CsharpSimpleModule – writing OMNeT++ Modules with C# and Mono

Andreas Lagemann  
BTU Cottbus  
Distributed Systems  
ae@informatik.tu-cottbus.de

Jörg Nolte  
BTU Cottbus  
Distributed Systems  
jon@informatik.tu-cottbus.de

ABSTRACT

Simulation normally serves one of two purposes. The first one is evaluation of certain algorithms. The second one is development and test of applications with infrastructural requirements which exceed those commonly available (e.g., distributed applications for wireless networks). In the latter case it is highly desirable that the code used for simulation can be easily adopted to real hardware with minor modifications. The .NET framework is – like Java – platform independent insofar as it only depends on a virtual machine implementation for each device it is meant to run on. Therefore for some application fields it is attractive to be able to write plain C# code which can then be run with a simulator like OMNeT++.  

This paper introduces CsharpSimpleModule, an extension to OMNeT++, which – like its companion JSimpleModule does for Java – allows you to write OMNeT++ simulation modules in C# and mix them freely with plain OMNeT++ modules, thus allowing you to build upon existing OMNeT++ frameworks (e.g. INET or MobilityFramework). Besides giving a short introduction to the usage of CsharpSimpleModule its general architecture will be illustrated and selected implementation issues will be discussed.

Categories and Subject Descriptors

I.6.5 [Computing Methodologies]: Simulation and Modelling—Model Development; D.1.5 [Software]: Programming Techniques—Object Oriented Programming

1. INTRODUCTION

Developing applications for wireless distributed systems is a tedious error prone process. Testing applications in real networks with hundreds of nodes is often prohibitively expensive and time consuming. Therefore the need for simulation arises. One common approach is to implement the algorithms in question in a way specific to the simulator used and then reimplement these algorithms for field tests on real hardware. This approach has two essential drawbacks. The first one is clearly the amount of time which is wasted in doing the same thing twice. The second drawback is that reimplementing always contains the risk of introducing errors or inconsistencies into the code. This is highly undesirable especially since debugging on non standard end devices can become a very tedious task.

OMNeT++ [11] is an object oriented discrete event simulator with a strong focus on network simulation. It has become quite popular in the last years and the user community is still growing. OMNeT++ can be roughly compared with NS2 [3] and OPNET [2].

This paper introduces CsharpSimpleModule, an extension to OMNeT++, which allows you to write simulation modules in C#. With .NET [10] being available on a lot of devices it is besides Java a good choice for developing distributed applications that shall be widely deployed to a great number of different devices. For applications like that CsharpSimpleModule can be used to greatly ease development by enabling OMNeT++ as a tool for testing and debugging. The authors unfold how you can enable an existing .NET application written in C# to run in OMNeT++ with only minor changes to the code. Additionally some design issues on how you can make your application code independent from the actual platform (i.e. OMNeT++ vs. real hardware) will be addressed. After that the overall software architecture of CsharpSimpleModule will be presented. Then some basic techniques which were necessary to create CsharpSimpleModule are briefly introduced before discussing some implementation issues that might help those who want to extend CsharpSimpleModule or accomplish something similar for another language than C# or Java. Finally related work will be presented and options for the future of CsharpSimpleModule will be discussed.

2. SOFTWARE ARCHITECTURE

The main objective of CsharpSimpleModule’s design is the integration of a C++ framework and runtime environment into the .NET platform, such that .NET applications (currently only C# is supported as programming language) have access to the framework on the one hand,
and the runtime has means of controlling certain .NET objects. Thus there are two main tasks to be solved. The first task is to make the framework’s API accessible from within C# code. For that purpose wrapper code was generated using SWIG (see section 4.1.1). The wrapper code is two tiered. All framework functionality (including the public methods of all classes which shall be accessible from within C# programs) is wrapped with corresponding C-style functions which are compiled into a library, that can then be used by special C# functions that access the corresponding native functions. The second layer consists of wrapper classes which mainly map method calls to a corresponding function, which in turn call the C-style functions of the first layer. The mechanism allowing for this call is denoted P/INVOKED (platform invoke). It uses metadata to locate the exported unmanaged functions and marshal its parameters at runtime (see [5], [6] and [9]). The application code only needs to access the wrapper classes without bothering with P/INVOKED. Figure 1 illustrates the layered approach with the P/INVOKED mechanism as a tie between them.

