
In modern application there is a need for more advanced graphics, with WPF is it possible to create 3D-graphics, Vector-graphics and animations. Underlaying implementation is based around Extensible Application Markup Language (XAML) to describe the layout and forms.

3.2 Comparison between C# and Java

3.2.1 Some examples of differences in syntax

Description	C#	Java
Call the base-class's constructor	Base()	super()
Convert a string (s) to integer	int.Parse(s)	Integer.parseInt(s)
Declaration of a constant	const double PI=3.14	final double PI=3.14
Foreach-loop	foreach(int i in numArray) { sum += i; }	for(int i : numArray) { sum += i; }

As shown in above examples there are some minor changes in syntax. One thing to notice is that method-names begins with a capital letter, the opposite to the Java Code Convention.[5]

3.2.2 Some useful features missing in Java

Operator Overloading With operator overloading it is possible to change the behavior of these operators:
Unary operators: `+, -, !, ~, ++, --, true, false`
Binary operators: `+, -, *, /, %, &, |, ^,` <<, >>, ==, ! =, >, <, >=, <=

Implicit and explicit cast
In C# it is possible to define how the cast between different classes should be handled. This is an explicit cast: `byte b=32; int i=(int)b;` and implicit: `byte b=32; int i=b;`. In listing 3 there is an example which shows some of the possibilities with this constructs. This example shows how two unrelated classes (no common base-class) can be converted

Listing 1: Example of Implicit cast:

```
class Volvo {
  public string name;
  public Volvo(string val) {
    name= val;
  }
  public static implicit operator SodaCan(Volvo car)
      {
    return new SodaCan("Coca cola: now with gasoline
        flavor");
  }
}
class SodaCan {
  public string name;
  public SodaCan(string val) {
    name= val;
  }
}
class myProgram {
  static void Main() {
    Volvo v = new Volvo("S60");
    System.Console.WriteLine("Our car: " + v.name);
    SodaCan specialSoda = v;
    System.Console.WriteLine("Our special soda
       flavor: " + specialSoda.name);
  }
}
```

Properties
To access variables in an object there is two common variants; public member (`obj.name`) or private member with an access-method (`obj.getName()` and `obj.setName(string s)`). With the latter the assignment can be made more safe with validation of the new name for an example make sure there is not another object with the same name.

In C# there is something called properties for this. The programmer declares `get` and `set` methods for the member and the result is as simple syntax as with a public member and the flexibility as with the handmade access-method.

4 Mono

Mono is an open source implementation of Microsoft's .NET Framework. Its goal was to embrace a successful, standardized software platform to lower the barriers when producing software for the Linux platform.[6] The Mono project was started in 2001 with the first public release in 2004. Since then Mono has developed rapidly, possibly because economic support from Novell. Mono can be run on a wide variety of systems including the three main desktop OS's: Linux, Windows and Mac OSX. The GUI systems for each OS platform has been ported to the others, giving Mono developers the possibility to target the system of their choice but still operate on the others.

4.1 Licensing

The Mono project uses four open source licenses for different parts of the project. They are: Massachusetts Institute of Technology license for the X Window system (MIT/X11), General Public License (GPL), GNU Lesser General Public License 2.0 (LGPL) and Microsoft Permissive License. GPL and LGPL are the least permissive and therefore are the ones limiting the usage of the library. The Mono tools are licensed under GPL while the runtime library is licensed under LGPL. This means the Mono-library can be dynamically linked into commercial applications, but modifications to any Mono code requires the changes to be made open, unless a license from Novell is obtained.[7] The parts licensed with Microsoft Permissive License are parts borrowed from the Microsoft Class Library, released as open source by Microsoft.

4.2 Applications using Mono

Linux enthusiasts have a tendency to dismiss Microsoft technology as well as other commercial technologies. Mono is therefore not widely used in open source applications. Mono has another usage though, that is business or commercial applications originally written for Windows where the companies try to expand to the Mac or Linux platform. Three popular applications and games are[8]:

- Electronic Arts' The Sims 3.
- Lego's Quest for R2-D2. - A game for iPhone.
- Tomboy - An open source desktop note-taking application for GNU/Linux.

4.3 .NET compatibility

Having full compatibility between Mono and .NET is desirable from the users perspective but as a result of the Mono project being open source, the mono-team does not develop features they feel are less important. This issue and that some features are simply not portable to other platforms have affected the compatibility with .NET.

The official statement on Mono-version 2.8 is that it supports everything in .NET 4.0 except: [9]

- WPF - Windows Presentation Foundation
- Entity Framework
- WF - Windows Workflow Foundation
- some parts of WCF - Windows Communication Foundation

With Mono-compatible code it is possible to compile code in either Microsoft's or Mono's compiler, on either Windows or Linux. Both of the executables (or libraries) are portable to either runtime environment.[11].

4.3.1 MoMA-tool

One way to check if a .NET assembly is compatible with the Mono runtime is to use "Mono Migration Analyzer" - MoMA. This tools loads a user specified assembly and tries to determine if it uses non mono-compatible calls. MoMA is not perfect and might result in false positives or pass unsupported code.[12]

5 Embedding Mono

Embedding is generally containing something in a surrounding mass but the name can me misleading when it comes to embedding software. This is because the embedded application still have full communication with the world.

Embedding lets a host application start up another application which is in some way incompatible with the host. This is achieved with the help of an embedding library. Different libraries exist for different purposes. The reason to embed applications is usually to take advantage of more modern programming languages in older applications, to embed new parts of the application is cheaper than rewriting the application in the new language. Another reason is to use the best of two (or more) worlds by building time-critical parts of an application in a low-level language and the rest in a user friendly high-level language.

When embedding Mono code in native code the Mono runtime library is used. The API consists of function calls to C code and usually look like `mono_X_get_Y(X)`. The fact that the API consists of C calls makes the usage of the API quite inelegant, the order in which commands are executed is reversed compared to the today common object oriented style of programming.

