ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

Dipartimento di Informatica - Scienza e Ingegneria DISI

TESI DI LAUREA

in

Fondamenti di Informatica T-2

tuProlog.NET: ANALISI DI SOSTENIBILITA' DEL PROCESSO DI SVILUPPO

CANDIDATO:
Matteo Castigliò

RELATORE:
Chiar.mo Prof. Enrico Denti

Anno Accademico 2017/2018

Sessione II

Indice

Introduzione

tuProlog è un sistema Prolog open-source che può essere utilizzato come ambiente di sviluppo o come motore interno per infrastrutture e applicazioni. Attualmente tuProlog è utilizzato come componente di diversi progetti, sia universitari che commerciali.

TuProlog è disponibile per le piattaforme Java e .NET. tuProlog in Java è sviluppato manualmente, mentre la versione .NET è attualmente ottenuta attraverso un processo di traduzione automatico con l'ausilio di IKVM.NET. Questo strumento consente la generazione di codice Common Intermediate Language a partire da bytecode e rende quindi possibile ottenere entrambe le varianti di tuProlog a partire dello stesso codice sorgente.

L'annuncio della interruzione dello sviluppo di IKVM, che rimane unicamente compatibile con Java 8, rende opportuno individuare una alternativa che possa supportare le più recenti versioni del linguaggio Java.

Nella prima parte di questa tesi saranno analizzate le conseguenze su tuProlog della fine dello sviluppo di IKVM, attraverso una catalogazione dei casi d'uso strettamente legati a tuProlog.NET. In seguito a questa prima analisi verranno poi proposti e confrontati eventuali approcci di soluzione, dei quali sarà inoltre valuta la fattibilità.

Nella parte finale, per ogni approccio di soluzione, verrà valutato se esistano alternative concrete disponibili.

1 tuProlog come applicazione multipiattaforma

tuProlog[1] è un framework leggero e open-source per applicazioni e infrastrutture basate sul linguaggio Prolog. A differenza di altri ambienti di programmazione per Prolog, progettati per fornire un sistema Prolog performante ma monolitico, tuProlog è progettato per essere minimale, dinamicamente configurabile e nativamente integrato con gli ambienti Java e .NET. La scelta progettuale prevede lo sviluppo di tuProlog a partire da un core che fornisce le funzionalità essenziali per un motore Prolog, mentre le funzionalità secondarie possono essere aggiunte attraverso l'inclusione di librerie. Le librerie forniscono predicati, funtori e operatori aggiuntivi e possono essere caricate e scaricate dal motore Prolog sia dinamicamente che staticamente. tuProlog è attualmente rilasciato in quattro versioni differenti: JavaSE, plugin Eclipse, Android e Microsoft .NET, tutte distribuite con licenza LGPL.

1.1 tuProlog Java

Java è la principale piattaforma di riferimento per tuProlog. La versione in Java è infatti la più completa e la più utilizzata. tuProlog è disponibile sia come vero e proprio ambiente di sviluppo per applicazioni Prolog in forma di file eseguile `2p.jar`, sia come libreria `tuProlog.jar` per chi vuole integrare le funzionalità di tuProlog all'interno di una applicazione o di un'infrastruttura senza però necessitare dell'ambiente completo.

tuProlog è quindi un vero e proprio interprete Prolog che può essere utilizzato in applicazioni esterne come supporto all'introduzione di parti di codice a paradigma logico, ma è anche framework Prolog che definisce vari predicati aggiuntivi. I predicati sono definiti all'interno delle librerie. In questo modo ogni volta che viene eseguito tuProlog è possibile configura, attraverso il caricamento e lo scaricamento delle librerie, quali sono i predicati utilizzabili.

Sviluppando quindi una nuova libreria è possibile definire nuovi predicati ed estendere l'espressività del linguaggio Prolog.

A partire dalla versione 3.0 data l'introduzione del supporto alle lambda expression la piattaforma di riferimento è diventata Java 8. Tuttavia escludendo le parti di codice relative a quest'ultime tuProlog è totalmente compilabile per Java 6.

tuProlog è fortemente integrato con Java in quanto fin dalle origini il suo design prevedeva un supporto alla integrazione multilinguaggio/multiparadigma. tuProlog offre la possibilità di:

- Utilizzare Classi, librerie e oggetti java all'interno del codice Prolog, offendo un passaggio dei parametri dei risultati trasparente, senza l'utilizzo di dichiarazioni o sintassi particolari, ma mantenendo separati i modelli computazionali dei due linguaggi al fine preservare le semantiche di entrambi.
- utilizzare il motore Prolog direttamente da codice Java in maniere trasparente rispetto ad una qualunque altra libreria, mantenendo separate le semantiche dei due linguaggi, ma portando la potenza della programmazione logica all'interno di un'applicazione Java.

1.1.1 OOLibrary in Java

Per scelta progettuale le funzionalità che supportano la programmazione multilinguaggio, non essendo necessarie per il funzionamento di un motore Prolog minimale, non sono state implementate nel core di tuProlog. Queste funzionalità sono quindi rese disponibili attraverso la libreria OOLibrary.

La libreria OOLibrary nella versione Java fornisce predicati che estendono il linguaggio Prolog rendendo possibile la manipolazione di oggetti e classi Java.

Utilizzando la libreria OOLibrary è possibile, all'interno di codice Prolog, creare istanze di oggetti Java, invocare metodi e accedere a proprietà.

La seguente figura, proveniente dal manuale tuProlog, mostra alcuni predicati Prolog che vengono resi disponibili dalla OOLibrary.

Functionality	Predicate(s)	Description

Object creation	`new_object(+ClassName, +ArgList, ?Ref)`
	`new_object(+'ClassName[]', [+Len], ?Ref)`
	Examples:
	`new_object('java.awt.Point', [2,3], P)`
	`new_object('java.lang.Integer', [303], n)`
	`new_object('java.awt.Point'[], [100], V)`
		Creates a *ClassName* instance (or an array of) with the constructor matching the arguments in *ArgList*. *Ref* is bound to a term representing the new object (possibly an atom like *'$obj_N'*).

Method invocation	*TargetRef* `<-` *MethodName*
	TargetRef `<-` *MethodName*`(+Arg0,+Arg1,...)`
	TargetRef `<-` *MethodName* `returns` *Res*
	TargetRef `<-` *MethodName*`(+Arg0,+Arg1,...)` `returns` *Res*
	Example 1:
	`new_object('java.awt.Point', [2,3], P),`
	`P <- getX returns X`
	Example 2:
	`Intclass = class('java.lang.Integer'),`
	`Intclass <- parseInt('200') returns N`
		Invokes the method *MethodName* on the object associated to the *TargetRef* term, possibly passing arguments *Arg0*, *Arg1*, etc., and possibly binding the return argument to the *Res* term. Static methods are invoked using `class(`*ClassName*`)` as *TargetRef*.

Lambda expressions	`new_lambda(+IFName, +Impl, ?ObjRef)`
	Examples:
	`new_lambda(`
	`'java.util.function.Predicate<Point>',`
	`'p -> p.getX()>0, myPred)`
	`new_lambda(`
	`'java.util.function.BiFunction<Long,Long,Long>',`
	`'(x,y) -> x+y', myIntBif)`
		Creates the lambda expression defined in *Impl*, adhering to the functional interface *IFName*, binding the resulting object to *ObjRef*.

Field properties	*TargetRef* `.` *FieldName* `<- set(+Arg)` *TargetRef* `.` *FieldName* `<- get(+Arg)`	Accesses the public field (property) *FieldName* of object *TargetRef* to set/get its value to *Arg*. For static fields, *TargetRef* takes the form `class(`*ClassName*`)`.

FIGURA 1 – Alcuni predicati della libreria OOLibrary per tuProlog Java

1.2 tuProlog.NET

A partire dalla versione 2.1, rilasciata in Aprile 2007, tuProlog è stato reso disponibile anche in versione .NET.

Originariamente tuProlog.NET era ottenuto da una riscrittura manuale in C# del sorgente Java.

A partire dalla versione 2.5 la versione .NET di tuProlog non è ottenuta da una riscrittura del codice sorgente ma con l'ausilio di IKVM.NET[3]. IKVM.NET è uno strumento che offre diverse funzionalità, tra cui l'esecuzione di applicazioni Java su piattaforma .NET, che è resa possibile da un'implementazione in .NET di una Java Virtual Machine, e la traduzione di bytecode Java in Common Intermediate Language(CIL). Grazie a questo processo di traduzione è stato possibile generare tuProlog.NET in forma di eseguibile .exe. All'interno dell'eseguibile sono incorporate alcune librerie distribuite conIKVM.NET che forniscono un'implementazione in .NET delle librerie Java standard utilizzate in tuProlog Java.

tuProlog.NET estende tuProlog in quanto applicazione multipiattaforma offrendo un'alternativa per le macchine che non dispongono di un JVM e in quanto applicazione multiparadigma, garantendo lo stesso livello di interoperabilità che la versione originale offre con JAVA con tutti i linguaggi del gruppo .NET.

tuProlog.NET è disponibile come eseguibile in forma di file exe e come libreria assembly.

1.2.1 OOLibrary in .NET

Nella versione 2.1 di tuProlog la libreria che consentiva l'interazione con codice Java era nominata JAVALibrary. Per la versione .NET, invece, fu creatala libreria csharpLibrary, che permetteva, analogamente alla libreria originale, la possibilità di utilizzare elementi della programmazione orientata agli oggetti all'interno di codice Prolog.

A partire dalla versione 2.2 la csharpLibrary venne sostituita dalla CLILibrary estendendo l'interoperabilità a tutti i linguaggi del gruppo .NET. Al fine di integrare le differenze sintattiche tra i diversi linguaggi venne introdotto il concetto di convenzione.

Le convenzioni sono utilizzate come supporto alla conversione dei nomi degli elementi sintattici. Grazie all'utilizzo delle convenzioni è possibile separare la sintassi relativa all'utilizzo di un'entità (classe, oggetto, metodo, proprietà) che può variare per poter essere resa coerente con quella del linguaggio orientato agli oggetti utilizzato, dalla realizzazione del metodo stesso. In pratica le convenzioni permettono di avere sintassi diverse per lo stesso predicato Prolog in base al linguaggio con il quale si sta interagendo.

La seguente figura, proveniente dal manuale di tuProlog, mostra il modo in cui l'utilizzo di convenzioni modifica la sintassi dei predicati Prolog. Ad esempio il predicato `Obj <- GetId returns StudentNumber`se vengono applicate le convenzioni di Visual Basic diventa `Obj.id <- get(StudentNumber)`, ottiene quindi una forma più coerente con il concetto di proprietà di .NET.

```
visualbasicWithoutConvention :-
  new_object('VBStudent.Student',[123456, john, smith], Obj),
  Obj <- 'PrintStudent' returns Student,
  Obj <- get_Id returns StudentNumber,
  class('VBStudent.Student,VBStudent') <- get_StaticProperty returns Value,
  new_object('VBStudent.Student, VBStudent[]',[10], Array).

visualbasicWithConvention :-
  load_convention('VBConvention.dll','VBConvention.VBDotNet',Conv),
  new_object(Conv,'VBStudent.Student, VBStudent',
                                  [123456, john, smith], Obj),
  Obj <- printStudent returns Student,
  Obj.id <- get(StudentNumber),
  class('VBStudent.Student, VBStudent').staticProperty <- get(Value),
  new_object(Conv, 'VBStudent.Student, VBStudent()',[10], Array).

javaWithoutConvention :-
  new_object('javastudent.Student',[123456, john, smith], Obj),
  Obj <- printStudent returns Student,
  Obj <- getId returns StudentNumber,
  class('javastudent.Student') <- printInfoUniv returns University,
  new_object('javastudent.Student[]',[10], Array).

javaWithConvention :-
  load_convention('JavaConvention.dll','JavaConvention.Java',Conv),
  new_object('javastudent.Student',[123456, john, smith], Obj),
  Obj <- 'PrintStudent' returns Student,
  Obj <- getId returns StudentNumbers,
  class('javastudent.Student') <- printInfoUniv returns University,
  new_object('javastudent.Student[]',[10], Array).
```

FIGURA 2 - Utilizzo delle convenzioni per Java e Visual Basic

A partire dalla versione 2.5 di tuProlog, grazie all'introduzione di IKVM, la libraria CLILibrary è stata rinominata OOLibrary ed estesa anche all'interoperabilità con JAVA.