The second task, i.e. providing means of control of some C# objects for the OMNeT++ runtime, is solved by introducing a special cSimpleModule called CsharpSimpleModule. This module is responsible to start the mono runtime and load the appropriate assemblies which were defined via the module’s parameters. Using the reflective mechanisms of the mono API it creates an object of the corresponding application module which must be derived from the C# counterpart to CsharpSimpleModule which is a C# class with the same name. This class is not a proper wrapper class for its C++ counterpart, because that would lead to infinite recursion, but there are certain methods on the C++ side, which can be accessed from the C# side and the other way round. Note that for every CsharpSimpleModule created there exists exactly one instance of the C++ and one instance of the C# class. These two objects must be linked very closely to avoid inconsistent behaviour. How this close linkage is technically achieved will be described in detail in section 4.2.

3. USING CSHARPSIMPLEMODULE

CsharpSimpleModule allows you to write simulation modules for OMNeT++ with C#. It is based on the mono runtime environment (see section 4.1.2) which is an open implementation of the .NET framework specified by Microsoft and uses the Simplified Wrapper Interface Generator (SWIG) (see section 4.1.1) to generate C# wrapper classes that present the OMNeT++ API to the .NET developer.

Before describing the usage of CsharpSimpleModule in detail, we will first recall the elements of a OMNeT++ model. Every model consists of modules. There are two types of modules. The simple modules are basic building blocks. They are the active components of the model, programmed in C++ and represented by the cSimpleModule class, which serves as a base for the implementing class. Simple modules can be combined into compound modules. Every module (simple as well as compound) can define gates. Modules inside a compound module can be connected via those gates. Connections between gates are simplex. Therefore there are input and output gates. Every input gate of the compound module as well as every output gate of a submodule can serve as an input. Similarly an output gate of the compound module and an input gate of a simple module can serve as an output. Note that every gate can only be used as endpoint of one connection. Direct connections between submodules contained in different compound modules are not allowed. Events are represented as messages in OMNeT++. A separate event class is not provided. A message is an instance of cMessage or any derived type. They are sent from one module to another; therefore the module where the event will occur is called the message destination. Parameters add more flexibility to the usage of modules and facilitate the reusability of modules. Each module can define an arbitrary number of parameters.

Basically the package provides an OMNeT++ module called CsharpSimpleModule (which is a cSimpleModule). A simulation is run by instances of CsharpSimpleModule which work as a proxy module for the C# modules. It has parameters allowing the definition of the class name, the namespace of the module and the assembly where it resides. To set up a run with C# code you simply need to insert CsharpSimpleModules into a topology description where appropriate.

To illustrate the usage of CsharpSimpleModule consider the following simple example application. The model consists of two modules, named “tic” and “toc”. The modules will send one message back and forth between them, where “tic” is responsible for creating the message and sending it the first time. Since both modules do essentially the same thing, it is sufficient to implement one single CsharpSimpleModule. We will call it Txc. Listing 1 shows the C# code implementing the Txc module. In the initialize() method the module with the name “tic” creates a HelloMessage which will be sent back and forth between the two modules. The handleMessage() method receives the HelloMessage (see below for an explanation why the cast has to be done this way), reads and prints the counter of the message and sends it back. The finish() method merely prints a message to the environment stating that it has been called.

