The Pi-ADL.NET project: An Inclusive Approach to ADL Compiler Design

ZAWAR QAYYUM, FLAVIO OQUENDO
VALORIA Laboratory
University of South Brittany
Tohannic Campus, Vannes 56017
FRANCE
zawar.qayyum@univ-ubs.fr, flavio.oquendo@univ-ubs.fr, http://www-valoria.univ-ubs.fr/ARCHLOG/

Abstract: - This paper describes results and observations pertaining to the development of a compiler utility for an Architecture Description Language π-ADL, for the .NET platform. Architecture Description Languages or ADLs, are special purpose high level languages especially construed to define software architectures. π-ADL, a recent addition to this class of languages, is formally based on the π-Calculus, a process oriented formal method. The compiler for π-ADL, named π-ADL.NET, is designed with the view of bringing the architecture driven software design approach to the .NET platform.

The process oriented nature and a robust set of parallelism constructs of π-ADL make the π-ADL.NET project a novel application of compiler techniques in the context of the .NET platform, with many valuable lessons learnt. This paper presents the π-ADL.NET effort from a compiler design perspective, and describes the inclusive approach driving the design that facilitates the representation of strong behavioral semantics in architecture descriptions. The subjects of parallel process modeling, communication and constructed data types are covered. The paper also documents the motivation, vision and future possibilities for this line of work. A detailed comparison with related work is also presented.

Key-Words: - π-ADL, Compiler design, Aspect oriented programming, CIL, Software architecture, Architecture description danguage

1. Introduction

A software architecture is a high-level view of a software system, focusing on commonalities amongst different software components, fundamental design choices, data schematics and other high-level features. The motive for a software architecture is the same as that for building architectures: to make plans before implementation, minimize uncertainties and risks, and deliver a standardized, high-quality product. To fully and clearly represent software architectures, various Architecture Description Languages (ADLs) have been proposed. A detailed survey and comparison of ADLs is reported in [1].

π-ADL is a relatively recent ADL, described in [2]. Its distinguishing features are that it is formally derived from π-Calculus and it is possible to define architectural styles using it. The language syntax is designed incrementally with higher level functionality built upon basic syntactic layers. The advantage to this approach is that it opens the possibility of extensions to the language, allowing for domain specific syntactic enhancements. The approach also allows greater leeway for syntactic experimentation and evolving the language specification.

The motivation for the π-ADL.NET project which is presented in this paper is the need to validate and compile an ADL on a mainstream platform. This has certain advantages. Firstly, it opens the possibility of integrating the ADL code with the code written in a detail oriented language, such has C# or Visual Basic.NET, since they are both compiling to the same target platform. That way, the software architect's investment in the design effort is employed directly in the resultant software solution. Secondly, the compiled ADL code can access and utilize the large number of reusable software libraries already developed for the platform. Third, it is an interesting approach to heterogeneous software development, whereby different portions of a software are programmed in languages better suited to their development. For example in implementing a large software project, an ADL can be used for the high-level architectural specification, and a 3G language be used for the detail oriented leg work.

These factors are increasingly relevant in the present-day context when software developers are faced with more and more complex functional, compatibility and standards conformance requirements for the systems they build. As the overall effort requirement for software development increases, there is a greater need for role separation amongst members of a software development team, while at the same time reducing redundancy of effort to a minimum. For this it is not enough to just clearly define and enact specialized roles in a software development team, but tool and technology support plays a pivotal role. The most pervasive example of this is the clear separation of the presentation component from the logic component of the software system being designed, when using certain web development technologies such as JSP and ASP.NET, and in the very recent case of using the Extensible Application Markup Language for desktop applications based on the Windows Presentation Foundation [21].

While the clear demarcation between presentation and logic has evolved from the nebulous state of first generation web technologies, the concept has yet to be fully implemented in tools and development methodologies for separating software architecture from detailed logic, even though we do see tool support from industry for software design. The need to separate architecture from detailed logic is well argued in literature [22] [23] [24]. What we need now is an integrative development environment that allows architects and programmers to conduct and seamlessly combine their respective areas of specialized work.