La libreria OOLibrary in .NET differisce dalla quella JAVA a causa delle convenzioni, e per questo motivo risulta essere l'unica parte di codice non generata automaticamente nel processo di traduzione.In seguito alla scelta di utilizzare le convenzioni, infatti, venne presa la decisione di non introdurle nella versione JAVA, al fine di non dover fare modifiche inutili su del codice funzionante. Le funzionalità specifiche della versione.NET sono quindi implementate direttamente in C# e sono contenuti nei file Convention.cs e OOLibrary.cs che estende il comportamento della OOLibrary originale, ottenuta tramite la traduzione IKVM, facendo uso di subclassing.

I Predicati PROLOG disponibili per la versione .NET differiscono quindi per la presenza delle convenzioni.

1.3 IKVM.NET

IKVM.NET[3] è un progetto open source creato da Jeroen Frijters, che ne ha poi portato avanti lo sviluppo insieme ad un piccolo gruppo di programmatori, ed è distribuito con una licenza[5] che non comporta restrizioni di utilizzo.

IKVM.NET è un'implementazione di Java per le piattaforme .NET e Mono, ed è costituito da diversi componenti:

- Java Virtual Machine implementata in .NET;

- Implementazione .NET di librerie JAVA.
- Strumenti per l'interoperabilità tra Java e .NET

Il meccanismo alla base del funzionamento di IKVM è un processo di traduzione che consente la generazione di Common Intermediate Language (CIL) a partire da bytecode Java. Questo processo può avvenire sia in modalità statica sia in modalità dinamica a tempo di esecuzione.

Complessivamente quindi IKVM.NET consente lo sviluppo in Java di librerie e applicazioni .NET.

1.3.1 Modalità statica

IKVM comprende al suo interno lo strumento ikmvc.exe. Questo strumento permette la traduzione di un'applicazione o di una libreria Java in Common Intermediate Language (CIL) e di memorizzare il codice in un assembly della piattaforma .NET (file .exe o dll).

L'operazione di traduzione viene eseguita a livello di bytecode, è necessario quindi essere in possesso di codice JAVA già compilato o di un compilatore java. Nel caso in cui eseguire una traduzione a partire da del codice sorgente, è sufficiente dotarsi di un qualunque compilatore Java standard, ed eseguire la fase di compilazione precedentemente a quella di traduzione.

Il processo di traduzione è supportato dalle implementazione in .NET della libreria standard di Java che viene distribuita con IKVM, ad esempio nell'assembly `IKVM.OpenJDK.Core.dll` sono presenti i package principali come `java.lang`,`java.io`, `java.util` e `java.awt`. Se viene tradotta un'applicazione Java che contiene riferimenti a queste librerie, nell'assembly generato vengono inseriti i riferimenti agli assembli IKVM corrispondenti ed è quindi necessario fare in modo che l'ambiente .NET sia in grado di localizzarli a tempo di esecuzione inserendoli nella GAC o nella cartella corrente dell'applicazione.

Non tutte le API della libreria standard sono implementate.

tuProlog.NET generato con IKVM è stato valutato essere il 15% più lento, in media, della controparte Java.

Questa modalità rende possibile la scrittura interamene in Java di applicazioni o librerie .NET, ed è per questo usata nel caso di tuProlog per ottenere la versione tuProlog.NET.

1.3.2 Modalità dinamica

Grazie all'implementazione .NET di una Java Virtual Machine fornita con IKVM, oltre alla traduzione statica è possibile effettuare la traduzione da bytecode a CIL in maniera dinamica durante l'esecuzione, ottenendo così l'esecuzione di codice Java direttamente di piattaforma .NET.

Il nome di questo strumento è ikvm.exe e può essere inteso come il corrispondente di java.exe, dal momento che vi è corrispondenza anche nelle opzioni disponibili (ad esempio è possibile specificare il classpath utilizzando l'opzione cp).

Ovviamente l'esecuzione diretta di codice Java in ambiente .NET risulterà molto meno performante perché dovrà essere eseguita anche la fase di traduzione.

Per quanto riguarda tuProlog la possibilità di eseguire direttamente codice Java viene sfruttata nella libreria OOLibrary della versione .NET per permettere l'interazione con oggetti e classi in forma di bytecode.

1.3.3 ikvmstub.exe

Per supportate lo sviluppo in Java di applicazioni .NET è fornito lo strumento ikvmstub.exe. Utilizzando ikvmstub.exe è possibile generare un file con estensione .jar a partire da un file assembly. Il file generato, come suggerito da nome, è un file *stub* ovvero contiene tutte le classi e le interfacce Java corrispondenti a quelle .NET ma non contiene alcun codice di implementazione. Questo file può essere incluso come libreria all'interno di un progetto Java, rendendo così possibile l'utilizzo, delle classi presenti nell'assembly. L'inclusione di questi file nel progetto consente di sfruttare le funzionalità di auto completamento del IDE e di superare il controllo sui tipi eseguito dai compilatori Java. In questo modo è possibile scrivere e compilare in Java un'applicazione che utilizzi classi standard del Framework .NET o definite in librerie .NET. Dopo essere stata compilata l'applicazione sviluppata deve essere tradotta con ikvmc ed integrata con gli assembly che contengono le classi utilizzate. Ovviamente, dato che i file jar generati non contengono l'implementazione, le applicazioni sviluppate con l'ausilio di ikvmstub.exe non possono essere eseguite su una JVM standard.

2 Obsolescenza di IKVM rispetto a Java 9

In Aprile 2017 attraverso un post sul blog di IKVM è stata annunciata l'interruzione dello sviluppo progetto[4].

Nonostante nel suddetto post l'autore si auguri la presa in carico del progetto da parte di un qualche altro sviluppatore, tutt'ora il progetto IKVM risulta fermo.

Nella sua ultima release IKVM rimane quindi compatibile con Java 8.

In settembre 2017 viene rilasciata come major release Java 9 che introduce come principale novità i moduli.

I moduli introducono un nuovo livello di aggregazione al di sopra dei package, fornendo un sistema più completo per la definizione della visibilità e delle dipendenze, rispetto a quello precedentemente offerto dai package, e definiscono la nuova unità di sviluppo e di rilascio rivoluzionando il modo di strutturare il progetto e portando una netta rottura con le versioni precedenti di Java.

Inoltre Oracle[6]ha annunciato una nuova politica per lo sviluppo di Java che prevede una maggiore frequenza di rilascio delle nuove versioni della piattaforma.

2.1 Moduli Java

I moduli Java sono un meccanismo per partizionare applicazioni e package Java.

Un modulo consiste in un insieme di package. Ogni modulo deve specificare quali tra i package al suo interno devono essere visibili agli altri moduli e deve definire quali sono i moduli da cui egli stesso dipende.

I moduli fanno parte del Java Module System (JPMS) noto anche con il nome di Jigsaw Project[7].

Tutte le API della piattaforma Java sono state riorganizzate in moduli. Grazie a questa nuova impostazione strutturale e possibile specificare quali sono i moduli richiesti da una specifica applicazione e incorporarli nel pacchetto di distribuzione. Il vantaggio che si ottiene da questa caratteristica sono minori dimensioni dell'applicazione in quanto prima di Java 9, non essendo presente la suddivisione in moduli, era necessario includere anche classi non utilizzate.

I moduli dichiarano esplicitamente quali dei package contenuti devo essere visibili all'esterno, di fatto questa caratteristica permette di introdurre un nuovo livello di incapsulamento che permette di distinguere tra package privati e pubblici.

Grazie alla presenza dei moduli la Java Virtual Machine è in grado di controllare l'intero grafo di dipendenza dell'applicazione durante la fase di caricamento dell'applicazione.

Nelle versioni precedenti la mancanza di una classe poteva essere rilevata solo a runtime nel momento in cui l'applicazione tentava di utilizzarla.

Un modulo Java viene definito tramite un Module Descriptor che consiste in file denominato module-info.java che contiene le seguenti definizioni.

- Nome Modulo, il nome deve rispettare le stesse convenzioni dei package Java
- Elenco dei package che vengono resi visibile
- Elenco dei moduli da cui dipende

All'interno dei moduli non sono permesse dipendenze circolari e ogni package deve essere esportato da un unico modulo per volta.

La locazione dei moduli utilizzati da un'applicazione deve essere specificata all'interno del `MODULEPATH` del progetto.

Per mantenere compatibilità con librerie di versioni precedenti di Java, è possibile utilizzare il classpath per utilizzare package che non sono distribuiti all'interno del modulo, Queste classi verranno automaticamente incluse in un modulo chiamato unnamed module che esporta tutti i package al suo interno. Le classi contenute nell'unnamed module non possono però essere utilizzate da altri moduli, è pero possibile specificare il path di un package all'interno del modulepath, in questo modo verrà generato un modulo automatico con lo stesso nome del package che potrà poi essere riferito dagli altri moduli.

Java 9 permette la creazione di file JAR che contengono codice compatibile con differenti versioni di Java.

Questa è la struttura di un file jar multiversione di Java.

```
jar root
  - A.class
  - B.class
  - C.class
  - D.class
  - META-INF
     - versions
        - 9
           - A.class
           - B.class
```

La cartella com a livello radice contenente le classi compatibili con versioni Java precedenti a Java 9. Dal momento che le versioni precedenti non comprendono i file multiversione, è possibile avere compatibilità con un'unica versione precedente a Java 9.

La cartella META-INFcontiene il file Manifest.MF e la cartella versions.

Le cartelle Versions contengono classi compatibili con diverse versioni di java.

Il nome delle sottocartelle corrisponde al nome della versione target.

2.2 Ciclo di rilascio di Java

Fino a Java 9 il periodo di tempo di tempo compreso fra il rilascio di due versioni di Java differenti era programmato per essere di due anni. Nonostante questo però il rilascio di Java 8 e Java 9 hanno avuto un ritardo di, rispettivamente, 8 e 18 mesi.

A partire da Java 9, Oracle ha quindi annunciato un cambiamento nella modalitàdi rilascio delle nuove release di Java, abbandonando l'approccio basato su major release a favore di uno basato su uno stream di minor release.

Questa decisione è stata motivata affermando che negli anni passati le grandi modifiche apportate dalle major release (lambda expression per Java e moduli per Java 9) causavano un ritardo nel rilascio degli altri aggiornamenti minori presenti nella nuova versione.

E' stato quindi definito un ciclo semestrale di rilascio per cui sono state rilasciare Java 10 a marzo 2018, Java 11 a Settembre 2018 e verrà rilasciata java 12 a Marzo 2019.

3 Analisi del problema

La versione principale di tuProlog è attualmente sviluppata in Java 8, ma si prevede in futuro il rilascio di una versione in Java 9 o successive che possa utilizzare i moduli e le altre caratteristiche introdotte.

L'aggiornamento di tuProlog a Java 9 andrebbe a creare un punto di rottura con la versione .NET essendo questa generata con l'ausilio di IKVM che limita il suo supporto a Java 8.

Nonostante la versione Java di tuProlog sia attualmente la più utilizzata e performante rinunciare a tuProlog.NET porterebbe alla perdita di alcuni casi d'uso.

La versione .NET permette l'esecuzione di tuProlog come applicazione a sé stante su piattaforme aggiuntive, ovvero .NET e mono, permettendone l'uso su macchine prive di JVM. Inoltre rende disponibile tutte il codice in forma di assembly garantendo così la possibilità di utilizzare tuProlog o sue componenti come librerie all'interno di applicazioni .NET.

tuProlog.NET dispone, inoltre, di una versione modificata della libreria OOLibrary (vedi 1.2.1). Grazie a questa libreria il supporto alla programmazione linguaggio viene esteso anche ai linguaggi della famiglia .NET (C#, F#, VB), ovvero viene reso possibile l'utilizzo, da codice Prolog, di classi, oggetti e altri elementi caratteristici dei suddetti linguaggi a paradigma orientato agli oggetti.

In conclusione i casi d'uso che devono essere considerati sono:

- tuProlog come applicazione a sé stante su piattaforma .NET
- Interazione con classi e oggetti contenuti in librerie .NET da codice Prolog
- Uso di tuProlog come libreria .NET all'interno di un'applicazione .NET

Tra i vari caso d'uso l'eventuale mancanza della piattaforma .NET comporterebbe un problema pratico minore, sia per la facilità con cui è possibile reperire e installare una Java Virtual Machine sia perché gli aggiornamenti di Java permettono ora il rilascio di eseguibili che incorporano una JVM minimale, che quindi può non essere presente sulla macchina destinazione.