Listing 2: Object oriented vs Procedural:

```
//OOP
human.getAge()

//Mono C-style (Procedural)
method = GetMethod("getAge");
InvokeMethodOnObject(human, method);
```

Embedding the Mono runtime consists of various steps

- Compiling and linking with the Mono runtime
- Initializing the Mono runtime
- Calling managed code
- Optionally expose C++ code to the managed world

5.1 Compiling and linking

To compile an application to embed Mono the application has to be linked with the Mono runtime libraries. The compiling and linking is as usual. The flags returned by pkg-config –cflags mono needs to be given to the compiler and pkg-config –libs mono to the linker.

5.1.1 Mono 2.8+

From Mono version 2.8 and forward the name to pkg-config is no longer mono, but mono-2 instead. Also Mono does not link to GLib internally anymore and the compiler flags have to be appended with GLib's directories (when used), commonly: -I/usr/include/glib-2.0 and -I/usr/lib/glib-2.0/include.

5.2 Initializing the runtime

The Mono runtime is initialized by a call to `mono_jit_init(name)`. The parameter is the name of the application domain. Application domains are used to isolate multiple applications on a single Mono virtual machine similar to processes in an operating system.[13] This call will initialize the default framework version for the current Mono version. The framework version can be specified to .NET 2.0 with a call to `mono_jit_init_version(name, "v2.0.50727")`. The initialization of the Mono runtime is only allowed once per application. The initialization calls return a pointer to an application domain.

When the application domain is open the actual C# code needs to be loaded, this is achieved with a call to `mono_domain_assembly_open(domain, name)`. The first parameter is the application domain and the second the path and name to the assembly (dll) being loaded. The call returns a reference to the assembly.

To be able to invoke member methods of a class in the assembly an instance of the class needs to be created. An image is the component of the assembly which holds the actual CIL code and can be extracted from the assembly by calling `mono_assembly_get_image(assembly)`. With the image and the knowledge of a class' name and namespace the class can be found with `mono_class_from_name(image, namespace, name)`. A call to `mono_object_new(domain, class)` will create an object whose definition is looked up using the class. The object is initialized by calling the constructor, this is done like calling any other method and is described in the following section. The name of the constructor is always `.ctor` when embedding Mono.

5.3 Calling managed code

Invoking methods in the managed world requires a number of Mono runtime calls. The class can be acquired as described in the previous section. The class is needed when obtaining the method with `mono_class_get_method_from_name(class, method_name, num_params)`. Calling managed code is done with `mono_runtime_invoke(method, object, params, NULL)`. The parameters are the method to invoke, the object to invoke it on and an array of pointers to the parameters to send to the method.

5.4 Exposing C code to the managed world

Being able to call code in the managed world is useful but the true power comes when the managed code can call back to the native world. This is especially useful when the C# code uses threading and wants to notify the C++ world of an occurring event later during execution. Mono uses `mono_add_internal_call(csharp_method_name, c_function_pointer)` to create a connection from a C# method to a function in C++. In C# the method has to be declared as:

```
[MethodImplAttribute(MethodImplOptions.InternalCall)]
static extern return_type MethodName(...);
```

All calls to this method is now forwarded to the C++ code. The programmer has to make sure that the formal parameters are compatible between the two connected functions since no parameter validation is performed.

6 WISE with Mono

This chapter describes the architecture and key components in the embedding of Mono with WISE. The goal is to help further development by giving a better understanding of the current solution.

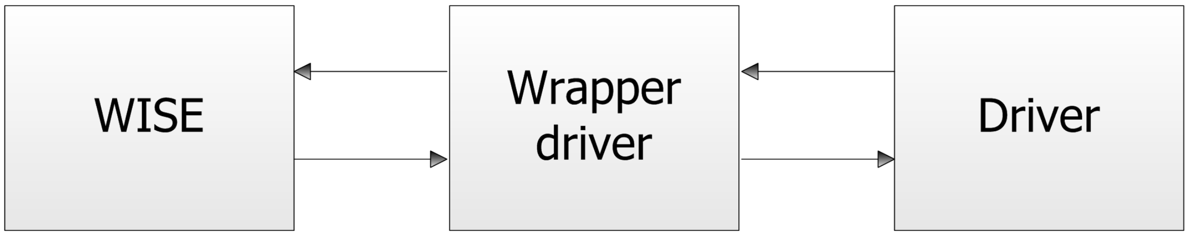

Figure 3: Overview of communication between WISE and an embedded driver.

The wrapper driver ("the wrapper") is the main component developed in this project. It, as well as WISE, is written in C++ while the driver is written in C#. The driver can be a customers driver or the chat/test drivers developed in this project. The wrapper exposes the same interface to the driver as WISE exposes to it. Therefore, the driver is unaware that it is not communicating directly with WISE. In C++ this interface is called `IWISEDriverSink` and in C# `INETWISEDriverSink`. The sinks are used when communication from right to left in the figure above. For communication in the other direction another interface is used, the driver interface. `IWISEDriver` in C++ and `INETWISEDriverSink` in C#. In the current implementation the Mono runtime library is used in the wrapper to communicate with the driver.

The wrapper driver is located in the middle of WISE and the loaded C#-driver where it interprets calls from both directions. One problem with the interpretation is to get a notice that something has happened and another is to convert and transfer data.

6.1 Design

With a good high-level design the low-level implementation is much easier. The first phase of the high-level design was to understand how WISE is designed and then create a checklist of the upcoming tasks.

When something is updated or in another way triggered in the WISE-runtime, a notification-call is made to the driver. This can for example be `OnAddObject(params, ...)`, this gives some basic knowledge about a new object that was created in the application database. To get the actual object the driver calls back to WISE with the sink. One minimal example of how to use the sink is presented in listing 3.

Listing 3: Example-code:

```
// Get the value of the attribute with handle 161
// and type long from
// object with handle 125
// in database with handle 101
long bValue = 0;
sink->GetAttributeValue(101, 125, 161, bValue);
printf("%u\n", bValue);
```

Figure 4 shows current high-level implementation and the classes used. `CMonoDriver` receives calls from WISE, marshalls data with the help of `Marshall` and passes it to the loaded C# driver (`ChatDriver` in this example). Marshalling means converting something so it can be transported. This driver communicates with the system it is designed for (not shown in figure). The C# driver uses `CMonoWISEDriverSink` to communicate back to the C++ world. `MonoDriverCallbacks` receives this data, marshalls it with help of `Marshall` and calls the corresponding method in WISE's sink. The classes are explained in more detail further on in this chapter.