The focus of this paper is to describe and evaluate a novel application founded on established compiler techniques: compiling π-ADL to the Common Intermediate Language (CIL), the assembly language for the .NET platform. The implementation itself supports in detail the use of flexible constructed data types and other language features that allow the software architecture developed using this platform more and more relevant to the end software product, thus addressing our above stated objectives of separation of concerns with seamless interoperation. Section 2 briefly describes the syntax of important π-ADL constructs in order to make the algorithm descriptions in subsequent sections understandable. Section 3 introduces the CIL. In Section 4 implementation details pertaining to the parallel processing constructs of π-ADL are presented. Section 5 presents implementation details for connection syntax and semantics. Section 6 gives an overview of the constructed data types supported by this compiler. Section 7 concludes this paper with comparison to related research.

2. π-ADL

π-ADL is a language designed for defining software architectures and is formally founded on the higher-order typed π-calculus described in [4]. In a π-ADL program, the top level constructs are behaviours and abstractions. Each behaviour definition results in a separate execution entry point, meaning that the program will have as many top level concurrent threads of execution as the number of behaviours it defines. Abstractions are reusable behaviour templates and their functionality can be invoked from behaviours and abstractions. An abstraction is capable of receiving a single argument when invoked.

The body of a behaviour or an abstraction can contain variable and *connection* declarations. Connections provide functionality analogous to channels in π-calculus: code in different parts of behaviours or abstractions can communicate synchronously via connections, and connections can also connect behaviours with abstractions or abstractions with abstractions. Connections are typed, and can send and receive any of the existing variable types, as well as connections themselves. Sending a value via a connection is called an output-prefix, and receiving via a connection is called an input prefix. Listing 1 shows a simple program in which a behaviour invokes an abstraction (known as the pseudo-application of an abstraction), and associates its connection *x* with the connection *y* of the abstraction through the *rename* clause. This enables communication between the behaviour and the abstraction during the course of their respective executions.

The *compose* keyword seen in Listing 2.1 serves the purpose of creating two or more parallel threads of execution within a program and corresponds to the *concurrency* construct in π-calculus. The generalized syntax for a compose block is:

```
composeBlock := "compose {" block [" and "
block]+ "}"
```

where each block inside the compose block results in a separate thread of execution. Note that if the two statements inside the compose block were coded to execute in a single thread, a deadlock would have occurred.

Another important π-ADL construct is the *choose* block. It has the following generalized syntax:

```
chooseBlock := "choose {" block [" or "
block]+ "}"
```

Only one of the sub-blocks inside a choose block is executed when execution passes into the choose block. For example, in Listing 1, only one of the two choose sub-blocks can execute. Since a value is available via y and not via x, the second sub-block will execute and the first sub-block will be terminated. When more than one sub-blocks are eligible for commencing execution at the same time, the selection criteria for the block to be executed is not defined in the language.

To provide the equivalent of the π-calculus replicate construct, π-ADL supports the *replicate* keyword, with the following syntax:

```
replicateBlock := "replicate {" block "}"
```

Semantically, this entails that the contents of the replicate block are infinitely replicated in parallel threads of execution. As we will see in Section 5, the implementation of replicate has been modeled with the limits of real world computers kept in mind.

3. CIL

The Microsoft Common Intermediate Language or CIL is a low-level stack-oriented language, designed to be able to express every feature of the Microsoft .NET common language runtime.

It is presented in detail in [7]. Given that the .NET platform was designed to be able to support the syntactic requirements of a host of different high-level languages, the CIL packs a lot of features with syntax for namespaces, classes, methods, templates, events, exception handling, and string manipulation – in addition to what is normally found in assembly languages. This is helped by the fact it is not tied to any particular native platform and it's limitations, but is instead just-in-time compiled to the host platform prior to its first execution.

For our purpose of representing π-ADL in terms of CIL, the elements described in Section 2 are not directly supported by CIL. The approach presented in this paper is therefore improvised, with the results accomplished by creating special classes and supporting methods to completely represent the semantic ramifications of the said π-ADL constructs in CIL.

4. Compiling π-ADL Parallel Processing Constructs

As mentioned in Section 2, each behaviour definition in a π-ADL program results in a separate thread of execution at startup. Other cases where parallel processing occurs are the compose, choose and replicate blocks. Here we treat the runtime implementation of each of these π-ADL constructs in turn.

4.1. Behaviour implementation

A π-ADL behaviour is compiled as a separate class in CIL. Variables and Connections declared in a π-ADL behaviour become class variables in CIL. While the π-ADL assumes its variables and connections to be initialized with declaration, it has to be done explicitly in CIL inside the default constructor for the behaviour class. The CIL code corresponding to the functionality defined in the π-ADL code for the behaviour is generated in the ep$ method, which is effectively the entry point of execution for each behaviour class. Note that the CIL representation of an abstraction also has a similar ep$ method, and is called against a pseudo-application in π-ADL. The entry point of a compiled π-ADL.NET program is the main method of an internal class Controller$. After all the behaviours are identified, the main method of the Controller$ class is generated.