La perdita nell'ambito della programmazione multiparadigma è, invece, più rilevante essendo questo un elemento di interesse nel progetto tuProlog fin dalle origini. L'introduzione di forme di interazione con i linguaggi della famiglia .NET è rilevante, in quanto questi linguaggi contengono costrutti linguistici non presenti in Java come delegati o eventi, e introducono quindi nuovi aspetti interessanti per la ricerca sulla programmazione multiparadigma con linguaggi a paradigma logico e ad oggetti. Inoltre nella OOLibrary versione .NET viene introdotto il concetto di convenzione non presente nella versione Java. Le convenzioni introducono nell'ambito della programmazione multiparadigma il concetto di variabilità sintattica dei predicati logici in relazione ai diversi linguaggi orientati agli oggetti. Questo aspetto verrebbe perso se Java tornasse ad essere l'unico linguaggio disponibile per l'interazione.

L'ultimo caso d'uso prevede l'utilizzo di tuProlog come motore Prolog per applicazioni in linguaggio orientato agli oggetti che vogliono introdurre parti di codice a paradigma logico.
tuProlog.NET essendo disponibile come assembly rendeva disponibile questa funzionalità alle applicazioni .NET, con estrema semplicità in quanto poneva come unico requisito la referenziazione della libreria all'interno del progetto.

I casi che sono stati elencati saranno il punto di partenza per la definizione dei possibili approcci di soluzione.

Si è scelto di prendere in considerazione unicamente i casi d'uso che prevedono l'utilizzo di tuProlog come prodotto finito. La disponibilità del codice completo di tuProlog nella forma di assembly rende infatti possibile apportare modifiche ed estensioni attraverso lo sviluppo di componenti.NET e consente il riutilizzo delle classi stesse di tuProlog all'interno di altri progetti.
Allo stesso modo si è scelto di non considerare il caso d'uso per cui è possibile definire nuovi predicati Prolog attraverso lo sviluppo di una libreria tuProlog in un linguaggio.NET.
La motivazione per cui è stata compiuta questa scelta è che, nonostante siano attualmente disponibili, questi casi d'uso non sono molto utilizzati in quanto la piattaforma di riferimento per lo sviluppo di tuProlog rimane sostanzialmente Java.

3.1 Possibili approcci

Non potendo più contare, per le future versioni di tuProlog, su IKVM per la generazione di una versione .NET si pone il problema di identificare, se esiste, un'alternativa che permetta di mantenere i casi d'uso attualmente forniti.

I primi approcci di soluzione sono stati individuati considerando l'alternativa più semplice e ottimistica, ovvero il mantenimento di tuProlog.NET come applicazione completa. Questa modalità è quella che richiederebbe la minor preparazione per poter essere attuato, in quanto necessiterebbe esclusivamente di modifiche il processo di traduzione. E' quindi evidente fin da subito in che modo si dovrebbe intervenire se fosse disponibile un'alternativa concreta.

Approcci di soluzione successivi sono stati definiti supponendo l'abbandono di tuProlog.NET. Questa ipotesi è stata fatta in quanto sono stati valutati più interessanti i casi d'applicazione specifici di quest'ultimo piuttosto che lo sviluppo di tuProlog.NET in sé per sé. Questa categoria di approcci prevede quindi un'estensione della versione Java al fine di integrare in essa la possibilità di interoperare con applicazioni o librerie .Net. Questo aspetto, essendo Java la piattaforma di riferimento, può essere valutato come un punto a favore.

Come terza possibilità si è valutata l'ipotesi di una piattaforma alternativa per tuProlog che però potesse avere una qualche forma di compatibilità con i linguaggi .NET. In questo caso sarebbe necessario intervenire sia sul processo di traduzione come avviene per il primo tipo di approccio, sia dipendentemente da caso specifico, per integrare i casi d'suo d'interesse.

E' evidente come non tutti gli approcci siano sullo stesso piano: il primo è sicuramente preferibile agli due. Tuttavia, valutando le difficoltà presenti nel trovare concretamente una soluzione adeguate, si è scelto di mantenere la base di ricerca più ampia possibile nel tentativo di massimizzare le possibilità di successo.

In seguito a questa prima schematizzazione sono state individuate le seguenti alternative.

- Mantenimento tuProlog.NET
    - Traduzione manuale
    - Traduzione automatica
- Abbandono tuProlog.NET e recupero dei casi d'uso in tuProlog Java
    - Bridging
    - tuProlog come ScriptEngine

- Nuova piattaforma a sostituzione di .NET
    - Windows Universal Platform

Al fine di valutare un approccio di soluzione è necessario considerare quali siano i casi d'uso che potrebbero essere recuperati. Eseguire questa valutazione è però complicato in quanto, a priori, non è possibile valutare l'impatto di un eventuale applicazione concreta. Come prima forma di catalogazione si è quindi scelto di associare ad ogni approccio l'elenco di casi d'uso recuperabili nell'ipotesi più ottimistica.

3.1.1 Mantenimento tuProlog.NET

L'approccio che prevede di mantenere una versione .NET separata che segua l'evoluzione della versione principale in Java è il più facile da analizzare in quanto corrisponde all'approccio attuale. Con questa alternativa, nel caso in cui fosse possibile ottenere tuProlog.NET completo di tutte le funzionalità di tuProlog Java, verrebbero recuperati tutti i casi d'uso.

In questo caso è necessario determinare un processo di sviluppo per tuProlog.NET. Le alternative che si presentano sono la scrittura manuale del codice o un processo di traduzione automatico con il supporto di uno strumento esterno. In entrambi casi l'obbiettivo finale sarebbe quello di ottenere una versione .NET totalmente coerente alla versione originale.

Nel caso di scrittura manuale sarebbe necessario scegliere uno dei linguaggi del gruppo .NET esi dovrebbe quindi effettuare una traduzione a partire dal codice Java. Questo processo necessita però di una particolare attenzione al fine ottenere lo stesso esatto comportamento della versione originale. E' quindi evidente che lo sviluppo manuale di tuProlog.NET comporta un alto costo come forza lavoro sia come tempo necessario per la scrittura del codice in sé sia per la necessità di mantenere le due versioni totalmente allineate.

Nel caso di traduzione automatica è invece necessario individuare uno strumento che possa sostituire IKVM. In questo caso la traduzione potrebbe avere come destinazione sia codice compilato, come nel caso di IKVM, sia codice sorgente. Per seguire questa seconda strada si potrebbero quindi ricercare dei convertitori sorgente a sorgente, ad esempio da Java a C#.

3.1.2 Abbandono tuProlog.NET e recupero dei casi d'uso in tuProlog Java

Considerando invece l'abbandono di tuProlog.NET e la conseguente inclusione dei casi d'uso d'interesse in tuProlog Java, possono essere considerati strumenti che permettono l'interoperabilità tra gli ambienti d'esecuzione Java e .NET. Questo tipo di approccio è denominato *Bridging*. In questa categoria rientrano sia soluzioni native ai linguaggi che strumenti esterni. Ovviamente una soluzione con questo approccio comporterebbe delle modifiche al progetto Java originale e sarebbe necessario valutarne compatibilità e invasività.

In alternativa per recuperare il caso d'uso di tuProlog come motore Prolog all'interno di applicazioni .NET si potrebbe pensare di sfruttare interfacce standard per motori di script. Questo approccio è già presente per la versione Java che può essere utilizzata come motore Prolog all'interno di una applicazione Java attraverso l'interfaccia standard definita nella JSR 223.

3.1.3 Nuova piattaforma a sostituzione di .NET

Per quanto riguarda il porting di tuProlog è necessario individuare eventuali piattaforme che presentino una qualche forma di compatibilità con la piattaforma .NET. Un caso interessante può essere la piattaforma UWP (Universal Windows Platform), per la quale si potrebbe, in generale, immaginare un tuProlog UWP. Un'applicazione UWP può utilizzare una parte delle librerie appartenenti al framework .Net, ovvero le API denominate .NET Standard. In base alle funzionalità disponibili in queste API potrebbe essere possibile recuperare parte dei casi d'suo evidenziati. Inoltre questa nuova versione renderebbe tuProlog disponibili per tutti i dispositivi fissi e mobili con Windows 8 o 10.

3.2 Analisi dei possibili approcci di soluzione

Preliminarmente è necessario definire dei parametri di valutazione che consentano l'analisi della applicabilità delle soluzioni concrete. Devono quindi essere individuate delle condizioni necessarie che siano valide trasversalmente per tutti gli approcci di soluzione.

Date queste premesse si determina che una soluzione può essere considerata applicabile se rispetta i seguenti requisiti minimi:

- Supportare componenti del linguaggio Java non supportate da IKVM
- Necessitare di una quantità di lavoro inferiore alla traduzione manuale.

Il primo requisito esprime la necessità di superare il limite di IKVM di non compatibilità con Java 9. Il secondo requisito invece evidenzia la criticità del lavoro necessario per applicare una soluzione. La traduzione manuale rappresenta un punto di riferimento minimo in quanto è l'approccio di soluzione che permette la massima qualità, in termini di prestazione e di copertura del linguaggio Java, di tuProlog.NET. Se una soluzione prevedesse un carico di lavoro maggiore alla traduzione manuale quest'ultima sarebbe sicuramente preferibile.
Chiaramente, rispettare questi requisiti è necessario ma non sufficiente.

La maggior parte degli approcci precedentemente elencati necessita del supporto di strumenti esterni. La reale applicabilità di una soluzione dipende quindi dalle caratteristiche concrete degli strumenti disponibili a supporto di un determinato approccio. Inoltre rendere tuProlog dipendente in parte da uno strumento appartenente ad un altro progetto comporta ulteriori problematiche anche in termini di costi o licenze, che è un aspetto tutt'altro che secondario.

Affinché uno strumento esterno possa essere considerato valido devono essere rispettate determinate caratteristiche.

- Deve essere compatibile con le ultime versioni di Java e .NET e con le librerie standard che vengono usate all'interno di tuProlog.
- Deve essere disponibile gratuitamente o a costo contenuto.
- Deve essere rilasciato con una licenza compatibile con la licenza LGPL di tuProlog.
- Deve essere utilizzabile con uno sforzo moderato.
- Deve fare parte di un progetto attivo e periodicamente aggiornato.

Per i punti legati al costo e la licenza, l'opzione migliore sarebbe quella di trovare uno strumento open-source. Trovare uno progetto open source che riesca a rimanere aggiornato rispetto

l'evoluzione delle piattaforme Java e .NET per un lungo periodo di tempo è una situazione abbastanza rara, in effetti IKVM ha presentato un'eccezione rispetto alla maggior parte di progetti a lui simili che invece sono stati caratterizzati da un tempo di vita breve. D'altro canto, progetti commerciali, che potrebbero garantire un supporto più stabile e duraturo, potrebbero non rispettare questi punti.

3.2.1 Mantenimento IKVM

Il mantenimento di IKVM è una possibilità da prendere in considerazione. Al fine di mantenere una versione di tuProlog.NET generata con IKVM è necessario verificare la possibilità di limitare l'utilizzo di Java 9 o superiore a punti critici del codice di tuProlog. Se questo fosse fattibile potrebbe essere possibile strutturare un sistema di traduzione ibrido tra IKVM e scrittura manuale o IKVM più altri strumenti.

3.2.2 Traduzione a livello di bytecode

L'approccio basato su traduzione a livello di bytecode necessita di uno strumento che riesca a sostituire le funzionalità di ikvmc. In questo caso valutare la compatibilità di uno strumento con la piattaforma Java e .NET significa individuare quali sono le librerie standard di Java la cui traduzione è supportata.
In questo aspetto IKVM può vantare la copertura di numerose librerie. Caratteristica secondaria è invece il calo di prestazione sul codice tradotto in quanto non ci sono requisiti troppo stringenti sui tempi di risposta e di esecuzione. E' fondamentale però che un eventuale strumento sia compatibile con Java 9 oche prometta compatibilità in release future.

3.2.3 Traduzione automatica a livello di codice sorgente

Invece che effettuare una traduzione a livello di bytecode si può pensare di intervenire a livello di sorgente ad esempio traducendo il codice Java in codice C#. Intraprendere questa via porterebbe alcuni vantaggi rispetto all'approccio attuale, in quanto sarebbero disponibile i sorgenti della versione .NET e sarebbe quindi possibile intervenire manualmente per la modifica di alcune parti di codice. Le problematiche di questo approccio sono le stesse presentate da quello basato su traduzione di bytecode in quanto si verrebbe comunque a dipendere da uno strumento esterno. Anche in questo caso gli eventuali limiti sarebbero quelli posti dallo strumento e le principali criticità sarebbero la copertura delle librerie Java e la compatibilità con Java 9.