Custom C# Helper library
A library in C# is developed alongside with the wrapper driver. This library contains helper classes to make the marshalling easier as well as the `CMonoWISEDriverSink` class. This library is referred to as the C# helper library.

CMonoDriver
This class receives calls from WISE and passes them to the loaded C# driver. In `CMonoDriver` there are 11 methods inherited from the `IWISEDriver` interface, this is the interface used by WISE to communicate with drivers.

The wrapper driver uses the same interface as normal drivers and the task in all methods except `OnInitialize` and `OnUninitialize` is to receive data from WISE, marshall it and pass it to the C# driver. Marshalling

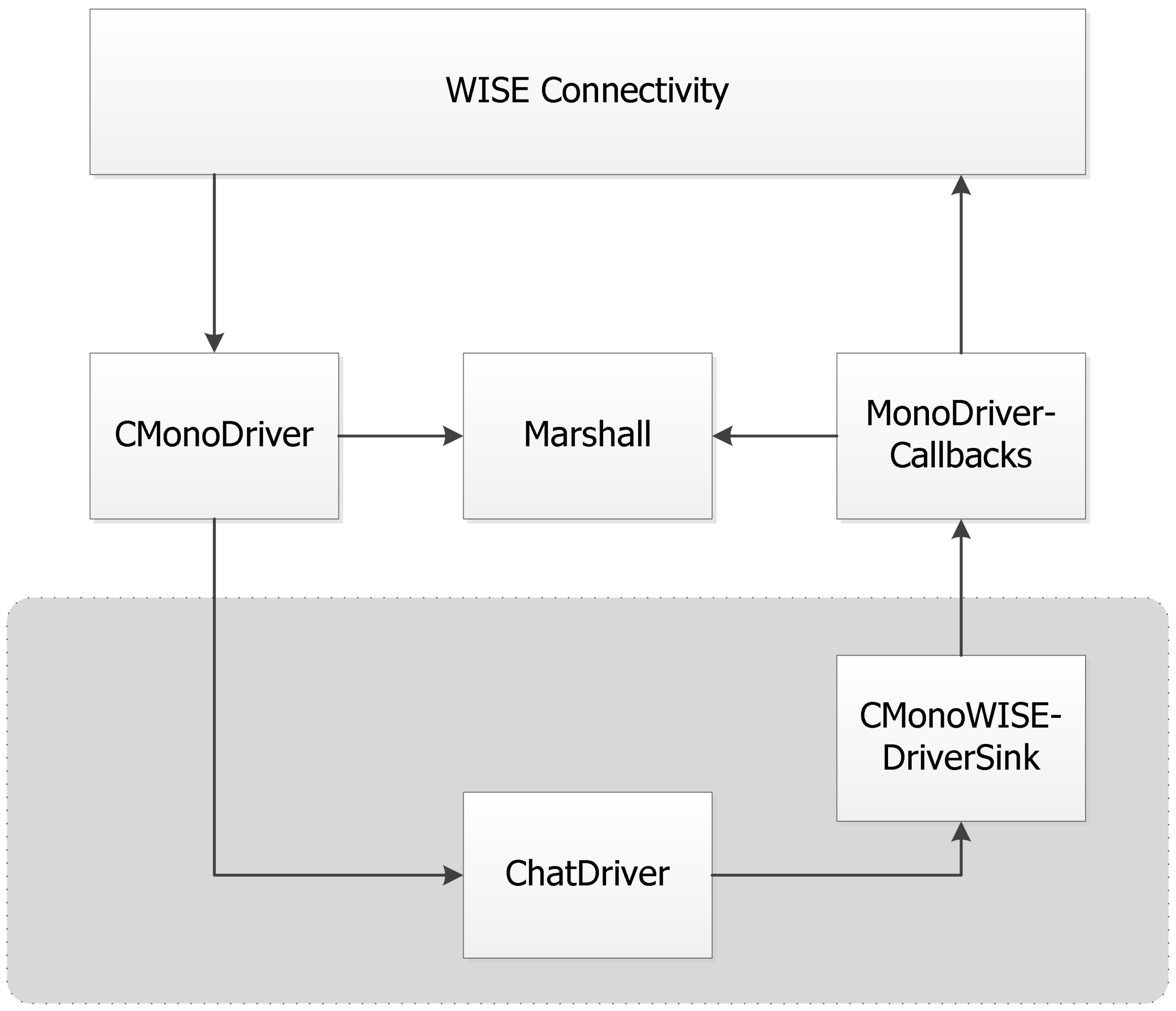

Figure 4: High-level design: The arrows illustrates in what directions the calls are made. Classes in they grey area is written in C#.

is forwarded to the `Marshall` class. `OnInitialize` reads settings from the configuration files including what C# assembly (dll file) to load as well as the search path to the Mono library. With this information the virtual machine is started and initialized as described in section 5.2.

The wrapper driver is written in C++ and for C# to be able to communicate back to C++ some specific code has to be written in C#, see section 5.4. The sink in the C# helper library contains this code and is loaded by the wrapper driver to avoid exposing it to the customer. An identifier to the current wrapper is also sent the sink to be used when the sink calls back to C++. This sink is set to an attribute in the customers driver, only the interface of the sink is shown to the customer. The methods in this sink is connected to `MonoDriverCallbacks` in C++ as described in section 5.4.

`CMonoDriver` stores the references to Mono's virtual machine so the other classes can reach it. Additionally `OnInitialize` finds all methods in the customers driver and saves references to them to gain performance during execution. `OnUninitialize` closes the C# driver and cleans up the virtual machine.