HelloMessage is implemented in C#. It is not derived from cMessage directly, because cMessage on the C# side is a wrapper class. Wrapper objects for cMessage are only valid until the message is sent to another module. When the

\footnote{Assembly is the .NET term for a partially compiled code library consisting of byte code. An assembly can either represent an executable or a sole library (which is what we need for our purpose).}
Listing 1: A small example CsharpSimpleModule

send() method is called on the C# side, the corresponding C++ message object is retrieved from the wrapper object and passed to the send() method on the C++ side. When the message arrives at the other module, it is re-wrapped with a newly created instance of the C# wrapper class. One design principle has been to allow CsharpSimpleModule to be used with an unaltered version of OMNeT++. Therefore the C++ side object contains no reference to its C# counterpart. In order to allow for messages fully implemented in C# a class named CsharpSimpleModule can be used. Virtually the whole simulation API can be used. However all C++ side classes are not pose any greater problem, because the use of gate vectors allows for dynamic gate creation.

There are some specialties to be considered when implementing simple modules in C#. Virtually the whole simulation API can be used. However all C# side classes are just wrapper classes containing no state information other than a pointer to the underlying C++ peer object. When a wrapper object is newly created from within a C# program the corresponding C++-object is created via a call of the appropriate constructor on the C++ side. Every invocation of a method from the wrapper object is redirected to a call to the C++ object. Also if you send a cMessage via send(), the wrapper function retrieves the pointer to the cMessage stored in the wrapper object and passes it to the method call of the underlying C++ object. After sending a cMessage the corresponding wrapper object should be considered invalid and must not be used anymore, since the underlying cMessage (on the C++ side) has then been deleted by the OMNeT++ simulation kernel or other modules. Similar, if methods return a reference to an object, the return value of the C++ method call is wrapped in the appropriate C# wrapper object which is returned as a result of the method call.

As a direct consequence you cannot compare two objects directly with the equals() method of the object class, because the identity of the wrapper object does not correspond to the wrapped object’s identity, i.e. the same object can (and will most probably) be wrapped by different wrapper objects. To solve this, each wrapper object provides a method named sameAs() which compares the pointers to the C++ object for equality rather than comparing the wrapper objects themselves.

Another implication is that a returned object reference

Listing 2: A small example NED file

3This is promoted to be a new feature in OMNeT++ 4.0
will always be wrapped by an object of the declared return type rather than an object of the actual type of the original object. This is an issue if the return type of the given method is more general than the actual type. To access methods of the actual type the object has to be re-wrapped. This can be done by the cast() method, which every wrapper object provides. The method behaves like a dynamic_cast(); it returns null if the conversion could not be performed4. Again the originally returned object should be considered invalid after casting and not be touched again.

Last but not least due to the highly dynamic nature of simulations the standard memory management approach of SWIG (see 4.1.1) is not used. The C# wrapper classes never delete the underlying C++ object. They have to be explicitly deleted by using the Dispose() method. Messages no longer needed should be disposed of, for example. After calling Dispose() the wrapper object naturally should be considered invalid. This is underlined by the Dispose() method setting the reference pointer to null.

3.1 Bringing the Real World to OMNeT++

The following example serves the purpose of illustrating how appropriate application design can lead to code which can be easily adapted to the use with OMNeT++ as well as concrete devices. Thus the OMNeT++ simulator (in conjunction with CsharpSimpleModule if applications are written with C#) can serve as a valuable development tool bringing great ease to debugging and testing distributed applications5.

In distributed applications the participating nodes need some means of communication usually described as “the network”. Access to this network is customarily provided via a well defined interface which offers some abstractions for data sent between nodes (often called messages) and some addressing scheme to distinguish nodes. This interface is introduced to decouple the application logic from the technical details of communication. So by regarding OMNeT++ as a certain kind of network one can easily hide all OMNeT++ dependent code behind that interface. Applications which are built upon that interface can then be run on any platform which provides a network implementation.

To illustrate this point further, consider the (simplified) application design shown in figure 2. The application (represented by the class Application) uses the interface Network to communicate with applications on other nodes being totally indifferent to the way the interface functionality is implemented. The class UDPPNetForDotNetCF implements the Network interface for the .NET compact framework using UDP/IP to transmit messages over the network. To simulate the same application with OMNeT++, you only have to provide an OMNeT++ module which implements the Network interface and replace the UDPPNetForDotNetCF with that module (which besides implementing the interface also has to inherit from CsharpSimpleModule as described above). The resulting design should look similar to that in figure 3. This seems to be simple enough; however there are some pitfalls which should be noticed.