The function of the Controller$ main method is to simply instantiate each behaviour class, and for each instance create a corresponding thread object. Each thread is then started with the ep$ method of the behaviour as its startup method. The main method of the Controller$ class initiates each behaviour thread in the order in which the behaviour is declared in the source program. However the actual execution order is determined by the .NET runtime.

Program a: Behaviours, Abstractions and Connections	**Program b: Compose and choose**

```
behaviour {
  x : Connection[Integer];
  compose  {
    via myAbs send 42 where {x renames y};
  and
    via x send 101;
  }
}
value myAbs is abstraction (argi : Integer) {
  y : Connection[Integer];
  i : Integer;
  via y receive i;
  argi = i * argi;
}
```
	```
	behaviour {
	  x : Connection[Integer];
	  y : Connection[Boolean];
	  a : Integer;  b : Boolean;
	  compose  {
	    via y send true;
	  and
	    choose {
	      via x receive a;
	    or
	      via y receive b;
	    } //end choose
	  } //end compose
	} //end behaviour
	```

Table 1. Two π-ADL programs demonstrating various language concepts.

4.2. Implementing compose

Semantically, π-ADL requires the sub-blocks of a compose block to execute in parallel without any precedence requirements. In order to accomplish that in CIL, the logic of the sub blocks is output in separate methods for each sub-block. These methods belong to the class representing the behaviour or abstraction and are called from the ep$ method of that behaviour or abstraction. The methods are named using the notation "method<methodIndex>", where methodIndex is a global integral value incremented each time a compose or choose sub-block is encountered.

To implement parallelism for all the sub-blocks inside compose, a separate thread is created for each sub-block and executes the method generated for that sub-block. If there is a compose block nested inside another compose block, more methods are created at the class level; there is no nesting of methods comparable to the representation in π-ADL, since such a nesting is not supported by CIL. Instead, the methods are named using the above mentioned convention and simply called from the method representing the parent sub-block, using the same threaded approach as the one used for the top level compose block. The π-ADL.NET compiler does not impose any limit on the number of nesting levels for any of the compose, choose or replicate constructs, and any of these constructs can be nested within each other in an arbitrary order.

4.3. Implementing choose

The logic of each of the different sub-blocks inside the choose block is output inside a separate method, very much like it is done for compose blocks. Each of these methods is invoked through a separate thread, as in compose. However the threads for the choose sub-blocks are in competition and the successful thread needs to be able to terminate the rest before it can execute. A basic approach would be that the first thread to execute its first statement would terminate the others. However there is a possibility that while one thread is busy terminating the other threads, another thread may resume execution and start the termination routine on its own. Therefore in addition to the task of terminating all competing threads, the chosen thread should also be able to signal the other threads of its nomination.

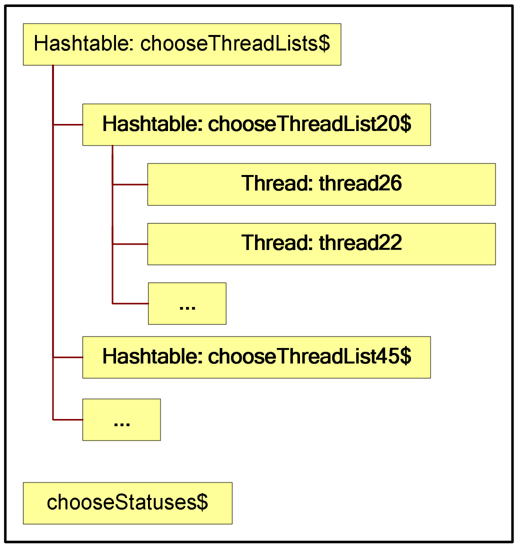
Hashtable: chooseThreadLists$
Hashtable: chooseThreadList20$
Thread: thread26
Thread: thread22
...
Hashtable: chooseThreadList45$
...
chooseStatuses$

Fig 1. Runtime metadata maintained by a π-ADL.NET executable for managing Choose constructs variables in PiParser

Fig 1 shows the data structures maintained by the π-ADL.NET runtime for thread management and signaling. Each choose block has a corresponding hash table named chooseThreadList<x>$, where the value of x is the integral index assigned to the choose keyword by the scanner. A global hash table chooseThreadLists$ maintains reference to all the chooseThreadList<x>$ objects.