MonoDriverCallbacks
This class contains 238 static methods used by the C# sink to communicate with C++. In every method the data from C# is marshalled and forwarded to the WISE sink. Sometimes WISE is supposed to update parameters sent to it. In C# these parameters are sent as `ref` parameters. When receiving `ref` parameters in `MonoDriverCallbacks` the parameter is prefixed with an extra * (pointer). This results in that MonoObjects is handled as a pointer-to-pointer variable and native data types is handled as a pointer. This gives the programmer maximum flexibility and can either assign a new object by dereferencing the pointer-to-pointer or simply use the dereferenced pointer to get the address to the MonoObject.

WISE can run multiple drivers in the same runtime at once. This affects the design of the wrapper in some aspects. `static` variables and methods will be shared between driver instances which also means problems with thread synchronization. Static's are avoided as much as possible but is sometimes required as with the methods in `MonoDriverCallbacks`. Since they are `static` the methods do not have any reference to any running driver instance. This is solved with a static pointer to a class (`DriverInstances`) which holds all driver instances. `MonoDriverCallbacks` gets an identifier from the C# sink of what driver instance to call. The identifier is used to get a pointer to the correct instance from the static `DriverInstances`. This class is thread synchronized with the help of synchronization classes from the WISE library.

Marshall
The Marshall class has a method for every data type that needs to be marshalled. It is overloaded to work in both directions. For lists and dictionaries template methods are used, this is because the marshalling is very similar, only a single or few rows change between different data types. Therefore these rows have been moved out to overloaded methods so templating can be used. More about marshalling is found in the following section.

6.2 Marshalling

To be able to transport data between the managed- and unmanaged-environment some type of conversion is needed, this is called marshalling. Some basic data types are marshalled automatically but more advanced types requires to be manually handled by the programmer.

WISE on Windows already have C# driver support, this means Saab already have data types in C# that are equivalents of the C++ types handled by WISE. These types are found in the `STS.WISE` namespace.

This chapter discusses how marshalling is done in our wrapper driver. First, common types are explained followed by WISE's custom data types.

6.2.1 Native data types

WISE uses 4 native data types: `long`, `long long`, `unsigned char` and `double`. Additionally a type `WISE_HANDLE` is used for storing ID's. This type is a type definition for a `long` and is treated in the same way as a `long`. The data types have different sizes in C++ and C#. The C# equivalent of C++ `long` is `int` and C++ `long long` is `long`. `unsigned char` is stored as `System.Byte` in C#.

Depending on the context, data is marshalled in different ways. Usually native data types do not need any marshalling, the only problem is to make sure the data types in the two environments are of the same size and have the same bit format. Floating point values have the same bit format in both C++ and C#. The time when a native data type need marshalling is when creating or reading the C# data type `System.Object` or when accessing properties in classes. Sending or receiving these types through parameters in function calls do not need any manual marshalling.

Objects of native data types from C# are accessed from C++ via the `MonoObject*` type. To be able to read the value this object points to, it has to be unboxed with the `mono_object_unbox(MonoObject*)` function, properly typecasted and dereferenced, eg.

```
long k = *(long*)mono_object_unbox(mono_object_of_long)
```

When creating a `MonoObject*` from native C++ data types the value has to be boxed with `mono_value_box(MonoDomain*, MonoClass*, void*)`. The parameters are the domain, the C# data type class to create and a pointer to the value being converted.

```
long k = 5;
mono_value_box(domain, mono_get_int_class(), &k)
```

6.2.2 String

WISE uses `std::wstring` to store string data. The data type is usually wider than normal `std::strings`, each character is 2 bytes wide on Windows[14] and 4 bytes wide on Unix-like systems.[15] In C# the `System.String` class is used which is a UTF-16 formatted string.[16]

`MonoString*` is the unmanaged representation of the `String` managed type. Since the `wchar_t`'s size is different on different computer systems, a type with a fixed size is used when sending and receiving data from Mono. This is to enable both Windows and Linux compatibility.

The `MonoString*` is converted to a `short int` array by using the function `mono_string_to_utf16(MonoString*)`. From this a heap-allocated `wchar_t` array with dynamic length is constructed and populated with the elements from the first array where each element is typecasted to `wchar_t`. This is later used in the constructor of `std::wstring` to create the final WISE compliant string.

The process to create C# strings from C++ is very similar, only reversed. A `short int` array is populated from a `std::wstring` and used in the function `mono_string_from_utf16(const short int*)` to create a `MonoString*`.

6.2.3 DateTime

For time and date representation WISE uses `timeb`. This is a standardized type in C++. It has an attribute of type `time_t` which contains the number of seconds since 1970-01-01, also known as Unix time. `timeb` has other attributes such as timezone but these are not used by WISE. In C# the standard type for storing time and date is used, this type is called `System.DateTime`. It contains a 64 bit integer where the two upper bits have special meaning, they describe if the time should be interpreted as local time or Coordinated Universal Time. The other 62 bits represent every 100 nanoseconds since the birth of Jesus Christ (0001-01-01 00:00:00).

To make sure that the `DateTime` is passed as a 64 bit integer, the method `ToBinary()` is executed in C# before the `DateTime` gets passed to the unmanaged method. The `ToBinary()` method returns the actual 64 bit data inside the `System.DateTime`. The conversion between the two date formats are as simple as making some calculations. To convert from `DateTime` to `time_t` the following lines of code is used (Listing 4):

Listing 4: DateTime to timeb:

```
long long datetime; // data from C#
long long unixtime; // temp variable
timeb time;

unixtime = (datetime & 0x3FFFFFFFFFFFFFFFULL) /
        10000000ULL - 62135596800ULL;
time.time = unixtime;
```

This line filters out the least significant 62 bits, converts 100 nanoseconds to seconds and subtracts the seconds between year 1 and 1970.

The conversion from `timeb` to `DateTime` is very similar. First do the math from Unix time to 100 nanoseconds since year 1. This 64 bit integer value is passed to the managed world and can be interpreted as a `DateTime`.

6.2.4 Vec3

WISE uses a class `CWISEVec3` to store positional information. It is a WISE specific class which consists of three `doubles` (v1, v2, v3) and a 32 bit `enum` (compareAND, compareOR). The C# equivalent type is `STS.WISE.Vec3`, a `struct` with the same attributes.