3.2.4 Bridging

Un approccio di soluzione basato su uno strumento che consenta *bridging*[8], a differenza degli approcci basati su traduzione potrebbe richiedere una riprogettazione dell'architettura del progetto. Infatti tale soluzione prevederebbe che determinate componenti del codice fossero eseguite su una Java Virtual Machine mentre altre su piattaforma .NET. E' necessario quindi valutare se l'introduzione di nuove funzionalità potrebbe essere eseguita limitando i cambiamenti nelle nuove parti di codice inserito, o se comporterebbe la modifica di parti del progetto già stabili. Data la diffusione della versione Java di tuProlog, è infatti evidente che non sarebbe semplice apportare modifiche all'architettura che non ne compromettessero l'utilizzabilità.
Inoltre è necessario verificare le potenzialità degli strumenti, sia in termini di librerie o più genericamente di funzionalità supportate, sia in termini di perdita di prestazione e di eventuali vincoli aggiuntivi che dovrebbero essere rispettati per garantire il funzionamento. In generale non è

quindi possibile, a priori, avere una prospettiva chiara su un eventuale, in quanto tutti gli aspetti citati dipendono dalle caratteristiche specifiche di eventuali strumenti concreti.

3.2.5 tuProlog come Script Engine

La disponibilità di tuProlog come libreria .NET serve principalmente, a renderlo disponibile come interprete Prolog all'interno di un'applicazione .NET esterna, per permettere lo sviluppo di applicazioni multiparadigma. In pratica, quindi, non è necessario avere visibilità sulle classi interne della libreria, ma unicamente sull'interfaccia che permette l'utilizzo di tuProlog come motore di script. Rendere disponibile tuProlog come ScriptEngine significa quindi, individuare una modalità di interfacciamento standardizzata che favorisca l'implementazione di una soluzione basata su *Bridging*. Ovviamente questo approccio consentirebbe esclusivamente il recupero di questo particolare caso d'uso.

3.2.6 Piattaforma UWP

La piattaforma UWP potrebbe permettere di recuperare alcune forme di interazione con la piattaforma .NET. L'applicabilità di questo approccio dipende da due diversi fattori. Infatti non tutte le API del framework .NET sono disponibili per piattaforma .NET, è quindi opportuno valutare se, effettivamente i casi d'uso evidenziati possano essere recuperati in un eventuale tuProlog UWP.
Come secondo aspetto è necessario individuare uno strumento che garantisca supporto al porting di tuProlog su questa piattaforma.

Tabella comparativa approcci

APPROCCIO	Casi d'uso recuperabili	Condizioni per utilizzabilità
Mantenimento IKVM	-tuProlog come applicazione a sé stante su piattaforma .NET -Interazione con classi e oggetti contenute in librerie .NET da codice Prolog -Utilizzo di tuProlog come motore Prolog all'interno di un'applicazione .NET	-Limitare l'utilizzo di versioni di Java superiori a 8 a specifiche parti di codice - Individuare un sistema per produrre le parti di codice che non possono essere tradotte da IKVM (traduzione o scrittura)
Traduzione Manuale	-tuProlog come applicazione a sé stante su piattaforma .NET -Interazione con classi e oggetti contenute in librerie .NET da codice Prolog -Utilizzo di tuProlog come motore Prolog all'interno di un'applicazione .NET	-Individuare un sistema che permetta di ridurre il carico di lavoro necessario a portare avanti questo tipo di approccio.
Traduzione Sorgente	-tuProlog come applicazione a sé stante su piattaforma .NET -Interazione con classi e oggetti contenute in librerie .NET da codice Prolog -Utilizzo di tuProlog come motore Prolog all'interno di un'applicazione .NET	-Trovare uno strumento adeguato in termini di costo, licenza e supporto -Trovare uno strumento compatibile con Java 9 -Trovare uno strumento che traduca una buona parte delle librerie standard di JavaSE
Traduzione bytecode	-tuProlog come applicazione a sé stante su piattaforma .NET -Interazione con classi e oggetti contenute in librerie .NET da codice Prolog -Utilizzo di tuProlog come libreria .NET all'interno di un'applicazione .NET	-Trovare uno strumento adeguato in termini di costo, licenza e supporto -Trovare uno strumento compatibile con Java 9 -Trovare uno strumento che traduca una buona parte delle librerie standard di JavaSE
Bridging	-Interazione con classi e oggetti contenute in librerie .NET da codice Prolog -Utilizzo di tuProlog come motore Prolog all'interno di un'applicazione .NET	-Individuare una strategia di riscrittura del codice Java che non sia troppo invasiva o dispendiosa. -Trovare uno strumento adeguato in termini di costo, licenza e supporto -Trovare uno strumento compatibile con Java 9 -Trovare uno strumento che garantisca l'interoperabilità tra le due piattaforme in entrambe le direzioni.
tuProlog come ScriptEngine	-Utilizzo di tuProlog come motore Prolog all'interno di un'applicazione .NET	-Trovare un'interfaccia standard all'interno di .NET che possa essere implementata da tuProlog.
Piattaforma UWP	-Interazione con classi e oggetti contenute in librerie .NET da codice Prolog	-Verificare l'adeguatezza della piattaforma UWP rispetto all'implementazione dei casi d'uso evidenziati -Trovare uno strumento che supporti porting da codice Java a piattaforma UWP.

4 Esplorazione delle possibilità

In questo capitolo verranno affrontati in maniera più dettagliata i vari metodi d'approccio e saranno presentati e commentati alcuni strumenti.

4.1 Mantenimento IKVM

Nonostante non venga più aggiornato dal 2017 IKVM rimane probabilmente il migliore strumento che permette la generazione di codice automatico. Il mantenimento di IKVM come strumento che permetta la generazione di tuProlog.NET per le future versioni di tuProlog è un approccio risolutivo da tenere in considerazione.

IKVM effettua inizialmente un controllo sulla major version del bytecode Java, e non esegue la traduzione se questa risulta essere ad una versione superiore alla 52.0 (ovvero quella corrispondente a Java 8. Questo significa che a prescindere dall'utilizzo di classi specifiche di Java 9 o superiore nel caso si volesse sfruttare IKVM sarebbe necessario compilare il progetto sia con jdk 1.8 che con jdk 9.

Il primo problema che si pone è la gestione dei moduli. I moduli però introducono scope di visibilità più stringenti di quelli di un progetto basato esclusivamente su package, inoltre in pratica consistono semplicemente in file .java nel livello più alto dell'albero dell'progetto. Date queste premesse è possibile introdurre i moduli nella versione tuProlog Java, e poi eliminarli facilmente per ottenere una versione Java compatibile con IKVM da cui effettuare la traduzione per ottenere la versione .NET.

E' invece più problematico l'utilizzo di nuove classi introdotte con Java 9, dal momento che le parti codice in cui verrebbero utilizzate non sarebbero più compatibili con IKVM. Quindi per ogni nuova funzionalità implementata con queste sarebbe necessario una scrittura alternativa o in Java 8 o direttamente in C#. La fattibilità di questa soluzione dipende dalla quantità di codice aggiuntivo che sarebbe necessario scrivere unicamente per la versione .NET.

Continuare ad utilizzare o meno IKVM dipende quindi dalla possibilità di strutturare il progetto tuProlog individuando punti critici nei quali utilizzare le funzionalità specifiche di Java 9 sulle quali effettuare un processo di traduzione manuale.

Si presentano le seguenti possibilità:

- Si individua un core sul quale si decide di non utilizzare nuove classi Java 9. In questo modo sarebbe possibile grazie a IKVM tradurre una buona parte del progetto e si limiterebbe l'intervento manuale. Questa soluzione risulta però di fatto impraticabile in quanto perché inverte le priorità tra versione tuProlog Java e .NET, limitando lo sviluppo del primo in favore del secondo.

- Si concede uno sviluppo libero lato Java. Questo approccio però rende non quantificabile lo sforzo di traduzione, in quanto non sarebbe presente un core ben definito e l'utilizzo di IKVM porterebbe alla generazione di librerie incomplete che necessiterebbero di integrazione manuale.

Più in generale continuare ad utilizzare IKVM è sensato solo in un contesto di sviluppo che preveda l'introduzione di modifiche al codice compatibili con Java 8. Infatti utilizzare in porzioni di codice

classi di funzionalità specifiche Java 9 renderebbe queste non traducibili con IKVM e quindi non trasferibili alla versione .NET. Allo stesso modo non si trarrebbe nessun vantaggio dall'utilizzo di IKVM per le parti di codice che rimarrebbero invariate dato che il codice della versione .NET rimarrebbe identico a quello attuale. L'introduzione di modifiche compatibili con Java 8 costituisce quindi l'unico caso per cui l'utilizzo di IKVM porterebbe ad una versione aggiornata di una componente dell'applicazione .NET. Facendo riferimento al primo caso ad esempio se il core dovesse rimanere totalmente invariato non si otterrebbe nessun vantaggio dal non utilizzo di classi Java 9 in quanto il codice generato automaticamente sarebbe quello già presente nella versione attuale.

In conclusione il mantenimento di IKVM non risulta essere una soluzione valida.
Infatti adoperare questo strumento richiederebbe un approccio di tipologia artigianale e difficilmente standardizzabile. Inoltre, è evidente che, con l'evoluzione delle piattaforme Java e .NET il numero delle funzionalità non supportate da IKVM continuerà ad aumentare. IKVM è quindi destinato a diventare obsoleto in breve tempo, per questo motivo approcci di soluzione basati su di esso non possono essere considerati applicabili a lungo termine.

Dalle osservazioni fatte risulta più sensato se si volesse mantenere una versione.NET, non basarsi su IKVM come sistema di traduzione, ma o considerare un sistema di traduzione differente o separare del tutto il processo di sviluppo delle due versioni ed estendere tuProlog.NET in maniera indipendente.

4.2 Traduzione manuale

La traduzione manuale era l'approccio utilizzato per lo sviluppo di tuProlog.NET nelle sue prime versioni. La riscrittura dei sorgenti richiede conoscenza sia del linguaggio Java che di uno dei linguaggi della famiglia .NET. Tra questi probabilmente il linguaggio più adeguato ad effettuare una trascrizione è C# che è quello più sintatticamente simile a Java. La traduzione da Java a C#, nel contesto di tuProlog ovvero di applicazione per sistemi desktop, non presenta particolari problematiche in quanto a espressività poiché non ci sono costrutti linguistici presenti in Java che non abbiano corrispondenze in C#. Se la traduzione avvenisse nel verso opposto invece sarebbe necessario reingegnerizzare parti di codice per poter sostituire alcune costrutti esclusi di .NET. Dal punto di vista dell'ottimizzazione la traduzione è l'approccio che consente la scrittura di codice più performante.

Tuttavia il processo di traduzione manuale venne abbandonato già a partire dalla versione 2.5 di tuProlog, in quanto le difficoltà dovute al mantenere coerenti le due versioni furono considerate eccessive. Questo tipo d'approccio venne quindi valutato essere, nella prospettiva di crescita del progetto, insostenibile. Dato l'aumento della complessità di tuProlog, reintrodurre questo approccio questo comporterebbe il dover affrontare le stesse problematiche, che anzi sarebbero ulteriormente accentuate, di conseguenza tale approccio appare essere ancor meno adeguato.

Risulta quindi evidente che un approccio di traduzione puramente manuale non può essere preso in considerazione, mentre la fattibilità di soluzioni ibride che prevedano la traduzione di parti codici limitate sono fortemente condizionate dagli eventuali strumenti di supporto.

4.3 Traduzione automatica

Automatizzare il processo di traduzione da codice Java a codice per piattaforma.NET permette di risparmiare il lavoro necessario per la traduzione manuale. In linea di principio il processo di traduzione può avvenire a livello di codice sorgente o di bytecode.

4.3.1 Conversione di bytecode

L'approccio basato sulla conversione da bytecode è quello attualmente utilizzato con IKVM grazie allo strumento ikvmc.exe.