The most significant problem lies in the semantic of network service primitives (like send or receive). In many communication frameworks (e.g. the UDP implementation of the .NET framework) the default behaviour of send and receive operations is synchronous, meaning that they block the calling entity until the operation was carried out successfully or a non maskable error has occurred. With OMNeT++ such a behaviour can only be achieved by using the activity() mechanism which leads to modules being implemented as coroutines [8] each with its own stack and thread of control. Besides being strongly discouraged by the author of OMNeT++, since it leads to poorly scalable simulation models, this feature is not supported by CsharpSimpleModule. So if you want to write applications which shall run with OMNeT++ as well as with real devices you are strongly encouraged to apply an event driven programming style. Otherwise it will become very difficult to avoid the use of synchronous primitives, which will make it hard if not impossible to port your code to OMNeT++.

Another problem lies in providing timeout facilities. Most real distributed applications need some means of measuring how long a certain operation takes and to specify a maximum time for that operation in order to detect communication failures. In real systems this is usually accomplished using some clock device, which generates interrupts in a certain time interval. An interrupt handler can inform registered entities about timer expiration. Another not very precise method, which additionally is somewhat prodigal of resources, is to setup an extra thread which sleeps some amount of time and informs registered entities each time it awakens. Neither threads (see discussion above) nor interrupts as such are available for CsharpSimpleModule. But OMNeT++ does provide a mechanism which is quite similar to a timer interrupt but more flexible and more easy to use: the self message. A self message is a plain Message which is sent via the scheduleSelfMessage method of cModule. With the help of self messages it is possible to add functionality similar to a timer to your module. Note that with OMNeT++ it is possible to define a separate module at which other modules can register to receive notifications of a timeout event. Such a module would probably be very similar to the NotificationBoard from the INET framework [12]. With CsharpSimpleModule such a solution is currently not possible, because the support for direct method calls to CsharpSimpleModules is not implemented yet. It will probably be supported in a future release, however, since

4 actually it is implemented using dynamic_cast()
5 although the focus of this paper lies on distributed applications, the principles shown can be easily adapted to other kinds of applications suitable for OMNeT++ as well
the authors regard it as a crucial element for developing well designed component based simulation models.

3.2 Integration with non C# modules

Since there are numerous frameworks out there for OMNeT++ you certainly want to use some of them for projects with C#SimpleModule as well. There are several ways to use your C#SimpleModule models together with existing OMNeT++ modules implemented in C++. In the most simple case the model you want to use does not specify special messages but simply operates with cMessages. In that case you simply connect your C#SimpleModule’s output gate with that module’s input gate and call the send method of your module passing a respective cMessage. However, many modules expect a certain message type to be passed to their input gates. To create an instance of such a message without the need to wrap the corresponding C++ classes you can use a generic interface for the manipulation of C++ classes provided by OMNeT++. If for instance you want to use the message defined in listing 3, you can retrieve an instance of it and access it like shown in listing 4.

4. IMPLEMENTATION ISSUES

This section introduces certain aspects of the implementation which are regarded crucial to successfully meet CSharpSimpleModules requirements. After presenting two basic techniques used to implement CSharpSimpleModule, potential problems concerning memory management are presented. After that the solution for providing a kind of cross-language-polymorphism is explained.

4.1 Basic Techniques

The implementation of CSharpSimpleModule is mainly based on two basic techniques. The wrapper generator SWIG greatly eases the implementation of C# wrapper code for C++ libraries. The mono runtime provides means to control the execution of C# programs from within an arbitrary C++ program.