When embedding Mono, structs are passed by value and can therefore be treated as the native types. The only problem is to create a compatible container in C++ to hold the data. `CWISEVec3` is not compatible because it contains more additional information than the 4 data types. Therefore a custom made `struct MonoVec3` is used as intermediate storage. To get the data into a CWISEVec3 a new object is created with the data as parameters in the constructor.

```
CWISEVec3(monovec.v1, monovec.v2, monovec.v3, monovec.c);
```

When a C# struct contains strings or references to objects they have to be read as MonoString*'s and MonoObject*'s respectively and marshalled properly.

6.2.5 Blob

Blob stands for binary large object and is used by WISE to store information not being compatible or practical with the other types. In C++ the type is named CWISEBlob, it is a class with an unsigned char array, the size of the array as well as two std::wstrings. The strings are used to describe the data by specifying mimetype and encoding. In C# a class STS.WISE.Blob is used to store the same information. This Blob uses the System.IO.MemoryStream class to hold the array.

Marshalling CWISEBlobs to Blobs consists of several steps. First the C# helper library is accessed to get the class of the Blob implementation. An object of this class is instantiated. The constructor of Blob takes a byte array as well as the two describing strings, therefore a managed array is constructed with the help of mono_array_new(domain, class, size) . Instead of copying the unmanaged array element by element, a low level memory copy is used. This will save a lot of overhead since the managed world isn't called on every element and is possible because both of the arrays are stored sequential in the memory. The string conversion is delegated to the string marshalling methods see section 6.2.2.

When converting Blobs to CWISEBlobs the properties of the class are accessed and marshalled by forwarding the job to the appropriate methods. A CWISEBlob is created and its array is set with a memory copy in the same way as describer in the previous paragraph.

6.2.6 Union

The WISE data type CWISEValueUnion is used for storing any of the other data types, but a maximum of one object at a time. Saab has no

C# equivalent type but uses the System.Object which is the type all objects in C# inherits from and can therefore hold any type of data. In C++, assignment operators are used to enable what data types to support. These sets the value as well as an enum which describes the data type currently being stored.

The marshalling between CWISEValueUnion and Object is very straight forward. When converting to Object the method GetValueType() in CWISEValueUnion is called to determine the current type. After this a switch statement evaluates the type and the correct marshalling method is called.

The other way around the mono_class_get_name(MonoClass*) is invoked to determine the type. When the type is known the marshalling is delegated to the correct method for the current type.

6.2.7 Lists

Every WISE data type except Blob can be stored in a list. In C++the std::list is used and in C# custom classes that extends the System.Collections.Generic.List are used.

To extract the values from a C# list in C++ code we first create an empty MonoArray. With this array as a parameter, the CopyTo method is executed on the C# list. CopyTo will populate the array with the data in the list. In the array the data is easily accessible with the Mono-API, using methods like mono_array_addr or mono_array_get.

To marshall data back to managed code we loop over every object in the native list, convert each value to a managed data type and execute the managed command Add on the C# list with the converted value.

6.2.8 Dictionaries

System.Collections.Generic.Dictionary is the C# equivalent of C++'s std::map, meaning that you can store a key with a specific value. Dictionaries are used in two cases in WISE, when accessing settings passed to the driver and when the driver uses the data type AttributeGroup.

For the extraction from the C# object, is the solution very similar to the one for lists. First all the keys and values are copied to an array by calling the managed `CopyTo` method. Both keys and values are placed in the array. A problem arises when using this method on dictionaries, keys and values have different sizes. Therefore the `mono_array_get` method cannot be used in the same way as with `lists`. The array can be interpreted as an array of key/value pairs or as an array of plain data. Important to know is that elements with a size smaller than the system pointer size will be padded to the pointer size, eg. the managed type `System.Byte`. If the array is interpreted as an array of plain data, a pointer can be used to iterate through the array. It is up to the programmer to know how many bytes each element is and to increase the pointer with the correct size. Once an object has been retrieved from the array its marshalling method is called and the result inserted in the C++data type.

For the marshalling in the other direction there is a problem in how to create managed generic objects from C++.[17, 18] One solution is to use a helper class in C# that have a field of the constructed type of generic. This is the method used and the C# helper library is called to get the class of the `Dictionary` and instantiate an object of this.

Another way to resolve the issue with creation of a generic class in unmanaged code is to define classes which implements the specific generic structure.

When the object is created it is as simple as with the lists: we loop through the unmanaged map, convert every key and value to the Mono equivalent-class and then execute the `Add` method on the managed dictionary with the key and value as parameters.

6.2.9 AttributeGroups

Attribute groups are a WISE data type that stores dictionaries of every, known by WISE, data type. The class in C++ is `CWISEAttributeGroup` and the C# equivalent is `STS.WISE.AttributeGroup`.

The marshalling to C# consists of calling the C# helper library to get the class definition of `AttributeGroup` and instantiate an object of this. Thereafter go through every dictionary in `CWISEAttributeGroup` and forward them to the dictionary marshalling methods. The same procedure is used when converting from C#.

6.3 Templates

To be able to write as generic code as possible the Marshalling class makes frequent use of template methods. A template method allows a method to work on many different data types without having to be rewritten for each type. This makes the maintenance of the code easier as well as the risk of typing errors decreases. A drawback with this is the somewhat increasing difficulty when debugging. When the same code is used multiple times with different data types it's harder to make the application break at the desired point in execution.

Template methods are easy and powerful when the code is exactly the same for the different data types. This is not always the case and there is no straight forward way to create conditional statements to execute different code paths depending on the current type. We solved this by using overloaded methods as well as template specialisations. By using overloaded methods for marshalling single data types the template methods can call the overloaded methods which contains the unique code for each data type. Template specialisations works in the same way, except they are used when the formal parameters to the overloaded methods can't be distinguished from another.

Template methods in C++ has a limitation: both the declaration and definition has to be visible when compiling the calls to template methods. This is usually solved by declaring and defining the templates at the same time in the class' header file. However, this was not possible in our case because we are required to use forward declarations. Forward declarations declares an identifier (variable name) to be defined later (or somewhere else). Our template methods use this variable and just the declaration is not enough to compile. The problem was solved by creating another header file containing the definitions for the templates, this was included in the cpp-file calling the template methods. The original header file with template declarations was included from the cpp's corresponding header file. This way only the declarations are seen from the header file while both the declarations and definitions are seen from the cpp-file, as shown in Figure 5.