Al fine di sostituire IKVM è necessario ricercare uno strumento che allo stesso permetta la generazione automatica di codice CIL nella forma di eseguibile .exe o di assembly dll a partire da bytecode java. Il bytecode Java è ottenibile attraverso la compilazione del codice sorgente, per effettuare questo procedimento è sufficiente dotarsi di una qualunque compilatore java come javac.exe

L'obbiettivo ideale di questa ricerca è trovare uno strumento che sia in grado di tradurre tutte le librerie java attualmente supportate da IKVM, che integri anche Java 9 e che possa essere utilizzato in maniera semplice senza imposizione di vincoli. Inoltre il processo di traduzione di IKVM interviene a livello di package trasformando file.jar in file .dll, uno strumento sostitutivo compatibile con Java 9 dovrebbe invece effettuare la conversione da modulo ad assembly.

Nel caso in cui non fosse disponibile nessun strumento con lo stesso livello di copertura di IKVM è possibile valutare eventuali strumenti in grado di tradurre solo una parte delle librerie proposte da IKVM, ma che siano ancora in via di sviluppo e che propongano o promettano una qualche gestione per le versioni più recenti di Java, in questo modo sarebbe possibile valutare un approccio di traduzione ibrido.

La ricerca è stata inizialmente indirizzata verso strumenti in grado di generare codice per piattaforma .NET.

Le strade seguite sono state le seguenti:

- JA.NET[9]
- XMLVM[10]
- Eventuali forks di IKVM

JA.NET comprende un'implementazione in .NET di una Java Virtual Machine e fornisce supporto per la compilazione e l'esecuzione di codice Java su piattaforma .NET. Questo strumentofu confrontato a IKVM come strumento per supportare la generazione tuProlog.NET nella tesi "*Porting e processo di mantenimento di applicazioni Java su piattaforma .NET*" di Alessandro Montanari. Data la similitudine con IKVM in termini di funzionalità fornite se questo progetto risultasse essere aggiornato e supportato, potrebbe presentare una valida alternativa per generare un'applicazione .NET. Purtroppo però la versione più recente di questo progetto risale al 2010 e, inoltre, il sito di riferimento *janetdev.org* non è più disponibile.

XMLVM permette la traduzione a livello di bytecode tra le piattaforme Java e .NET. Il processo di traduzione avviene tramite una trasformazione intermedia in cui le istruzioni in bytecode vengono rappresentate come tag XML. Questo progetto non presenta però aggiornamenti a partire dal 2011.

Essendo IKVM un progetto open source tutto il codice sorgente è disponibile. Una fork di IKVM è un qualunque progetto implementato a partire dal codice di IKVM.

L'unica fork di IKVM individuata è stata sviluppata come parte del progetto Codename One per consentire la traduzione da Java a piattaforma UWP(vedi paragrafo 4.4.2).

Al momento non vi sono dunque strumenti che possano supportare il processo di traduzione a livello di bytecode, in quanto nessuna delle alternative valutate presenta aggiornamenti recenti.

4.3.2 Conversione codice sorgente

Una possibile alternativa potrebbe consistere nell'appoggiarsi a strumenti che permettano la traduzione di codice sorgente tra diversi linguaggi. Nel caso di questa ricerca si potrebbe ad esempio intervenire traducendo sorgenti java in sorgenti c#.

Questa soluzione porterebbe diversi vantaggi rispetto all'approccio di generazione di eseguibili, in quanto sarebbe disponibile tutto il sorgente anche della versione .NET e sarebbe possibile intervenire su esso per modificarne delle parti. Potenzialmente facendo alcune modifiche su un sorgente modificato si potrebbe ottenere una versione di tuProlog.NET più performante di quella attuale.

Come nel caso della traduzione di bytecode uno strumento per essere considerato adeguato dovrebbe:

- Garantire la traduzione di un numero soddisfacente di librerie di JavaSE
- Supportare versioni di Java superiori a Java 8.
- Essere costantemente aggiornato e supportato.

Questi sono stati gli strumenti individuati:

- Sharpen[11]
- Java Language Conversion Assistant[12]
- Java To C# Converter Tangible Software [13]

Sharpen è un progetto open source utilizzabile da linea di comando o come plug-in Eclipse che permette la conversione di codice sorgente da Java a C#, l'ultimo aggiornamento risale però al 2015.

Java Language Conversion Assistant è uno strumento per la conversione di sorgenti da Java a C# sviluppato da Microsoft e integrato con Visual Studio. Il progetto è decisamente obsoleto in quanto ha ricevuto il suo ultimo aggiornamento nel 2005 a causa della decisione di Microsoft di interrompere lo sviluppo di tecnologie destinate a favorire l'integrazione di Java su piattaforma .NET.

Java To C# Converter di Tangible Software Solutions è uno strumento disponibile in versione a pagamento o gratuita ma con limite sul numero di righe traducibili. Questo progetto però supportala maggior parte delle librerie standard di Java, il che lo rende inadeguato al caso concreto.

Dai i risultati della ricerca si evince che non è possibile sfruttare un approccio basato sulla traduzione manuale nel contesto di tuProlog. Si può, inoltre, affermare che non possono essere presi

in considerazione neanche approcci di traduzione ibridi automatico-manuale. Infatti data la scarsa quantità di librerie per le quali è supportata la traduzione automatica, la quantità di lavoro di scrittura manuale necessaria a produrre una versione completa continuerebbe ad essere insostenibile.

4.4 Bridging

Mantenere una versione .NET completa di tuProlog non è l'unica strada percorribile. Come già analizzato la perdita della piattaforma aggiuntiva può essere considerata meno rilevante rispetto al mantenimento dei casi d'interazione tra tuProlog e linguaggi .NET.

Il termine bridging indica soluzioni che permettano l'interazione tra codice Java eseguito su Java Virtual Machine e codice .NET eseguito su piattaforma .NET. Queste soluzioni solitamente offrano delle classi proxy per entrambi i linguaggi che nascondono al loro interno i livelli di presentazione e di comunicazione.

Percorrendo questa strada quindi si abbandonerebbe totalmente la versione .NET mantenendo esclusivamente quella Java. Questo approccio potrebbe in linea di principio risultare vincente in quanto darebbe luogo a un processo di sviluppo più semplice.

La versione attuale di tuProlog.NET offre la possibilità di interazione in entrambe le direzioni, ovvero utilizzare tuProlog come engine Prolog all'interno di un applicazioni .NET e classi e oggetti .NET all'interno di codice Prolog.

La prima forma di interazione è resa possibile dal fatto tuProlog è interamente disponibile in forma di libreria dll e che quindi può essere incorporato in una qualunque progetto .NET.

Il secondo caso d'uso è disponibile unicamente se si carica all'interno di tuProlog la libreria OOLibrary. Il codice di questa libreria usa un approccio basato sulla riflessione, per poter definire il tipo e i metodi degli oggetti utilizzati a tempo di esecuzione. Grazie e alla traduzione dinamica di bytecode in CIL, la libreria permette l'interazione anche con librerie scritte in Java

L'applicazione di un approccio basato su *bridging* richiede una strategia specifica per ciascuno dei due casi.

Nel primo caso, immaginando di applicare questo tipo di soluzione, tuProlog non sarebbe più disponibile come libreria .NET ma unicamente come libreria Java, sarebbe quindi necessario individuare un meccanismo che permettesse di utilizzare componenti Java all'interno di un'applicazione .NET. Al fine di non apportare modifiche potenzialmente dannose all'architettura di tuProlog, l'approccio migliore consisterebbe nel lasciare tuProlog invariato e nel concentrare la logica necessaria al *bridging* tra le piattaforme in componenti esterne.
In questo caso quindi il lavoro necessario a impostare tale soluzione, come ad esempio eventuale generazione di file o scrittura di codice aggiuntivo, sarebbe a carico dello sviluppatore dell'applicazione. Ovviamente però un approccio di questo tipo renderebbe l'integrazione di tuProlog in applicazioni .NET molto meno appetibile rispetto alla situazione attuale.

Per risolvere questo problema un approccio è quindi possibile utilizzare un Pattern Proxy, ovvero realizzare, grazie ad un supporto per il *bridging*, una classe.NET che deleghi l'esecuzione dei metodi a tuProlog in Java.
Per raggiungere questo obbiettivo, in primo luogo, sarebbe necessario definire quali sono le parti di tuProlog che andrebbero esposte ad un'applicazione che lo integrasse. Essendo interamente

disponibile come libreria tutte le classi pubbliche di tuProlog sono potenzialmente visibili esternamente, tuttavia recuperare tutta questa visibilità necessiterebbe di molto lavoro per lo sviluppo dell'interfaccia. Come si è già osservato nell'analisi dei casi d'suo però, per quanto riguarda tuProlog.NET, esporre esternamente classi strutturali non è una funzionalità interessante, infatti l'unico caso rilevante è l'utilizzo di tuProlog come motore Prolog.

La figura mostra un semplice programma d'esempio in cui tuProlog viene utilizzato come motore Prolog, all'interno di un'applicazione .NET.

```
using alice.tuprolog;
using alice.tuprolog.@event;
using java.io;


{
    class Program
    {
        static void Main(string[] args)
        {
            SolveInfo info;

            //carico il file di test
            InputStream stream = new FileInputStream("../../../prolog/test.pl");
            Theory t = new Theory(stream);
            //preparo il motore prolog
            Prolog engine = new Prolog();
            //OOLibrary.OOLibrary lib = new OOLibrary.OOLibrary();
            //engine.unloadLibrary("alice.tuprolog.lib.JavaLibrary");
            //engine.loadLibrary(lib);
            engine.setTheory(t);

            //Listener per output e spy
            engine.addOutputListener(new MyOutputListener());
            engine.addSpyListener(new MySpyListener());
            //engine.setSpy(true);

            info = engine.solve("goal");


        }
    }
```

Figura 4 – Esempio di utilizzo di tuProlog da codice in C#

Come si può osservare dal codice la classe chiave per realizzare questo comportamento è la classe Prolog. Questa classe infatti espone i metodi necessari all'utilizzo di tuProlog come interprete Prolog, in particolare il metodo `setTheory(Theory t)` che permette il caricamento di una teoria, ovvero di una porzione di codice Prolog contenente fatti e regole e il metodo `solve(String s)` con la quale viene richiesta la risoluzione di richiesta.

Al fine quindi di mantenere questo caso d'uso sarebbe quindi necessario realizzare una classe Proxy che esponesse l'interfaccia IProlog, ovvero l'interfaccia che definisce i metodi della classe Prolog.

La seguente figura mostra lo schema della soluzione.

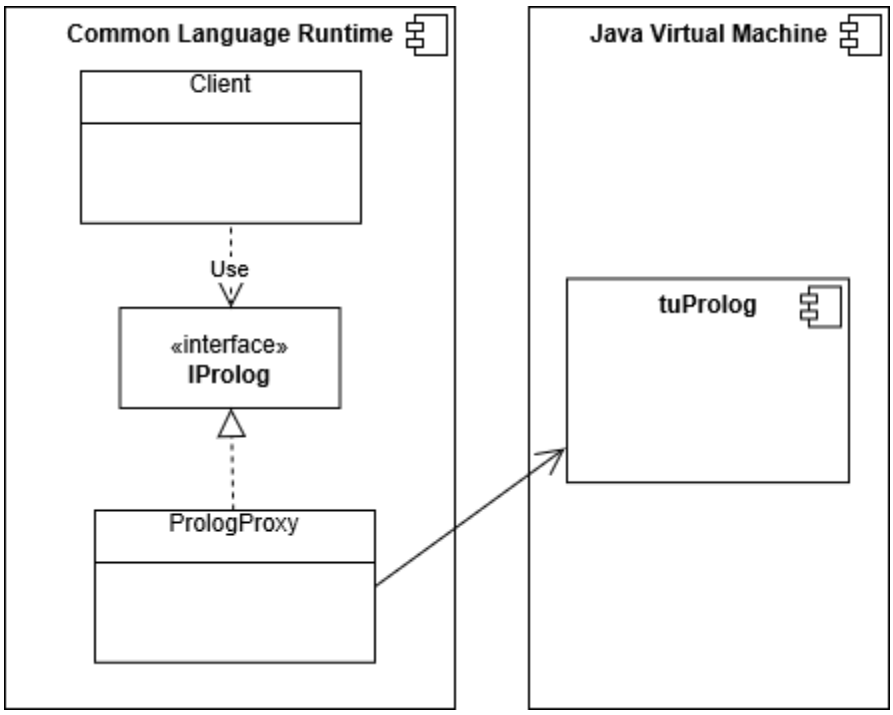

Figura 5 – Applicazione pattern Proxy per utilizzo di tuProlog Java da applicazione .NET

Ovviamente le considerazioni fatte finora non rappresentano un quadro completo della situazione, in quanto come si può osservare dalla figura x, l'applicazione non utilizza unicamente la classe Prolog ma anche la classe Theory e persino la classe InputStream della libreria java.io che grazie a IKVM ha un'implementazione in .NET. Per questi motivi un eventuale applicazione di questo approccio necessiterebbe di una reingegnerizzazione dell'interfaccia che eliminasse le dipendenze tra l'utilizzatore e classi Java a lui non disponibili.