Each of these hashtables in turn maintains a reference to all threads corresponding to the sub-blocks of its associated choose block. In order to support signaling, the hashtable chooseStatuses$ is used. It contains reference to each choose block by its token index number, and against this key it maintains a Boolean value indicating whether one of its threads has commenced execution. By default this value is false for each choose block. Upon starting execution, any choose sub-block thread does the following:

a) Acquire a lock on chooseStatuses$ by using the Monitor.Enter method.
b) Check Boolean value for the associated choose block in chooseStatuses$. If true then branch to e).
c) Update the value for the associated choose block in chooseStatuses$ to true.
d) Call a special internal method cleanupChooseNonTermins$ to terminate all the threads for the given choose block. This method takes a string argument for the name of the chosen thread so as to exlude it from the termination sequence.
e) Release the lock on chooseStatuses$ by using the Monitor.Exit method.
f) If the value examined in b) is true then the method returns and the thread terminates. Otherwise it continues executing the logic encoded in its associated sub-block.

4.3.1. Special case for abstractions

This algorithm ensures that one and only one thread is executed for any choose block. However there is one caveat in the metadata structure which becomes evident when multiple instances of the same abstraction are executing in parallel. Since the chooseThreadList<x>$ variables attempt to maintain exclusivity of executing choose blocks through their unique indexed position in the π-ADL code, that property is not applicable when the same code is executing in different threads at the same time.

To resolve this issue, a custom class Int64Obj has been developed to encapsulate a 64-bit integral value. Each class representing an abstraction contains a static variable of this type named choose$exe$number. In addition, a local variable choose$exe$local of type Int64Obj is instantiated for each choose block found at some level in an abstraction. Every time the abstraction is executed, choose$exe$number is incremented and this local variable is assigned the resulting value in a thread safe manner so that each abstraction instance holds a unique value for it’s choose$exe$local.

Subsequently, the entry made to the chooseStatuses$ hash table incorporates the value of the thread unique choose$exe$local variable in its key value, adding the desired level of differentiation in naming to allow multiple abstractions to be identified separately. Furthermore, in the case of abstractions the naming format of the hash table chooseThreadList<x>$ is modified to chooseThreadList<x>$-<choose$exe$local> to achieve the desired level of differentiation. This allows cleanupChooseNonTermins$ to terminate only those threads that belong to the one particular abstraction instance being executed.

For the above modification to accommodate abstractions, we observe that modifying the design thus has the desired effect of exclusivity without changing the above mentioned algorithm, or drastically modifying the structure in Fig 1.

4.4. Implementing replicate

Semantically, the grammar of the π-ADL replicate construct can be recursively defined as:

```
  replicate{<set of statements>} ::=
compose{<set of statements> and replicate{<set
of statements>}}
```

This implies that an infinite number of threads will be created to execute the <set of statements> in parallel. While this syntax establishes a close correspondance between the replication concepts of π-ADL and π-Calculus, it creates obvious problems for the compiler implementation, where an arbitrary number of parallel processes will quickly overrun the processor and memory ressources of the system executing a π-ADL program. To resolve this problem, the following solutions were considered:

a) Have a preset maximum number of parallel threads that can execute at any given time. As soon as one thread terminates, another one is launched.
b) Extend the replicate syntax to allow the provision of an integral parameter. The value of this parameter will specify to the system the maximum number of parallel threads to be launched at any given time. Compared to a), this approach gives the flexibility of tuning the concurrency level according to the problem at hand.
c) Have only one thread active at any given time. This would give the computational equivalent of an infinite iterative loop.

The π-ADL.NET compiler implements choice c). The rationale is that although the approach of a) allows a larger problem set to be executed in conformance with the π-Calculus formalism, no preset value for the maximum number of parallel threads can ensure the π-calculus conformant execution of all possible programs. Choice b) atleast gives problem specific assurances for the correct simulation of the replicate formalism, but at the cost of deviating from π-ADL syntax. The advantage of c) is the simplicity of the implementation while conforming to the π-ADL syntax. Using c), a functionality similiar to b) can be implemented through other means, such as illustrated in Table 2.

```
behaviour {
  compose {
    via myAbs send 42;
  and
    via myAbs send 42;
  and
    via myAbs send 42;
  }
}
value myAbs is abstraction (argi : Integer)
{
  i : Integer;
  replicate {
    via in receive i;
    i = i * argi;
   via out send i;
  }
}
```