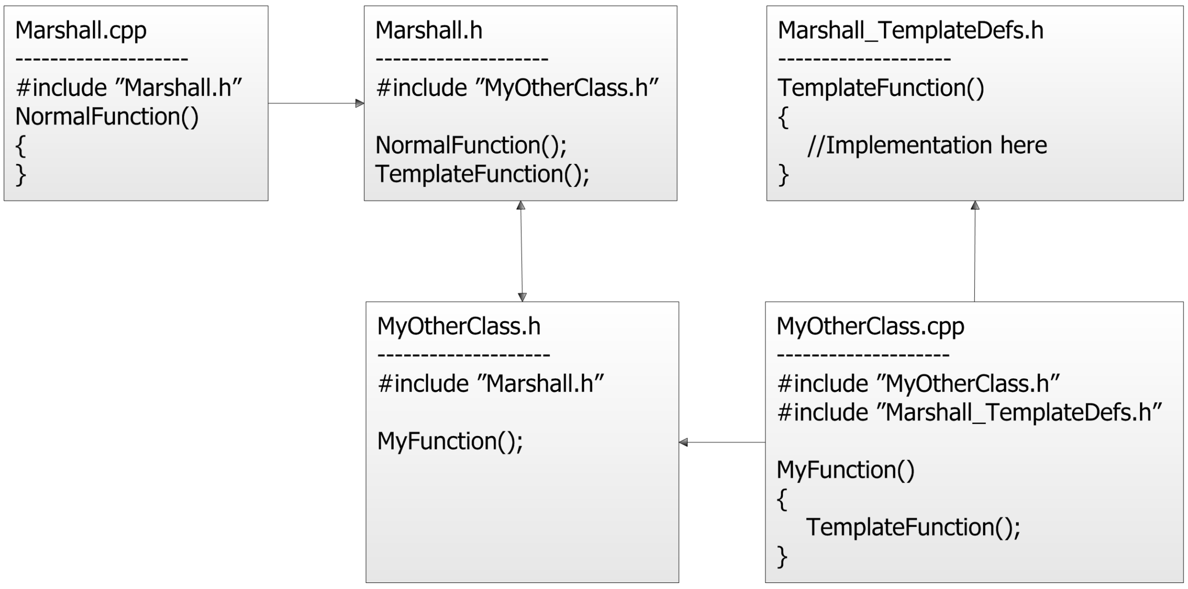

Figure 5: How template methods can be used in an environment where circular dependencies exist. Arrows show include paths. This way will never `TemplateFunction` be redefined.

6.4 Evaluation of Mono versions

The initial version chosen was the latest long-term supported. This was version 2.6.7 during this project. Problems were experienced quite early with this version on Windows. The `mono_jit_cleanup`-function always crashed when embedding Mono in WISE. The second version evaluated was Mono 2.10.1 this was the latest stable version at the time. No problems has come to our attention with this version on Windows.

One problem still exists on Linux, this is a problem with restarting the driver through the interface of the WISE runtime. Doing this causes a segmentation fault. This problem exists with both version 2.6.7 and 2.10.1 and does not seem to be related to the version used. We have tested this without the involvement of WISE and the problem persists. This might be a problem with the Linux architecture since restarting works on Windows.

The conclusion is that Mono version 2.10.1 is the one to use with WISE.

6.5 Garbage collection

Many modern languages come with a garbage collector (GC). The garbage collector takes care of the deallocation of objects for the programmer. With a garbage collector there should not be any problems with common memory management errors like double free, memory leaks or premature free. In the early days of garbage collection there were a significant trade-off between the convenience of a garbage collected program and the program's efficiency. Today the algorithms have improved and with the combination of more computing resources is this trade-off much less of an issue.

6.5.1 Description of the Mono-GC

The current version of Mono (2.6.7 and 2.10.x) uses a garbage collector based on "Boehm's Garbage Collector" by default. Boehm's GC is by default a conservative GC, meaning it interprets every variable as a potential pointer to an object to check if every object is referenced. In Mono's implementation the collector instead uses precise mode for most of the areas. With precise mode the GC only scans valid pointers for objects to be deallocated. Other modifications include support for thread-local storage (keep global data on a per thread basis) and concurrent garbage collection.[20]

The garbage collector scans the following data areas: [21]

- The heap (where other managed objects are allocated)
- thread stacks and registers
- static data area
- data structures allocated by the runtime

6.5.2 API

To prevent objects from removal by the garbage collector the API offers four different functions:[22]

Listing 5: GC Handles:

```
uint32_t mono_gchandle_new(MonoObject *obj,
        mono_bool pinned);
uint32_t mono_gchandle_new_weakref(MonoObject *obj,
        mono_bool track_resurrection);
MonoObject* mono_gchandle_get_target(uint32_t
        gchandle);
void mono_gchandle_free(uint32_t gchandle);
```

The function `mono_gchandle_new(...)` flags the object to the GC that it should not remove this object, the return value is used for identification towards the GC. The parameter ”pinned” defines if the GC is allowed to move the object to a different location on the memory-heap. The difference between a `weakref` and a normal `gchandle` is that the `weakref` does not prevent removal, it is just used to check if the object still exists or have been removed. `mono_gchandle_get_target(...)` uses the `gchandle` to receive the `MonoObject`. `mono_gchandle_free(...)` tells the collector that the object should be treated as an ordinary object and therefore possible for removal.

6.5.3 The wrapper driver implementation

A `MonoObject` created in C++ will not be collected by the GC as long as the current function has not yet returned.[23] With the marshalling implementation which delegates work down from advanced data types to simple data types, objects might be collected before they are sent to C#. The solution, as already pointed out, is to use `mono_gchandle_new(...)`. The bigger problem is to return the collection of the objects to the garbage collector after the objects are referenced by C# code. All `gchandles` have to be stored in some way to be able to be released later. This is solved with a custom class which takes care of the `gchandles` for us. The class is simple, it stores a stack with handled `gchandles` and inserts a new object with the method `add(...)` and releases them for collection with `free()`. An instance of this class is created before every marshalling and a reference to it is passed down the call hierarchy to every marshall method, so each method can add its objects to the stack. The free method is called after the call to C# with `mono_runtime_invoke(...)`.