4.4.1 OOLibrary

La libreria OOLibrary di tuProlog implementa la componente del motore Prolog che si occupa dell'interpretazione dei predicati legati all'utilizzo di oggetti da codice esterno.

Il codice mostra ad esempio l'utilizzo della classe Student, che rappresenta uno studente con gli attributi nome cognome e matricola, in una porzione di codice Prolog,

```
test :-
    load_convention('lib/Conventions.dll','Conventions.CSharpConvention',CSConv),

    new_object(CSConv,'CStudent.Student, CStudent',[796917,'matteo','castiglio'], Student),

    /* INSTANCE METHOD INVOCATION  */
    Student <- printStudent returns StudentInfo,
    write(StudentInfo),

    /* INSTANCE METHOD INVOCATION (WITH ARGUMENTS)  */
    Student <- signExam(30) returns Comment,
    write(Comment),

    /* PROPERTY GET */
    Student.id <- get(Id),
    write(Id),

    /* PROPERTY SET */
    Student.id <- set(1234),
    Student.id <- get(NewId),
    write(NewId).
```

Figura 6– Codice Prolog con predicati forniti dalla OOLibrary per utilizzo di classi C#

L'implementazione dei predicati Prolog utilizzati nell'esempio appena mostrato come ad esempio `new_object/3` o `object <- return` si trova nella libreria OOLibrary.

Nella versione attuale della libreria, il cui sorgente è presente solo in versione Java, per poter utilizzare oggetti provenienti da librerie esterne le cui classi non sono note a tempo di compilazione vengono sfruttate le funzionalità di riflessione, ad esempio nell'implementazione del predicato new per risalire alla classe dell'oggetto che deve essere istanziato viene utilizzato il metodo `Class.forName(String className)`. La versione Java della OOLibrary definisce quindi dei predicati Prolog utilizzabili unicamente con classi Java.

La libreria .NET, con l'unica eccezione delle convenzioni, viene ottenuta con l'ausilio di IKVM.
A differenza della controparte Java, però OOLibrary nella versione .NET estendo l'uso dei predicati Prolog a classi di tutti i linguaggi .NET, mantenendo inoltre grazie alla jvm integrata in IKVM la compatibilità con Java.

Decidendo di utilizzare una soluzione basato su un supporto per il bridging è necessario modificare il codice della libreria OOLibrary al fine di rendere possibile a tuProlog in Java di interagire con oggetti .NET.

Gli approcci possibili sono due. In entrambi supponiamo di abbandonare tuProlog.NET e di estendere tuProlog Java al fine di rendere disponibile con esso dei predicati Prolog che consentano l'utilizzo di classi .NET. Ovviamente per poter utilizzare questo approccio è necessario disporre a tempo di esecuzione di una piattaforma .NET.

Il primo approccio consisterebbe nell'implementare i predicati Prolog necessari per utilizzare gli oggetti.NET in una nuova libreria tuProlog, che si potrebbe chiamare OOLibraryNET. Questa libreria che dovrebbe essere implementata con una classe .NET dovrebbe diventare un vero e proprio componente esterno indipendente da tuProlog. Allo stesso tempo stesso però l'implementazione sarebbe molto simile a quella della OOLibrary attuale.

In questo caso il supporto per il *bridging* dovrebbe essere utilizzato al fine di rendere questa libreria utilizzabile da tuProlog e si tratterebbe quindi di un problema molto simile a quello discusso nel paragrafo precedente riguardo all'esposizione di tuProlog con l'interfaccia IProlog. In questo caso però rendere indipendente la OOLibrary risulta molto più complicato in quanto questa ha numerose dipendenze da altri package di tuProlog.

La figura mostra la porzione di codice della OOLibrary Java in cui vengono elencate le dipendeze.

```
import alice.tuprolog.Int;
import alice.tuprolog.Library;
import alice.tuprolog.Number;
import alice.tuprolog.Struct;
import alice.tuprolog.Term;
import alice.tuprolog.Var;
import alice.tuprolog.lib.annotations.OOLibraryEnableLambdas;
import alice.tuprolog.JavaException;
import alice.util.AbstractDynamicClassLoader;
import alice.util.AndroidDynamicClassLoader;
import alice.util.InspectionUtils;
import alice.util.JavaDynamicClassLoader;
```

Figura 7 – Parte di codice sorgente della OOLibrary

Il secondo caso invece prevede di sviluppare il codice della OOLibrary in una classe Java, e di inserire in quest'ultima la logica necessaria per il *bridging*. In questo caso quindi il problema non consisterebbe nel rendere OOLibrary una libreria .NET indipendente e nell'interfacciare tuProlog con essa, ma nel trovare uno strumento che possa fornire delle funzionalità che permettano il recupero e la definizione delle proprietà di oggetti .NET a tempo di esecuzione. In questo caso non sarebbe quindi sufficiente uno strumento in grado di generare staticamente delle classi proxy a partire da delle librerie .NET, ma sarebbe necessario avere un supporto a tempo di esecuzione per l'utilizzo di classi non note a tempo di compilazione, ovvero dovrebbero essere forniti dei metodi equivalenti a quelli utilizzati nella OOLibrary per sfruttare le proprietà di riflessione (come ad esempio Class.forName) che permettessero però l'utilizzo di oggetti .NET. Un esempio di questa applicazione verrà fornito nel paragrafo 4.4.6.

La figura mostra lo schema strutturale dei due casi presentati.

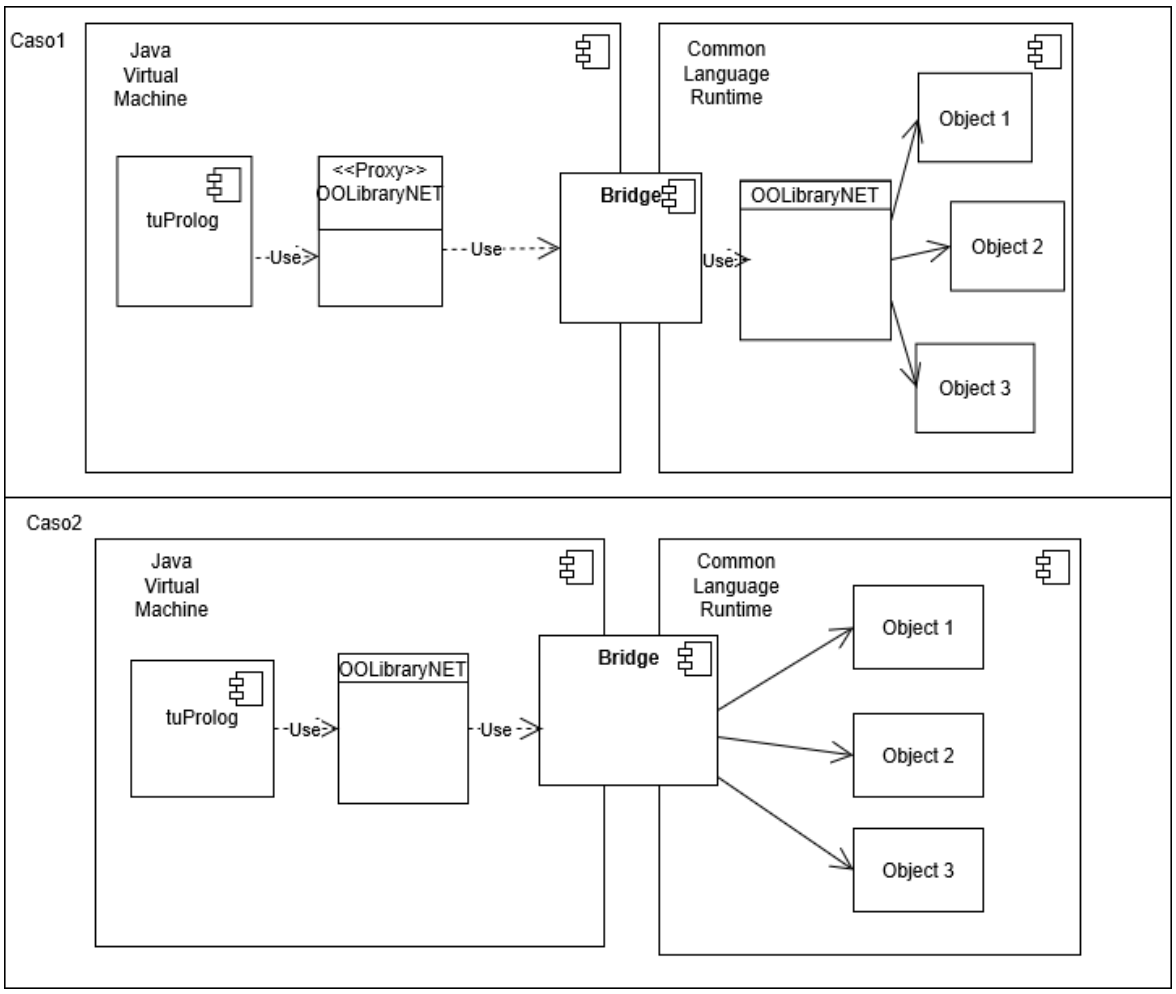

Figura 8 – Schema delle architetture possibili per lo sviluppo di OOLibraryNET

Un ulteriore differenza è che nel primo caso dovendo realizzare una nuova libreria sarebbe necessario definire dei predicati Prolog differenti per l'utilizzo di oggetti Java piuttosto che .NET, ovvero ad esempio al posto del predicato Prolog `new_object/3` dovrebbero essere definiti i predicati `java_new_object/3` e `net_new_object/3` mentre nel secondo anche mantenere lo stesso predicato per entrambi i casi. Si deve comunque considerare che definire predicati separati sviluppando una libreria OOLibraryNET apposita permetterebbe di estendere il funzionamento di tuProlog senza apportare modifiche a componenti già funzionanti.

4.4.2 Java Native Interface

La Java Native Interface[17]o JNI è un interfaccia di programmazione standard che definisce come utilizzare codice nativo all'interno di metodi java o come incapsulare una Java Virtual Machine all' interno di un'applicazione nativa. Con il termine codice nativo ci si riferisce a codice specifico per un determinato sistema operativo o per una particolare architettura, o più in generale, a codice scritto nei linguaggi di programmazione C, C++ o assembly.

Generalmente la Java Native Interface viene utilizzata per la scrittura di parti di codice che svolgono funzionalità di basso livello strettamente legate all'ambiente che non possono essere implementate direttamente in Java.

Nel caso di studio di questa tesi JNI, data la possibilità di poter utilizzare parti di codice scritte in C, C++, o C++/CLI può essere valutata come mezzo di collegamento tra Java e .NET. Inoltre gli strumenti più ad alto livello che offrono supporto all'interoperabilità fra Java e .NET sono strutturati a partire da JNI.

I metodi definiti come nativi permettono chiamate di funzioni in codice native all'interno di Java. Un metodo nativo viene definito attraverso la keyword native. Per utilizzare del codice nativo all'interno di una classe è necessario richiamare in una fase precedente il metodo System.loadLibrary(String).

Il seguente codice mostra la classe Test che definisce il metodo nativo f.

```
class Test {
     native double f(int i, String s);
     static {
     System.loadLibrary("pkg_Cls");
     }
}
```

Il nome utilizzato all'interno del metodo System.loadLibrary viene modificato in base all'architettura: ad esempio, i sistemi Solaris convertono `pkg_Cls` in `libpkg_Cls.so` mentre i sistemi con win32 in `pkg_Cls.dll`.La libreria può essere utilizzata attraverso linking statico o dinamico.