Table 2 Simulating controlled concurrent replication

5. Compiling π-ADL Connections

In developing the π-ADL.NET compiler, the implementation of connection related functionality went through multiple iterations before reaching its present form. The reasons for this are the following issues:

- The send and receive operations have to be atomic. The possibility of modifying the variable being transfered in a parallel thread should be avoided.
- The Connections must provision for having multiple send operations queued up. Receive operations must execute without the loss of any data sent in such a case.
- Connections must also cater to the synchronization needs of choose statements in-case the first statement of a choose sub-block is a receive statement.
- Connections are mobile. They can be sent and received through other connections, and can be passed as arguments to abstractions. Earlier compiler implementations created a new CIL connection class against each π-ADL connection declaration, and then instantiated it. However, mobility entails that both the sender and recipient share the same class definition. To retain both definitional consistency and type flexibility, a single generic connection class is defined.

Table 3 shows the Connection class implementation in C#. The CIL generated and used by the π-ADL.NET executable is isomorphic with this code. Looking at the class declaration in line 1 we see that the class is declared as a .NET generic class, similar to a C++ template class. The variable declarations on lines 2, 3 and 4 assist in the correct functionality of the send and receive methods. There are 2 AutoResetEvent instances declared in line 2.

An AutoResetEvent is like a logical gate. Threads block on an AutoResetEvent object when they call its WaitOne() method, and if the AutoResetEvent object is in the non-signaled state. When the Set() method is called on the AutoResetEvent object, the waiting thread is unblocked and resumes execution.

The AutoResetEvent object allows only one thread to be unblocked for each Set() call, and reverts to the non-signaled state thereafter [8].

The Interlocked class used both in the send and receive methods provides atomic operations for variables that are shared by multiple threads [9]. The two methods Increment and Decrement of the Interlocked class used in the send and receive methods are used to atomically increment and decrement long values. The Read method reads the value of a long variable atomically.

As can be seen in the implementation in Listing 3, AutoResetEvents, the Interlocked class and the Monitor class are employed to ensure correct synchronous interaction between the send and receive methods.

The synchronous communication facility provided by the send and receive methods can result in a myriad different contexts of interaction. This can occasionally result in unexpected forms of interaction, resulting in the .NET framework level exceptions. A simple and practically proven solution is to re-try the send or receive operation, as can be seen in the exception handling code in lines 29, 54 and 78.

We also note that there are 2 versions of the receive method. The one declared at line 60 is specifically written for choose blocks to ensure thread safe execution of the choose algorithm explained in Section 5 as well as of the receive operation.

```
public class ConnectionTemplate<T> :
AbstractConnection {
AutoResetEvent e, receiveE;
long sendWaiting, receiveWaiting;
Queue<T> valQueue;
private string toStringVal;
public ConnectionTemplate() {
e = new AutoResetEvent(false);
receiveE = new AutoResetEvent(false);
sendWaiting = 0;
receiveWaiting = 0;
valQueue = new Queue<T>();
}
public void send(T a)
{
bool noException = true;
valQueue.Enqueue(a);
do {
if(Interlocked.Read(ref receiveWaiting)==0)
{
Interlocked.Increment(ref sendWaiting);
e.WaitOne();
Interlocked.Decrement(ref sendWaiting);
}
else {
try {
e.Set();
receiveE.WaitOne();
}
catch (System.InvalidOperationException)
{
noException = false;
}
}
} while (noException == false);
}

public void receive(ref T a)
{
bool noException = true;
do {
if (Interlocked.Read(ref sendWaiting) == 0) {
Interlocked.Increment(ref receiveWaiting);
e.WaitOne();
a = valQueue.Dequeue();
Interlocked.Decrement(ref receiveWaiting);
receiveE.Set();
}
else {
try {
a = valQueue.Dequeue();
e.Set();
noException = true;
}
catch (System.InvalidOperationException) {
noException = false;
}
}
} while (noException == false);
}

public void receive(ref T a, ref Hashtable
chooseStatuses) {
bool noException = true;
do {
if (Interlocked.Read(ref sendWaiting) == 0) {
Interlocked.Increment(ref receiveWaiting);
e.WaitOne();
Monitor.Enter(chooseStatuses);
a = valQueue.Dequeue();
Interlocked.Decrement(ref receiveWaiting);
receiveE.Set();
}
else {
Monitor.Enter(chooseStatuses);
try {
a = valQueue.Dequeue();
e.Set();
noException = true;
}
catch (System.InvalidOperationException) {
noException = false;
}
}
} while (noException == false);
}

public bool IsEquivalent(object x) { /*…*/}
public void setToString(string val) { /*…*/}
public override string ToString(){ /*…*/}
public void deepClone(out object obj) { /*…*/}
}
```