4.1.1 Generating Wrapper Code with SWIG

SWIG is an interface compiler which was basically designed to integrate the power of (existing) C/C++ libraries with the flexibility offered by scripting languages. From 1996 it steadily evolved to offer support for a huge amount of scripting languages and has also supported Java and C# for some years. It works by parsing the declarations from a C/C++ header file and generating appropriate wrappers for the given target language. The original C++ code, which is accessed through the wrapper code, need not be touched, which is one of the most important aspects of SWIG generated code.

Basically you need to create an interface file. In the most simple case this file consists of import statements for some C++ header files containing the interfaces that shall be wrapped. This simple method will suffice for small applications with rather trivial usage patterns. However SWIG allows for more complex mechanisms to solve more demanding tasks. For a deeper introduction to SWIG see [4].

Given a simple class like shown in listing 5, SWIG will generate a proxy class in the target language (C# in our case). This class is solely a wrapper class of low level functions, which in turn call the methods of the C++ class through some means supported by the target language (P/INVOKE (see section 2) in the case of C#). The resulting proxy class will roughly look like depicted in listing 6.
The concept of proxy objects keeping references to native objects raises many memory management issues, which are addressed by SWIG by adding some API for ownership control to the generated proxy classes. Each proxy object contains a reference to the corresponding native object as well as a flag indicating whether it is currently the owner of the object, i.e. if it is responsible for deleting it. Thus the native object is only deleted if at the time when the proxy object is destroyed, the ownership flag is set. We will discuss memory management issues related to OMNeT++ wrapping in more detail in section 4.3. Code for memory management and ownership tracking has been omitted in listing 6 for clarity and simplicity. The Dispose() method (also omitted) is called by the destructor. It is responsible for calling the native destructor and releasing .NET side references.

Most importantly you can configure almost everything of the wrapper code generated to your specific needs. While for simple tasks the procedure described above is fully sufficient and easily applied, it is possible to adjust SWIG's generated code in such a way that you are able to accomplish more complex tasks, while leveraging the automatic code generation done by SWIG, which saves a lot of time when implementing interfaces for native libraries.

### Embedding the Mono Runtime with C++

The mono project is an open development initiative sponsored by Novell to develop an open source UNIX version of the Microsoft .NET development platform. It implements various technologies which were developed by Microsoft and have then been submitted to the ECMA for standardization. Currently it provides three main components:

- A common language interface (CLI) [1] virtual machine that contains a class loader, just in time compiler and a garbage collecting runtime.
- An API implementation of the standard .NET API as well as most of Microsoft's API and a mono specific API.

6Ecma International is an industry association founded in 1961 and dedicated to the standardization of Information and Communication Technology (ICT) and Consumer Electronics (CE).

---

```
Listing 6: C# proxy generated by SWIG

```public class Foo : IDisposable {
    public Foo() {
        monoObject = (new MonoClass("example.dll"));
        monoObject.set_with_partial_name("example.dll");
    }
    public virtual void Dispose() {
        monoObject.dispose();
    }
    public int bar(int i) {
        int ret = examplePINVOKE.Foo.bar(swigCPtr, i);
        return ret;
    }
}
```

Listing 7: A simple C# object hierarchy