6.6 Threads

WISE creates a dynamic number of threads for communication with different drivers. For the Mono-runtime to work in a threaded environment each thread needs to be registered to the runtime. This is done with a call to `mono_thread_attach()`. This function checks if the thread is already registered, so it is safe to call it multiple times with the same thread. We have tested to attach the same thread up to a thousand times without experiencing any problems.

6.7 Mono in multiple driver environments

WISE can run a multiple number of drivers in the same runtime at once. For Mono to be compatible with this setup the initialization requires some adjustments. Just calling `mono_jit_init(...)` from every driver does not work. This call may only be called once in each runtime. With the help of the `static` and thread-safe class `DriverInstances` we can find out if our driver is the first instance, if so, call `mono_jit_init(...)`. Otherwise call `mono_domain_create_appdomain(..)`. This call will create another domain inside the root domain.

6.8 Quality assurance

6.8.1 Test-driver

The test-driver developed is a special C#-driver where it is possible to run automated tests for every new revision of the wrapper-driver. This driver is multi-platform meaning it will run on both Linux and Windows and test the wrapper-driver as well as testing Saab's existing code for running C# drivers on Windows.

The current implementation of the test-driver tests the most common calls. With over 200 different calls in the sink-class the number of lines needed to achieve a complete test-driver is rather extensive.

6.8.2 WISE Test Tool

Within the WISE product suite there is a graphical tool for testing, called the WISE Test Tool. This tool connects through the debug-port to the WISE Runtime Environment. With the connection set up it is possible to monitor the databases, create objects, remove object and create events. This GUI is extended with scripting-capabilities in the Lua programming language, which makes it possible for more advanced testing scenarios and automated testing. We used Test Tool together with the test driver to verify that data sent from C# was successfully received by WISE.

This tool is not available on Linux but can connect through a socket to a Linux-machine.

6.8.3 Chat-driver

Another way to verify that the wrapper-driver behaves as it is designed, is to create a simple but proper implementation of a driver. To verify the core functionality of WISE a chat-system is a good choice. In the chat-system users are represented as objects and messages as events.

In our implementation the driver communicates over a TCP-connection between the user interface and WISE. This protocol is as simple as it can get, first there is an enum, called command, after that the name of the user and last extra data for example the actual message. Every field have a dynamic length and separated with a special character, the end of the data is recognized with an end-of-transmission character (ASCII code: 0x04).

6.9 Platform/Deployment

6.9.1 Deployment

One common problem with software development is that a program works fine in the development environment but as soon as it gets deployed, some things do not work as expected. This chapter will try to eliminate this problem.

The first thing to do with a new environment is to install WISE and all its dependencies. Ensure Saab.Mono.dll and MonoCSharp.dll are located in the WISE/bin/-folder. Install and configure Mono as described in the following table:

	Linux (Ubuntu)	**Windows (XP)**
Installer package	mono-runtime_2.10.1-5ubuntu3_i386.deb	mono-2.10.1-gtksharp-2.12.10-win32-2.exe
PATH (Environment variable)	*no adjustments needed*	C:\Program Files\Mono -2.10.1\bin\

6.9.2 64-bit systems

For the driver to work in a 64bits environment some adjustments have to be made. The WISE-platform uses long as an equivalent to 32bits integer, this is not true on *nix where they are 64bits. These variables need to be changed to `int32_t` instead. Another positive side effect would be that C# long (64bits) does not get mixed up with the native `long`.

With the change of pointer size on 64bits-systems some marshalling-methods might need adjustments.

Saab currently has no plans to compile WISE as native 64bits binaries.

7 Conclusion

The wrapper driver developed during this thesis enables drivers written in C# to communicate with WISE on both Linux and Windows based operating systems. Developing new drivers in C# is generally easier and faster than programming the driver in C++, this speeds up the process of building new drivers, and saves money for both Saab and its customers. Our wrapper driver have reached a high level of functionality and can be used in current condition. Saab have plans to release our wrapper in one of the upcoming releases of WISE.

The quality in the wrapper driver have been assured with the help of the C# drivers we have developed. Our C# drivers have helped us find problems in the implementation of the wrapper driver. The test driver was used to send data types from C# into WISE. With this we could quality asure the marshalling part of the wrapper. This is not a complete test of the wrapper and another driver was developed to test more functionality, this was the chat driver. The chat driver behaves more like a potential future driver and test other calls in the sink than the test driver. The combination of the two creates a good test to assure quality in the wrapper. The chat driver also functioned as a demonstration of our work to those unversed in our project.

7.1 Thesis reflection

The method for this project did help us with some structure especially during the beginning. The early research did pay off later on with the WISE-implementation. With the basic knowledge of how Mono-embedding worked the work could be focused upon the WISE integration. The second phase took a lot more time than our early estimates and the third phase did suffer to some extent. The reason for this was the higher demands from Saab in functionality in the wrapper driver than we originally thought.

The current status of the wrapper is full support for all methods and data types in WISE, but one issue remain, the restart-issue is unresolved. Mono seems to not cleanup all its resources during an unload of the Mono-runtime with the result that a new runtime will not start. One solution for this is tighter integration with WISE, with the result that the machine does not get teared down with a driver unload. Instead only unload the domain for that driver and the Mono-runtime remain alive until WISE receives an exit signal.

As a whole we are pleased with Mono and were surprised how feature-complete it was. The team of Novell-developers and driven individuals have made a good job. During this project the conditions for Mono development have changed, Attachmate, Novell's parent company, did layoff all of their Mono-developers. Attachmate claims that they are still committed to Mono-development, but the future is definitely uncertain for Mono. The major issue we found during this project was the lack of documentation. There is some documentation on the website but for the most part is it not explaining any functionality, just showing the function-name and the parameters. This resulted to a trial and error approach and a couple of e-mails to the developers mailing list.