Table 3. Connection implementation for π-ADL.NET (C# code)

6. Constructed Data Types

For a high-level architecture description language, π-ADL represents a large and varied set of data types, in addition to the primitive types Integer, String, Boolean and Float. These constructed data types are: Tuple, View, Any, Union, Sequence, Set, Bag, Quote, Variant, and Location.

After evaluating the extent of effort required in implementing all of these constructed data types, the first 5 data types in the above list where chosen as a representative set of data types to be implemented. A short description of each of these data types is as follows:

- **Tuple**: The values of a tuple type `tuple[`T_1`, …, `T_n`]`, for n≥2, are n-tuples tuple(v_1, …, v_n) where each v_i is of type T_i. The individual values within the tuple can be projected into other variables using the `project` syntax e.g. `project t as a, b;`.
- **View**: Views are labeled forms of tuples. The values of a view type `view[label`$_1$` : `T_1`, …, label`$_n$` : `T_n`]`, for n≥2, are views view(label$_1$: v_1, …, label$_n$: v_n) where each v_i is of type T_i. In addition to projecting like a tuple, a view also supports short-hand projection, much like accessing a class variable in Java or C# e.g. in order to reference the member `a` of view `v`, we use the syntax `v::a;`.
- **Union**: A union type `union[`T_1`,…,`T_n`]`, for *n*≥2, is the disjoint union of the types T_1, …,T_n, with values union(T_i::v) where v is of one of the types T_i. The current type of a union can be determined by processing it in a select-case block of code, where each case is represented by one of the member data types of the union.
- **Any**: An any type is like a union type with no constraint on the type of value it can hold.
- **Sequence**: A sequence is a collection in which the elements are ordered. An element may be part of a

sequence more than once. The values of a sequence type sequence[T] are sequences sequence(v_1, …, v_n) where each v_i is of type T.

Despite this conservative short listing, the implementation of specialized constructed data types presented an inherent challenge: each of the 5 implemented constructed data types is capable of containing one or more of the other data types, as well as connections. This composition requirement entails that the containing data type must be sensitive to the peculiar properties of the contained data types where necessary. Roughly speaking, the effort to implement a certain number of data types had a polynomial relationship to the number of data types since it had to accommodate all the existing data types, and the existing data types also needed to be updated to support the new data type.

Furthermore, no assumption was made regarding the recursive limits and possibilities engendered by the constructed type specifications. A sequence of views can be defined to contain views, each of which can contain named sequences of views and so on. Several marginal cases resulted from this commitment to provide a functional system of recursive type definition, and had to be handled appropriately as they arose.

In general the following roles and behaviors are expected from each data type:

- Declared in a behaviour or an abstraction.
- L-value or an r-value (both as variable and as constant) in an assignment.
- Sent or received via a connection.
- Sent or received via the output or input user interface stream respectively.
- Argument to an abstraction and be used in a pseudo-application thereof.

In addition, the following behaviors are specific to one or more data types:

- **View and tuple**: projection of member values into a set of variables.
- **View**: shorthand projection.
- **Union and any**: type identification using a select-case statement. The union or any type consequently takes up the properties of its current type within the scope of the correct type case. Also, the ability to assign other data type values directly e.g. `a = x;` where `a` is of type any and `x` is an integer. Constraint checks must be in place when performing such an assignment to a union.
- **Sequence**: member access via an integer index.

While the existing variety of constructed data types give the π-ADL.NET implementation a rich syntactic basis for modeling the structure and behavior of a software architecture, the implementation experience shows that a more viable approach would have been to provide a meta-type system equivalent to the classes and objects in object oriented languages. Using such a framework would enable the user to easily recreate all the properties embodied by the specialized data types and more, with relatively little complexity of implementation at the compiler level. This is easy to conclude from the widespread success of the object paradigm today.

The formal specification work required to develop a meta-type system compatible with the π-calculus foundations of π-ADL is well outside the domain of the π-ADL.NET project, and can form a line of separate research work in the future.

