Abstract: This paper describes some of the results and algorithms from 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 also documents the motivation, vision and future possibilities for this line of work.

Index Terms: π-ADL, Compiler Design, CIL; Software Architecture; Architecture Description Language

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 as 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.

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. 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 concludes this paper.

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 improvisional, 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.
composeEtoEexecuteEinEparallelEwithoutEanyEprecedenceEtheEthreadsEforEtheEchooseEsub)blocksEareEinEcompetitionEandEinvokedEthroughEaEseperateEthread,EasEinEcompose.EHoweverElikeEitEisEdoneEforEcomposeEblocks.EEachEofEtheseEmethodsEisEchooseEblockEisEoutputEinsideEaEseperateEmethod,EveryEmuchEsub)block.ETheseEmethodsEbelongEtoEtheEclassErepresentingEtheEsubEblocksEisEoutputEinEseperateEmethodsEforEeachErequirements.EInEorderEtoEaccomplishEthatEinECIL,EtheElogicEofEarbitraryEorder.EanyEofEtheseEconstructsEcanEbeEnestedEwithinEeachEotherEinEanEforEanyEofEtheEcompose,EchooseEorEreplicateEconstructs,EandEforEtheEtopElevelEcomposeEblock.ETheEEπ)ADL.NETEcompilerEandEsimplyEcalledEfromEtheEmethodErepresentingEtheEparentEmethodsEareEnamedEusingEtheEaboveEmentionedEconventionEsuchEaEnestationEisEnotEsupportedEbyECIL.EInstead,EtheEmethodsEcomparableEtoEtheErepresentationEinEπ)ADL,EsinceEmethodsEareEcreatedEatEtheEclassElevel;EthereEisEnoEnestationEofEcomposeEblockEnestedEinsideEanotherEcomposeEblock,EmoreEexecutesEtheEmethodEgeneratedEforEthatEsub)block.EIfEthereEisEaEcompose,EaAseperateEthreadEisEcreatedEforEeachEsub)blockEandEmethodEofEthatEbehaviourEorEabstraction.ETheEmethodsEareEtheEbehaviourEorEabstractionEandEareEcalledEfromEtheEep$E

doesEnotEimposeEanyElimitEonEtheEnumberEofEnestationElevelsEdeclaredEinEtheEsourceEprogram.EHoweverEtheEactualEbehaviourEthreadEinEtheEorderEinEwhichEtheEbehaviourEisEbusyEterminatingEtheEotherEthreads,EanotherEthreadEmayEbeEableEtoEsignalEtheEresumeEexecutionEandEstartEtheEterminationEroutineEonEitsEown.EThereforeEinEadditionEtoEtheEtaskEofEterminatingEallEcompetingEthreads,EthechosenEthreadEshouldEalsoEbeEableEtoEsignalEtheEotherEthreadsEofEitsEnomination.E

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 nonestation of methods comparable to the representation in π-ADL, since such a nestation 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 nestation 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.

<table>
<thead>
<tr>
<th>Program a: Behaviours, Abstractions and Connections</th>
<th>Program b: Compose and choose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>behaviour {</code></td>
<td><code>behaviour {</code></td>
</tr>
<tr>
<td><code>  x : Connection[Integer];</code></td>
<td><code>  x : Connection[Integer];</code></td>
</tr>
<tr>
<td><code>  compose {</code></td>
<td><code>  y : Connection[Boolean];</code></td>
</tr>
<tr>
<td><code>    y via myAbs send 42 where {x renames y};</code></td>
<td><code>  a : Integer;  b : Boolean;</code></td>
</tr>
<tr>
<td><code>    and</code></td>
<td><code>  compose {</code></td>
</tr>
<tr>
<td><code>    via x send 101;</code></td>
<td><code>    via y send true;</code></td>
</tr>
<tr>
<td><code>  }</code></td>
<td><code>    and</code></td>
</tr>
<tr>
<td><code>}</code></td>
<td><code>    choose {</code></td>
</tr>
<tr>
<td><code>value myAbs is abstraction (argi : Integer) {</code></td>
<td><code>    via x receive a;</code></td>
</tr>
<tr>
<td><code>  y : Connection[Integer];</code></td>
<td><code>  or</code></td>
</tr>
<tr>
<td><code>  i : Integer;</code></td>
<td><code>    via y receive b;</code></td>
</tr>
<tr>
<td><code>  via y receive i;</code></td>
<td><code>} //end choose</code></td>
</tr>
<tr>
<td><code>  argi = i * argi;</code></td>
<td><code>} //end behaviour</code></td>
</tr>
</tbody>
</table>

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

Fig 1 shows the datastructures maintained by the π-ADL.NET runtime for thread management and signalling. Each choose block has a corresponding hashtable named chooseThreadList<x>$, where the value of x is the integral index assigned to the choose keyword by the scanner. A global hashtable chooseThreadLists$ 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 signalling, 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 cleanupChooseNon Termins$ 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 exclude 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.

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<×>$ 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. This issue has been identified and will be addressed in future work by making the metadata structure dynamic at its second level i.e. instantiate an exclusive chooseThreadList<×>$ variable at runtime for each choose block in an abstraction. We observe that modifying the datastructure thus will have the desired effect of exclusivity without changing the above mentioned algorithm.

4.4. Implementing replicate

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

\[
\text{replicate}\{\text{set of statements}\} ::= \text{compose}\{\text{set of statements} \land \text{replicate}\{\text{set of statements}\}}
\]

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 correspondence 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 resources of the system executing a π-ADL program. To resolve this problem, the following solutions were considered:

g) 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.

h) 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.

i) 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) at least 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 similar to b) can be implemented through other means, such as illustrated in Table 2.

\begin{verbatim}
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;
    }
}
\end{verbatim}

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 transferred 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.

We also note that there are 2 versions of the receive method. The one declared at line 40 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.

6. 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, some prototyping work has also been done to use the compiler as a foundation for a visual programming and architecture design environment, 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.

About 1000 lines of code have been written to test the various functions of the π-ADL.NET compiler. It 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 both 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].

6.1. 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.

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 modelling 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

I’d like to acknowledge the guidance of my PhD advisor Dr. Flavio Oquendo, who developed π-ADL, and in particular guided me with regard to the implementation of the replicate construct. Special thanks to Adam Wright for his comments and in-depth review of the paper. This work is supported by the Higher Education Commission of Pakistan under the Program Overseas 2005-776582.

References