La firma dei metodi nativi all'interno della libreria utilizzata deve rispettare un determinato standard.
Il primo argomento deve essere di tipo JNIEnv, il secondo deve essere un riferimento a una classe o ad un oggetto dipendentemente dal fatto che il metodo sia statico o meno, mentre gli argomenti successivi e il valore restituito devono essere di tipi corrispondenti a quelli della firma in Java.
Sia per C che per C++ le definizioni delle strutture dato e delle funzioni sono presenti all'interno del file jni.h che viene generato, a partire da Java 8, con il comando `javac -h`;

Ad esempio a partire dal metodo nativo `double f(int i, String s)` viene generato un metodo C corrispondente con la seguente firma:

```
jdouble Java_pkg_Test_f__ILjava_lang_String_2 (
     JNIEnv *env, jobject obj, jint i, jstring s)
```

JNI inoltre definisce un'interfaccia che espone una serie di metodi che permettono di controllare esternamente una macchina virtuale Java. Questi metodi offrono la possibilità di ricavare informazioni specifiche della macchina virtuale che si sta utilizzando e di utilizzare remotamente tipi primitivi e oggetti Java.

Nonostante consenta potenzialmente di raggiungere il risultato desiderato, un approccio di soluzione così a basso a livello richiederebbe una quantità di lavoro troppo elevata.
Utilizzare JNI in sostanza, richiederebbe l'apertura di un nuovo progetto solo per portare a termine lo sviluppo di una componente di tuProlog. Tale approccio è quindi evidentemente impraticabile.

JNI,di per sé, non è quindi adeguato all'obbiettivo della tesi, tuttavia esso è alla base di diversi strumenti più ad alto livello utilizzabili direttamente in Java. Questi strumenti forniscono le funzionalità necessarie a supportare l'interazione tra le piattaforme Java e .NET senza richiedere

all’utilizzatore la scrittura di codice in linguaggi a basso livello. Nei prossimi paragrafi saranno presentati degli strumenti appartenenti a questa categoria.

4.4.4 jni4net

jni4net[18] è un progetto open source distribuito con licenza GPL basato su JNI. Lo strumento offre la possibile di utilizzare oggetti e classi.NET da Java e viceversa. jni4net è disponibile per Windows a 32 e 64 bit, tuttavia, non ricevendo aggiornamenti dal 2014, non è compatibile con le versioni più recenti di Java e .NET.

jni4net consente l’esecuzione delle macchine virtuali java e .NET all’interno dello stesso processo. La comunicazione avviene attraverso l’utilizzo di classi proxy, generate tramite lo strumento Proxygen[19]. Proxygen è un eseguibile che può esser lanciato via linea di comando che permette la generazione automatica dei proxy, sollevando l’utilizzatore dal dover utilizzare linguaggi di basso livello per implementare i meccanismi di comunicazione. Proxygen è distribuito con licenza GPL.3.0.

Essendo una soluzione di *bridging* Jni4net prevede l’esecuzione contemporanea di entrambe le piattaforme.

Le chiamate da Java a .NET sono state implementate a partire dai metodi nativi forniti da JNI.
Grazie a Proxygen vengono generati file proxy Java con metodi marcati con la keyword native.
Il generatore si occupa in oltre di effettuare le modifiche necessarie per risolvere le differenze presenti nelle convenzioni di nomi e negli elementi strutturali del linguaggio (le proprietà .NET vengono tradotte come metodi get e set). I proxy implementano il finalizzatore di java in questo modo quando il garbage collector elimina l’istanza della classe proxy viene eliminata anche la rispettiva istanza reale .NET.

Le chiamate da .NET a Java sono implementate utilizzando l’interfaccia JNI che espone i metodi che permettono di controllare la JVM. I metodi esposti da questa interfaccia permettono di controllare gli oggetti gestendone instanziazione e ciclo di vita. Questa interfaccia che viene comunemente in esposta in C++ nel file header `jni.h` è stata tradotta in C# per poter essere utilizzata direttamente da codice .NET. Proxygen permette la generazione di proxy che nascondo l’utilizzo dell’interfaccia JNI.

La figura, proveniente dalla documentazione dal sito web di Jni4net, mostra l’architettura su cui è basato il funzionamento di jni4net. Come si può osservare il funzionamento di JNI4net è basato sull’utilizzo di classi proxy che possono poi essere utilizzate in maniera semitrasparente.

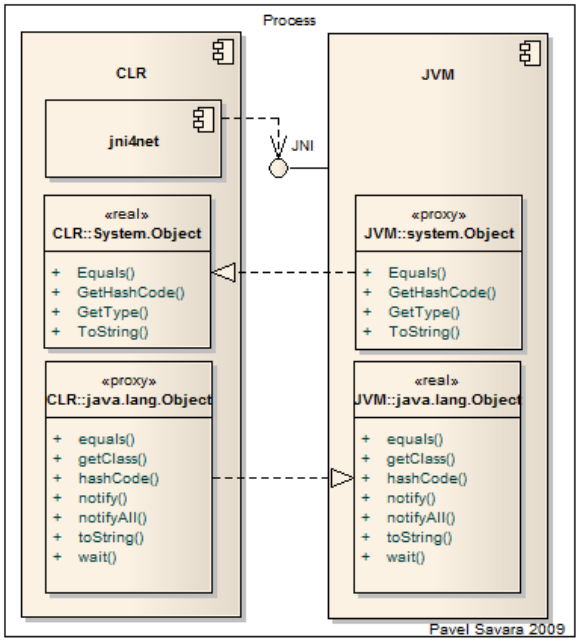

Figura 9 - Schema architetturale di jni4net

Esempio di applicazione basata su jni4net

La figura seguente mostra una classe Test realizzata in C#

```
public class Test
{
    public void Hello()
    {
        Console.WriteLine("Hello Java, from .NET!");
    }

    public void Repeat(string line)
    {
        Console.WriteLine(line);
    }

}
```

Figura 10 – codice classe C# di esempio

Per utilizzare jni4net come prima cosa è necessario utilizzare Proxygen per generare i proxy java a partire da una libreria dll. La libreria `testlib.dll` contente la classe Test mostrata precedentemente.

Con l'ausilio di Proxigen è possibile generate a partire dalla libreria i file `testlib.j4n.dll` e `testlib.j4n.jar`.

Per poter utilizzare jni4net è infine necessario importare la libreria originale e i file generati all'interno di un progetto Java.

Il seguente codice mostra un esempio di applicazione Java che utilizza Jni4net per accedere alla classe Test all'interno della libreria testlib.dll.

```
import java.io.File;
import java.io.IOException;
import testlib.Test;
import net.sf.jni4net.Bridge;


public class testJNI4net {

    public static void main()
    {
            try {

                Bridge.setVerbose(true);
                Bridge.init();

                Bridge.LoadAndRegisterAssemblyFrom(new File("testlib.j4n.dll"));

            } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }

            Test test = new Test();
            test.Hello();
            test.Repeat("It works!");

    }
}
```

Figura 11 – Codice applicazione esempio basata su jni4net

Test per utilizzo di tuProlog Java da applicazione C# con jni4net

jni4net è potenzialmente compatibile con sistemi operativi Windows 7 o superiore, Java 5 o superiore e con CLR 2.0 e 4.0. Determinare l'applicabilità di jni4net per il caso d'interesse della tesi non è semplice in quanto la documentazione presente è incompleta abbastanza confusa e non è chiaro quali siano i limiti di questo strumento. Inoltre è necessario considerare che jni4net non riceve aggiornamenti a partire dal 2014.

Per valutare l'applicabilità dello strumento si è quindi cercato di utilizzare jni4net per rendere tuProlog Java utilizzabile da un'applicazione .NET.

I test sono stati svolti su Windows 7 a 64 bit con jdk 1.8 a 32 bit e CLR 4.0 a 32 bit.
Si è quindi tentato di generare i proxy a partire da *tuProlog.jar*, utilizzando proxygen con il seguente il comando, indicando con l'opzione -cp le dipendenze del progetto.

```
C:/mypath/proxygen.exe C:/mypath/tuProlog.jar
-wd C:/mypath/ -cp C:/mypath/gson-2.6.2
```

In seguito all'esecuzione del comando sono stati generati automaticamente i file sorgenti delle classi proxy e l'eseguibile `build.cmd`.

Eseguendo `build.cmd` sono stati poi generati i file `tuprolog.j4n.jar` e `tuprolog.j4n.dll`. In seguito è stato creato un nuovo progetto Visual Studio al quale sono stati aggiunti i riferimenti a `tuprolog.j4n.dll e jni4net.n-0.8.8.0.dll`.

Infine sono stati aggiunti i seguenti file alla file nelle cartella del eseguibile .exe:

```
tuprolog.jar
gson-2.6.2.jar
tuprolog.j4n.dll
tuprolog.j4n.jar
jni4net.n-0.8.8.0.dll
jni4net.n.w32.v40-0.8.8.0.dll
```

l'applicazione di test, rappresentata dalla seguente figura, funziona correttamente.

```
using alice.tuprolog;
using alice.tuprolog.@event;
using net.sf.jni4net;

namespace ConsoleApp1
{
    class Program
    {
        static void Main(string[] args)
        {
            var bridgeSetup = new BridgeSetup();
            bridgeSetup.AddAllJarsClassPath(".");
            Bridge.CreateJVM(bridgeSetup);
            Bridge.RegisterAssembly(typeof(alice.tuprolog.Theory).Assembly);
            alice.tuprolog.SolveInfo info;
            java.io.InputStream stream = new java.io.FileInputStream("Factorial.pl");
            Theory t = new Theory(stream);
            Prolog engine = new Prolog();
            engine.setTheory(t);
            engine.addOutputListener(new MyOutputListener());
            engine.addSpyListener(new MySpyListener());
            info = engine.solve("fact(4,X).");
            System.Console.WriteLine(info.toString());
            System.Console.ReadLine();
        }
    }

    public class MySpyListener : SpyListener
    {
        public void onSpy(SpyEvent se)
        {            System.Console.WriteLine(se.toString());        }
    }
    public class MyOutputListener : OutputListener
    {
        public void onOutput(OutputEvent e)
        {            string message = e.getMsg();
            if (!(message.StartsWith("'board") || message.StartsWith("Board")))
                System.Console.Write(e.getMsg());        }
    }
`
```

Figura 12 – Esempio di applicazione C# che utilizza tuProlog Java con jni4net

Per raggiungere questo risultato è stato però necessario attuare alcuni accorgimenti:

- Sono stati rimossi da tuProlog.jar le componenti che dipendevano da IKVM, e che erano significative solo in prospettiva di un'esecuzione dell'applicazione sulla macchina virtuale di IKVM. Questo accorgimento ha quindi permesso di eliminare la dipendenza di tuProlog.jar da ikvm-api.jar.
- Sono stati rinominati tutti i metodi che presentavano il carattere $ che faceva sì che i file sorgente C# generati automaticamente non potessero essere compilati (3 metodi nella classe BasicLibrary).
- E' stato necessario modificare il file build.cmd automaticamente generato aggiungendo un carattere ad il comando jar.

  originale

  ```
  jar – cvf  tuprolog.j4n.jar –C target/classes
  alice/tuprolog/AbstractSocket_.class" …–C target/classes
  "alice/util/JavaDinamicClassLoader_.class"
  ```

  modificato

  ```
  jar – cvf  tuprolog.j4n.jar –C target/classes
  alice/tuprolog/AbstractSocket_.class" …–C target/classess
  "alice/util/JavaDinamicClassLoader_.class"
  ```

  Questa modifica è stata necessaria in quanto nonostante il file generato fosse corretto durante l'esecuzione di build.cmd si otteneva un errore con riferimento alla directory target/classe, che evidenziava il fatto che il comando "perdesse" un carattere.
- E' stato rinominato il package tuprolog.util.proxyGenerator in tuprolog.util.proxygenerator. La presenza del carattere maiuscolo rendeva le classi contenute nel package inaccessibili durante l'esecuzione.

Un secondo test è stato svolto su Windows 7 a 64 bit con jdk 10 a 64 bit e CLR 4.0 a 64 bit. In questo caso tuprolog.jar, modificato analogamente al test precedente è stato compilato con target jre 10. In seguito sono stati ripetuti i passaggi effettuati nel test precedente e si è tentato di eseguire la medesima applicazione. Differentemente dal test precedente l'esecuzione non è andata a buon termine a causa di un eccezione System.Exception.TargetException sollevata in seguito alla chiamata al metodo Bridge.RegisterAssembly().

Nonostante siano richiesti test più approfonditi per mettere in luce quelli che sono i reali limiti di questo strumento, in base al risultato di questi primi test e al fatto che non sia più mantenuto dal 2014, si può concludere che una soluzione basata su jni4net non può essere considerata sostenibile.