The π-ADL.NET constructed data types are implemented as distinct classes in an external C# library linked with executable π-ADL code at runtime. This helps keep the executable size small, with a minor performance penalty. Also this approach helped to simplify the implementation of data type functionality, which would otherwise have been a difficult and time consuming exercise in CIL programming.

The current implementation of constructed types does not support type definition i.e. the definition of a particular tuple, view, union or sequence global signature that can be used in behaviours and abstractions. Without this feature, the programmer has to define and redefine a particular constructed type each time he intends to use it, causing programming overhead. The implementation of type definition functionality has also been identified as future work, although by providing a meta-type system implementation we may alleviate that need.

7. Conclusion

The π-ADL.NET compiler currently compiles a large subset of π-ADL. This subset is Turing complete and provides coverage for all the operations derived from π-calculus. In subsequent development phases, it will completely implement the π-ADL specification and will add implementation specific extensions to the language syntax for providing access to external .NET class libraries. For the latter work, the language extensions will be formally derived in order to conform to π-ADL's formal design approach.

Apart from the core compiler design work, we have done some prototyping work to use the compiler as a foundation for a visual programming and architecture design environment using the GME tool, and in exploring approaches to the use of 3D modeling and animation in defining a viable visual syntax. This line of work is discussed in [10] along with some initial results.

Under current development is a DirectX based 3D programming and modeling interface being developed as a plugin to Microsoft Visual Studio 2005 and later. The objective of this initiative is provide a 3D visual interface to the π-ADL.NET compiler and experiment with concepts such as program simulation via 3D animation, visual debugging, simultaneous multi-aspect views of the architecture during design and simulated execution, and the simultaneous orchestration of these concepts. This ambitious project will form the core of the next generation π-ADL toolset as a complement to the π-ADL.NET compiler.

About 1500 lines of code have been written to test the various functions of the π-ADL.NET compiler. In addition, a functional web-service architecture has been developed in collaboration with the computer science department of the University of Rey Juan Carlos, constituting a novel application of π-ADL compared to pre-π-ADL.NET modeling work. The π-ADL.NET compiler is currently under consideration at the VALORIA laboratory for modeling multi-agent systems, as well as the High Level Architecture (HLA) [11]. The parallel and high-level nature of π-ADL makes it suitable for development work in all of these application areas. There is also some current work at our laboratory in visually modeling π-ADL and generating code using the DSL tools for Visual Studio .NET. The DSL tools are reported in [5].

The long term objective of the π-ADL.NET project is to test the modeling of more and application domains, and consequently improve the toolset towards an industry strength solution for software architecture modeling.

7.1. Related work

While there is no existing research work in compiling a π-calculus based language or an ADL to the .NET platform, there are some compilers for formally founded languages for .NET. One notable example is the .NET compiler for the F# language [12], which is based on the ML language [13], a formally founded functional programming language. Another ML based language for the .NET platform is SML.NET [18] based on Standard ML '97. L Sharp.NET [19] is an implementation of the Lisp functional programming language for the .NET platform. DotLisp [20] is a lisp like interpreted .NET language.

Turning from formally founded functional language compilers for .NET to non-.NET compilers for process oriented languages, we find that the BoPi language reported in [14] implements a distributed computing semantic based on asynchronous π-calculus i.e. the send and receive prefixes are asynchronous. Compared to this, the π-ADL.NET implementation is based on synchronous π-calculus. A compiler implementation also exists for the occam-pi language [15], which embraces elements of both CSP and π-calculus.

The PiLib [25] domain specific language is an interesting alternative solution for providing a process coordination abstraction in that it is implemented as a software library hosted by the Scala language. This delivers the functionality to program process oriented constructs without having to implement a separate compiler and runtime – that of Scala is used. It is notably relevant to the π-ADL.NET work that this implementation posed the researchers a set of challenges in adapting the semantic style of Scala for the purpose of supporting PiLib. However the contrast in semantics and the type system between Scala and PiLib isn't nearly as great as that between π-ADL and CIL. Also, the application domain of PiLib is primarily as a simple π-calculus interpreter designed for teaching purposes. The π-ADL.NET work, of course, has different goals. During the implementation of π-ADL.NET we realized the limitations imposed on the design, had a high level language such as C# been used to host π-ADL.NET instead of the assembly level CIL. The most notable limitation is the legal character choices in variable naming, which is the same for π-ADL and C#, and supports no easy resolution of conflicts between the naming of internal and user defined variables.

Amongst modeling tools for ADLs, the Honeywell® MetaH has a workspace environment for which the source module components [16] are of relevance to this work. They allow the generation of some aspects of the executable from architectural specifications written in MetaH.