```java
public class Base {
    public virtual void myOwnMethod();
}
public class Derived : Base {
    public new void myOwnMethod();
}
```
tained (6-10). Creating an object involves two steps. First the space for the given object has to be allocated, and then the object has to be initialized by calling the standard constructor\(^7\) (11-12). To get a handle for a certain method, one has to iterate through all methods of the given object and compare them to the name of the desired method. If the name is ambiguous, you can also query the number of arguments. If that is still not sufficient for dis-ambiguation, the argument types have to be considered. The example shows only the case where the method name is sufficient to distinguish it from other methods (17-41). Note that the first method retrieved will always call the method defined in class Base and the second one always the method defined in Derived when invoked on an object of type Derived. If you have virtual methods and want to achieve the usual semantics for calling a virtual method (i.e the most specialized version of the method in a given object hierarchy is called) you have to use the function mono_object_get_virtual_method() (see line 39). The actual invocation of the method is performed via a single function call (43-47).

It expects the method handle, the object on which the method call shall take place as well as a list of parameters and a pointer to a MonoObject pointer which will hold an exception object if an exception has been triggered by the call.

The mono embedding API provides a lot of more functions, which let you control almost every aspect regarding the execution of .NET programs. Listing all of them here would lead too far and is out of the focus of this paper. The reader should have gotten a general picture of how embedding the mono runtime can be achieved and should now be able to comprehend the more detailed description of some implementation aspects given below. For more information see [7].

### 4.2 Cross-Language Polymorphism

A logical object, which consists of two objects each written in a different programming language, is denoted by the term bilingual object in the context of this paper. CsharpSimpleModule and CsharpMessage (see section 3) are two examples for such an object. The object implemented in one language will be designated the peer of the other object. The term cross-language polymorphism denotes a property of bilingual objects. This property holds if the object can be extended in one programming language and the correct type – according to the well-known rules of polymorphism – can be retrieved from a reference to the peer object.

To illustrate the implementation of bilingual objects which provide the property of cross-language polymorphism we picked CsharpMessage as an example; the implementation of CsharpSimpleModule is analogous in this respect. To fulfill the requirements of cross-language polymorphism, the C++ side object must hold a reference to its C# peer object. This reference must be retrievable from the C# side, when for instance a HelloMessage is retrieved via handleMessage(). As described in section 3, when the message is passed to the C# side method, a wrapper object of type cMessage is created. At this point the original C# peer has to be retrieved. For this purpose CsharpMessage provides a method named swigCsharpPeerOf(), which is implemented on the C++ side. It accepts a pointer to a cPolymorphic object. Using dynamic_cast() it determines if the object is of the correct type. If so it calls the method swigCsharpPeer() which returns a pointer to the peer object. Otherwise null is returned. For convenience CsharpMessage provides a method named cast(). This method accepts a reference to a cPolymorphic wrapper object. That object is passed to swigCsharpPeerOf(). The formal return type of swigCsharpPeerOf() is MonoObject* on the C++ side which resolves to object on the C# side. This object is transferred into a CsharpMessage via a static cast.

In order to retrieve the peer object as described above, it must somehow be stored in the C++ side peer first. Providing a method on the C++ side, which stores the reference passed as a MonoObject* and accessing it via a wrapper method, is the first solution that comes to mind. However, it does not work. This approach uses P/INVOKE (see section 2) and therefore involves parameter marshalling. Currently there is no support for marshalling C# objects as MonoObject pointers. To implement a routine, which performs that task would rely on implementation details of a particular mono version. Since that details are most likely subject to change with later mono versions, depending on them should be avoided.

In the .NET framework a delegate denotes a type-safe function pointer. When bound to a managed method, it can be resolved to a C-style function pointer. The automatic P/INVOKE marshalling does exactly that. The function pointer can be used to call that method from within unmanaged code. If the return type of the method is object it is resolved to a MonoObject* when called via the function pointer.

Knowing this, the given problem can be solved as follows: A method getReference() is introduced to the C# side CsharpMessage. This method is then bound to a delegate. At object creation time the delegate is passed to the C++ object by calling the method setCsharpPeerCallback(). After storing the function pointer it is directly called to retrieve and store the reference to the peer object. Additionally a handle of the peer object for the garbage collector is added to avoid the collection of the peer object when there are no more references on the managed side.