8 Future development

This chapter describes further improvements that can be made with the wrapper driver and some guidelines on how to implement new features.

8.1 Potential performance improvements

Most of the needed mono-methods are stored in a couple of `std::map`'s for fast execution. This could be expanded in the future if a performance hit is noticed in current implementation.

Methods for calls to the constructor on different objects are not stored. Constructors are called in two ways, with the wrapper `mono_runtime_object_init(obj)` and with a ordinary lookup with `mono_class_get_method_from_name(...)` with the name .ctor. The wrapper does the same thing but does not handle parameters for the constructor.

Another place where caching is not implemented is some Add- and CopyTo-methods located in the MonoDriverCallbacks- and Marshall-class.

Both marshall methods for `List` and `Dictionary` currently use the `CopyTo`-method. By using this method the data is copied one extra time compared to reading it directly. This is unnecessary if the API gets extended or a method which uses the `Enumerator` (C# version of C++ iterator) can be used and loop directly over the elements.

8.2 New methods in the sink

To add new methods in the sink there are three different locations to make adjustments:

In the class CMonoDriver located in MonoDriver.cpp/h, the internal call have to be registered to the runtime.

In MonoDriverCallbacks.cpp/h the new method needs to be declared and call marshalling of the data that is to be passed in to the WISE-sink.

In CMonoWISEDriverSink.cs the updated interface, which should have triggered the whole process, have to be implemented with a secondary wrapper method that links the correct method in MonoDriverCallbacks.

To verify that everything works as it should, some new test-cases could be implemented in the test-driver.

8.3 Design improvements

There are some marshalling-methods that could be merged into a template method instead. This would lead to less duplication of code.

The marshalling of dictionaries from C# can be changed to make the code easier to maintain and understand. Currently the array is interpreted as an array of plain data, interpreting it as an array of key/value pairs as described in section 6.2.8 would make the code easier to understand.

In `MonoDriverCallbacks` is every method from the sink implemented for every possible data type. This could be changed so methods accept several data types. This would decrease the number of methods to maintain in `MonoDriverCallbacks`.

8.4 Upcoming mono versions

The Mono developers are currently working on a new garbage collector.[24] The big feature in the new one is the copying/moving operation. This operation uses a similar operation as a disk defragmentation-tool it tries to compact the objects so there is not any memory holes after removed objects. The negative thing about this operation is that pointers to moved objects will be broken. This might have a negative impact on the current marshalling-implementation.

References

[1] Saab in brief.
http://www.saabgroup.com/About-Saab/Company-profile/Saab-in-brief/. [2011-05-09].

[2] WISE Connectivity.
http://www.saabgroup.com/en/Land/Training_and_Simulation/Virtual-Constructive-Integration/WISE_Connectivity/. [2011-05-09].

[3] .NET Framework: Overview.
http://www.microsoft.com/net/overview.aspx. [2011-05-09].

[4] .NET Framework Conceptual Overview.
http://msdn.microsoft.com/en-us/library/zw4w595w.aspx. [2011-05-09].

[5] Code Conventions for the Java(TM) Programming Language.
http://www.oracle.com/technetwork/java/codeconvtoc-136057.html [2011-05-09].

[6] What is Mono - Mono.
http://mono-project.com/What_is_Mono. [2011-03-04].

[7] FAQ: Licensing - Mono
http://mono-project.com/FAQ:_Licensing. [2011-03-04].

[8] Software - Mono.
http://mono-project.com/Software. [2011-04-28].

[9] Compatibility - Mono.
http://go-mono.com/Compatibility. [2011-04-28].

[10] Mono - Class Status pages.
http://go-mono.com/status/. [2011-04-26].

[11] Justin. (March 2011). C# - How does Mono work - Stack Overflow.
http://stackoverflow.com/questions/216841/how-does-mono-work. [2011-03-04].

[12] MoMA - Mono.
http://www.mono-project.com/Moma. [2011-04-28].

[13] Mono Documentation.
http://www.go-mono.com/docs/. [2011-05-25].

[14] wchar_t Attribute (Windows).
http://msdn.microsoft.com/en-us/library/aa367308.aspx. [2011-05-25].

[15] Re: sizeof wchar_t.
http://gcc.gnu.org/ml/gcc/1998-08/msg00747.html. [2011-05-25].

[16] String Class (System).
http://msdn.microsoft.com/en-us/library/system.string.aspx. [2011-05-25].

[17] Robert Jordan.
http://go-mono.com/forums/#nabble-td1505983. [2011-04-27].

[18] Robert Jordan.
http://go-mono.com/forums/#nabble-td1538089. [2011-04-27].

[19] Embedding Mono.
http://www.mono-project.com/Embedding_Mono. [2011-04-26].

[20] .net - How is the current performance of the Mono virtual machine? - Stack overflow.
http://stackoverflow.com/questions/1150002/how-is-the-current-performance-of-the-mono-virtual-machine. [2011-05-25].

[21] Mono:Runtime - Mono.
http://www.mono-project.com/Mono:Runtime. [2011-05-12].

[22] Mono Documentation - GC Handles.
http://www.go-mono.com/docs/index.aspx?link=xhtml%3adeploy%2fmono-api-gchandle.html. [2011-05-11].

[23] Paolo Molaro.
http://go-mono.com/forums/#nabble-td1530913. [2011-05-24].

[24] Mono - Generational GC.
http://www.mono-project.com/Generational_GC. [2011-05-09].

Dictionary

Assembly	Container (.dll or .exe) for Managed code.
CIL	Common Intermediate Language, a common language Managed code is copiled into.
CLR	Common Language Runtime, the runtime which executes CIL code.
GC	Short for garbage collector.
GCC	GNU Compiler Collection.
GPL	GNU General Public License.
LGPL	GNU Lesser General Public License.
Marshalling	In this context: conversion between data types from different environments (C++and C#).
Managed code	Code that runs on the CLR VM (Mono runtime, C#-code).
Unix time	Number of seconds since midnight 1970-01-01.
Unmanaged code	Native code (C++).