A distantly comparable work is also available in the form of the analysis and constraint checking tools for AcmeStudio [17], the eclipse based modeling environment for the Acme ADL.

Thus we find .NET compilers for functional languages, compilers for π-calculus based languages and parser tools for ADLs. While the work reported in this paper may be related to all of these three categories of research, it forms a distinct domain of work in itself by virtue of being the only π-calculus based ADL compiler for the .NET platform.

Acknowledgements

This work is supported by the Higher Education Commission of Pakistan under the Overseas Program 2005-776582.

References

[1] N. Medvidovic, R. Taylor, A Classification and Comparison Framework for Software Architecture Description Languages, *IEEE Transactions on Software Engineering*, Vol.26, No.1, 2000, pp.70-93.

[2] F. Oquendo, π-ADL: An Architecture Description Language based on the Higher Order Typed π-Calculus for Specifying Dynamic and Mobile Software Architectures, *ACM Software Engineering Notes*, Vol.29, No. 3, 2004, pp 1-14.

[3] F. Oquendo, *Tutorial on ArchWare ADL – Version 2 (π-ADL Tutorial)*, ArchWare European RTD Project IST-2001-32360, 2005.

[4] R. Milner, *The Polyadic π-Calculus: A Tutorial*, Logic and Algebra of Specification, Springer-Verlag, 1993.

[5] J. Greenfield, K. Short, Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools, *18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 2003, pp 16-27.

[6] Z. Qayyum, *Visual Modeling and Concrete Implementation of the π-Architecture Description Language*, Masters 2 Final Report, Université de Bretagne-sud, 2006.

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

[8] *AutoResetEvent Class (System.Threading)*, Microsoft online MSDN Library. See http://msdn2.microsoft.com/en-us/library/system.threading.autoresetevent.aspx

[9] *Interlocked Class*, Microsoft online MSDN Library. See http://msdn2.microsoft.com/en-us/library/system.threading.interlocked(VS.71).aspx

[10] Z. Qayyum et. al., π-ADL Visual Notation and its Application to Formally Modeling the High Level Architecture, *19th International Conference on Software and Systems Engineering and their Applications*, 2006.

[11] IEEE Standard 1516-2000, *IEEE standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Framework and Rules*.

[12] D. Syme, J. Margetson. *The F# website, 2007*. See http://research.microsoft.com/fsharp/.

[13] R. Milner et al, *The Definition of Standard ML (Revised)*, MIT Press, 1997.

[14] C. Samuele et al, BoPi – a distributed machine for experimenting Web Services technologies, *Fifth International Conference on Application of Concurrency to System Design*, 2005, pp 202-211.

[15] F.R.M. Barnes, P.H. Welch, Communicating Mobile Processes, *Lecture Notes in Computer Science*, Vol. 3525, 2005, pp 175-210.

[16] *MetaH Language and Tools*, See http://www.htc.honeywell.com/metah/tools.html.

[17] *The AcmeStudio Home Page*, See http://www.cs.cmu.edu/~acme/AcmeStudio/index.html.

[18] N. Benton et al., Adventures in interoperability: the SML.NET experience, *Proceedings of the 6th ACM SIGPLAN International conference on Principles and Practice of Declarative Programming*, Vol.6, 2004, pp 215-226.

[19] R. Blackwell, *The L Sharp.NET website, 2007*, See http://www.lsharp.org/.

[20] R. Hickey, *The DotLisp website, 2007*, See http://dotlisp.sourceforge.net/dotlisp.htm.

[21] K. Januszewski and J. Rodriguez, The New Iteration: How XAML Transforms the Collaboration between Designers and Developers in Windows Presentation Foundation, *Microsoft Corporation* white paper, 2007.

[22] D. E. Perry and A. L. Wolf, Foundations for the Study of Software Architecture, *Software Engineering Notes, ACM SIGSOFT*, Vol.17, No.4, 1992, pp 40-52.

[23] T. Maginnis, Engineers don't Build, *IEEE Software*, Vol. 17, No.1, 2000, pp 34-39.

[24] V. Ambriola and G. Tortora, Advances in Software Engineering and Knowledge Engineering, *World Scientific Publications*, 1993.

[25] V. Cremet and M. Odersky, PiLib: A Hosted Language for Pi-Calculus Style Concurrency, *EPFL Dagstuhl Proceedings: Domain-Specific Program Generation*, 2003.