### 4.3 Memory Management

The tight coupling of the two corresponding objects of type CsharpMessage and CsharpSimpleModule has some implications on how object duplication must be implemented. Another issue regarding memory management is that of automatically freeing memory allocated in unmanaged memory. Although such mechanisms can add some convenience for the user when implemented correctly, the dynamics present in discrete simulations strongly prohibit such techniques. Tracking ownership (which essentially includes the right to delete the objects owned) becomes cumbersome and error prone, at least if there are two objects to be tracked simultaneously, one being subject to garbage collection. Therefore the deletion of no longer needed objects of type CsharpMessage is left to the user. The class CsharpMessage implements the IDisposable interface which provides the Dispose() method. A call to this method checks if it has a valid (non null) reference to a C++ object and if so deletes it. To indicate that the object is not valid any longer it sets the pointer to the unmanaged object to null.

---

\(^7\) One can call non standard constructors as well. It has been omitted for brevity.
When copying CsharpMessages, a new C++ object of type CsharpMessage has to be created (a copy is not needed, because all data is kept in the C# object) and the C# side object must be cloned. This can be achieved with `mono_object_clone()`, which basically does a `memcpy()` of the given object. After that the object has to be registered with the garbage collector. The registration yields a handle. That handle represents a reference to the object. As long as the handle exists, the object will not be fetched by the garbage collector. Note that until now the pointer to the unmanaged peer object points to the same C++ object as the original object, since we did a simple `memcpy()`. Therefore that pointer has to be updated to point to the newly created object. This procedure is implemented in the copy constructor of CsharpMessage naturally. This way the implementation of the dup() function for the C++ side only has to create a new object using the copy constructor.

5. RELATED AND FUTURE WORK

The implementation of CsharpSimpleModule is greatly inspired by JSimpleModule [13]. The SWIG interface files of JSimpleModule could be reused with only some minor adaptations, mostly at those sections, where Java code was explicitly stated in the interface file. Since the Java and the .NET platform are very similar, especially in terms of their interface to native libraries, most concepts could be adopted in a straightforward fashion. Although already being in development when JSimpleModule was released, the design of CsharpSimpleModule was greatly influenced by it. The author of JSimpleModule being also the author of OMNeT++, its design provided a deeper insight in the inner workings of OMNeT++, and therefore the implementation of CsharpSimpleModule could benefit from that.

As stated in section 3.1, CsharpSimpleModule lacks support for direct method calls to other modules. To be able to call the methods of other modules directly, you have to register that call with the simulation kernel. Until now the corresponding part of the API of OMNeT++ has not been wrapped. The mechanism cannot be transferred to the C# world directly because the helper class uses a constructor/destructor mechanism which makes use of a C++ feature. Namely that objects allocated in automatic memory (i.e. “on the stack”) are automatically destroyed when the scope they were declared in is closed. C# does not provide automatic memory, so some other technique has to be applied to realize this in C#. As stated above, support for direct method calls will be added in the near future, because they are regarded as an important feature.

Both frameworks are missing support for the activity()-feature, which allows implementing modules as cooperative threads (based on a coroutine concept [8]), which allows synchronous sending and receiving of messages. The support for that feature is missing for a good reason: apart from its great performance implications and the less structured code it produces, its implementation will be very intricate. It would involve adaptation of the coroutine implementation of OMNeT++. So either the mono runtime would have to be embedded into OMNeT++ coroutines, where it would execute and instrument the C# code in the activity()-method of CsharpSimpleModule, or a coroutine implementation for .NET would have to be developed, which could work together with the OMNeT++ framework seamlessly.

6. CONCLUSION

This paper presented CsharpSimpleModule. It has been shown how to use it in existing projects and how to adapt real world applications to CsharpSimpleModule in order to simulate them with OMNeT++. Basic techniques used for the implementation of CsharpSimpleModule were presented. The overall software design was explained. Implementation details have been discussed. On the one hand this can serve as kind of tutorial introduction to CsharpSimpleModule for developers, on the other hand it provides enough conceptual and general discussion to serve as a guideline to implementing something similar for languages other than Java or C#.

7. ACKNOWLEDGEMENTS

The author would like to thank Andras Varga for supporting the deployment and development of CsharpSimpleMod-ule and for providing JSimpleModule [13] in the first place. Special thanks go out to Grzegorz Sobanski and Lukasz Piatkowski being the first external users of CsharpSimpleMod-ule and providing some essential bug fixes and new features like the opportunity of using a debugger and profiler tools with CsharpSimpleModule. A very special thank you to Sören Hockner and Tobias Senmer who had to use Csharp-SimpleModule in very early development stages and provided valuable feedback to improve it.

8. REFERENCES