4.4.5 Javonet

Javonet[20] è un altro progetto che propone una soluzione di bridging tra gli ambienti Java e .NET. Javonet prevede l'esecuzione all'interno di un unico processo di sistema di una Java Virtual Machine e di Common Runtime Environment .NET garantendo la traduzione delle chiamate a metodi tra le parti e offrendo in maniera trasparente funzionalità di conversione di tipi di dato, gestione della memoria e sincronizzazione dei thread. L'approccio basato su un unico processo rende questa soluzione più performante e sicura di alternative basate su comunicazioni via servizi web. Javonet è un progetto commerciale ed è distribuito con diversi piani che prevedono un costo mensile fisso, è però possibile ottenere una licenza accademica compilando una richiesta formale.

Javonet richiede come prerequisiti la presenza di un JDK di versione 1.6 (o superiore) e di visual c++ Runtime 2013, inoltre, per poter utilizzare Javonet è necessario aggiungere il riferimento al file `javonet.jar` nel build path del progetto.

L'attivazione di Javonet è un punto critico dell'utilizzo di questo strumento. Questa può attraverso la chiamata al metodo`Javonet.activate(email,license-Key,javonet.framework)`, questo metodo durante la sua prima chiamata contatta i server e genera un file `javonet.lic` nella directory della applicazioni. Le chiamate successive di questo metodo verificano il file e non contattano i server. Alternativamente è possibile specificare i parametri nel file `javonet.xml`, in questo modo il metodo `Javonet.activate()` potrà essere utilizzato senza la necessità di scrivere i parametri direttamente nel codice sorgente. L'utilizzo del file xml facilita quindi la distribuzione di un'applicazione e l'aggiornamento delle licenze.

API di Javonet

Per poter referenziare librerie .NET è necessario utilizzare i metodi `Javonet.addReference(lib, class)` o `Javonet.addReference(lib)` dove lib può essere il nome di una libreria standard o di una dll.In alternativa è possibile specificare le librerie .NET che si vogliono utilizzare all'interno del sopracitato file javonet.xml che deve essere post nella radice del progetto.

Le API di Javonet sono sviluppate per permettere l'utilizzo di classi e oggetti .NET senza che questi siano noti in fase di compilazione. Ovvero per utilizzare classi e oggetti .NET da Java non è necessario generare dei Proxy staticamente perché vengono sfruttate le classi `NType` e `NObject` che rappresentano rispettivamente classi e oggetti .NET il cui comportamento viene definito a tempo di esecuzione.

Per utilizzare una classe .NET, è necessario creare un variabile di tipo `NType` e assegnarli il valore restituito dal metodo `Javonet.getType(className)`.Per utilizzare metodi statici invece si utilizza il metodo `NType.invoke(methodName,args[])`.

Ad esempio il codice seguente codice mostra in che modo è possibile utilizzare il metodo Show(String s) della classe System.Windows.Forms.MessageBox.

Codice C#	Codice Java con Javonet
`Messabox.Show("Hello!");`	`NType msgbox =Javonet.getType("MessageBox");` `msgbox.invoke("Show","Hello!");`

Per istanziare un oggetto è disponibile il metodo `Javonet.New()`, l'oggetto istanziato viene assegnato ad una variabile di tipo `NObject`. Per utilizzare i metodi di un oggetto si utillizza il metodo `NObject.invoke(methodName,args[])`.Con il metodo invoke è possibile chiamare un qualunque metodo di un oggetto o un classe .NET con un qualunque numero di argomenti. Gli argomenti di tipo valore vengono automaticamente trasformati nel corrispettivo .NET.E' inoltre possibile passare istanze di `NType` o `NObject` come tipi riferimento.

Il seguente codice mostra come utilizzare un oggetto della classe System.Random e come invocare il metodo System.Random.Next().

Codice C#	Codice Java con Javonet
`Random r = new Random();` `int value = r.Next(10,20);`	`NObject r = Javonet.New("System.Random");` `Integer value = r.invoke("Next",10,20);`

Per poter utilizzare un metodo generico deve essere specificato il tipo da utilizzare per l'esecuzione del metodo, per farlo si devono utilizzare istanze della classe `NType`. Per Utilizzare oggetti generici invece è possibile utilizzare il metodo Javonet.getType() specificando come primo argomento la classe seguita da un apostrofo e dal numero tipi generici da definire, e come argomenti successivi i tipi che si vogliono utilizzare. Il seguente codice mostrain che modo può essere utilizzato un oggetto di classe Dictionary<String,List<String>>. L'esempio mostra che per utilizzare di una variabile di tipo Dictionary<K,T> sia necessario utilizzare prima il metodo Javonet.getType() per definire i tipi K e T e poi il metodo NType.Create() per l'instanziazione.

Codice C#

```
Dictionaryd = new Dictionary<String,<List<String>> ();
List<String> newList = new List<String>();
newList.Add("a"); newList.Add("b");
newDict.Add("List1",newList);
```

Codice Java con Javonet

```
NType typeList = Javonet.getType("List`1","String");
NType typeString = Javonet.getType("String");
NType type = Javonet.getType("Dictionary`2",typeString,typeList);
NObject newList = typeList.create();
newList.invoke("Add","a");
newList.invoke("Add","b");
newDict.invoke("Add","List1",newList);
```

Per poter accedere alle proprietà di un classe o di una struttura si utilizzano i metodi `get(fieldOrPropertyName)` e `set(fieldOrPropertyName, newValue)`.

Il codice d'esempio mostra in che modo è possibile accedere al valore della proprietà Now della classe Datetime.

Codice C#	Codice Java con Javonet

`Datetime d = Datetime.Now`	`NType dateTimeClass = Javonet.getType("DateTime");`
	
	`NObject nowDateObj= dateTimeClass.get("Now");`

E' presente supporto all'utilizzo delle keyword`in`, `ref` e `out` per il passaggio pe riferimento di argomenti a metodi: Il comportamento della keyword `in` è quello ottenuto di default mentre per `ref` e `out` sono disponibili i tipi dedicati `NRef` e `NOut`, queste classi devono essere utilizzate come wrapper delle classe `NObject`.

Javonet permette l'utilizzo di array di tipi valore o di tipo riferimento. Gli array con elemento di tipo valore vengo convertiti in array del tipo valore Java corrispondete, mentre gli array con elementi di tipo riferimento vengono convertiti in array di `NObject`.

L'approccio che che Javonet propone per utilizzare elementi Java da codice .NET è simmetrico rispetto a quello appena visto ed è basato sulle classi `JType` e `JObject`, tuttavia la manualistica presente è molto scarsa è il prodotto non è ancora disponibile.

A differenza di JNI4net Javonet è un progetto attivo che riceve periodici aggiornamenti.
Inoltre lo stile di Javonet ispirato all'implementazione della riflessione in Java lo rende un valido candidato per una riscrittura della libreria OOLibrary della versione Java. Seguendo questo approccio sarebbe quindi possibile recuperare il caso d'uso di utilizzo di oggetti e classi appartenenti a librerie .NET da codice Prolog. Dato il mancato supporto all'interazione nella direzione inversa non sarebbe possibile invece recuperare il caso d'uso di tuProlog come motore Prolog per applicazioni .NET, tuttavia il progetto Javonet risulta essere in evoluzione e questa mancanza potrebbe essere risolta nelle release future.

Le problematiche riguardanti l'utilizzo di Javonet sono invece legate alla gestione della licenza.
La licenza accademica di Javonet non è adeguata ad un progetto open source. Facendo richiesta per una licenza accademica è possibile ottenere un numero variabile di licenze in base al numero di persone che lavorano allo sviluppo di un progetto. Tuttavia, dato che è necessario possedere una licenza per ogni macchina su cui viene utilizzata un'applicazione che utilizza Javonet non sarebbe possibile distribuire tuProlog liberamente in quanto ogni utilizzatore di tuProlog dovrebbe disporre di una propria licenza. Inoltre utilizzando la licenza accademica di Javonet non sarebbe possibile integrare tuProlog in nessuna applicazione commerciale.

Javonet, in termini di funzionalità, è quindi potenzialmente applicabile, tuttavia data la non compatibilità della licenza con la distribuzione di tuProlog non risulta essere adeguato.

Eventuale Introduzione di Javonet nella libreria OOLibrary

Osservando nel dettaglio l'implementazione della libreria OOLibrary si può notare che i riferimenti agli oggetti che devono essere utilizzati da codice Prolog sono inseriti in una `Collection` oggetti `Object` denominata currentObjects. In un'implementazione con l'ausilio di Javonet sarebbe possibile affiancare a questa struttura ad un'altra Collection di `NObject`. Attraverso questa struttura sarebbe quindi possibile gestire direttamente da Java oggetti .NET.

```
/**
 * java objects referenced by prolog terms (keys)
 */
private HashMap<String,Object> currentObjects = new HashMap<String, Object>();
/**
     * inverse map useful for implementation issue
     */
private IdentityHashMap<Object,Struct> currentObjects_inverse = new IdentityHashMap<Object, Struct>();

private HashMap<String,Object> staticObjects = new HashMap<String, Object>();
private IdentityHashMap<Object,Struct> staticObjects_inverse = new IdentityHashMap<Object, Struct>();
```

Figura 13 – Parte di codice della OOLibrary

Prendendo come esempio un frammento di codice del metodo `new_object_3` che implementa il predicato `new_object/3` si può notare sono utilizzati i metodi Class.forName che consente di definire la Classe di un oggetto a tempo di esecuzione a partire dal metodo della classe e il metodo Constructor.newInstance per l'instanziazione del suddetto oggetto. Utilizzando Javonet sarebbe possibile sostituire questi due metodi rispettivamente con `Javonet.getType(typename)` e `NType.Create()`.

```
Class<?> cl = Class.forName(clName, true, dynamicLoader);
Object[] args_value = args.getValues();
Constructor<?> co = lookupConstructor(cl, args.getTypes(),args_value);
if (co == null) {
    getEngine().warn("Constructor not found: class " + clName);
    throw new JavaException(new NoSuchMethodException(
            "Constructor not found: class " + clName));
}

Object obj = co.newInstance(args_value);
if (bindDynamicObject(id, obj))
    return true;
```

Figura 14 – Parte di codice della OOLibrary

Nel metodo java_call_3 che implementa il predicato per l'invocazione dei metodi vengono utilizzati l'oggetto Method e il metodo Method.invoke. Utilizzando Javonet è possibile sostituire questo metodo con il metodo NObject.Invoke().

Obj è il riferimento all'oggetto contenuto nella struttura currentObjects sul quale si vuole utilizzare il metodo di nome methodName

```
if (obj != null) {
    Class<?> cl = obj.getClass();
    Object[] args_values = args.getValues();
    Method m = lookupMethod(cl, methodName, args.getTypes(),args_values);
    if (m != null) {
        try {
            m.setAccessible(true);
            res = m.invoke(obj, args_values);
```

Figura 15 – Parte di codice della OOLibrary

Questi sono solo alcuni esempi che dimostrano quale sarebbe l'idea alla base della introduzione di Javonet nella libreria OOLibrary: ovviamente una reale implementazione avrebbe un livello di complessità molto superiore.

Il modo in cui Javonet potrebbe essere integrato nella libreria OOLibrary dipenderebbe da quali predicati Prolog dovrebbero essere implementati. Nel caso in cui si volesse rimanere coerenti con i predicati forniti dalla OOLibrary .NET, ovvero predicati universali sia per librerie Java che .NET, il codice Javonet andrebbe inserito all'interno dell'implementazione della OOLibrary attuale.
I metodi attuali andrebbero quindi ridefiniti per integrare le convenzioni, e nell'implementazione si dovrebbe gestire sia la logica di utilizzo degli oggetti Java, ovvero quella già presente, sia la logica di utilizzo degli oggetti .NET, che verrebbe resa disponibile dalle API di Javonet. Come già discusso questo approccio sarebbe discutibile perché comporterebbe la modifica del comportamento di parti di tuProlog già solide. Un utilizzo più coscienzioso di Javonet prevederebbe invece la definizione di una nuova libreria OOLibraryNET che fornisse dei predicati aggiuntivi specifichi per l'interazione di Prolog con classi .NET. In questo modo inoltre sarebbe possibile mantenere tuProlog indipendente da Javonet tranne per quest'unica componente.

4.5 tuProlog come Script Engine

tuProlog supporta la specifica JSR-223.La specifica JSR-223 definisce l'interfaccia standard Java Scripting API[22] per l'implementazione di scriptEngine, ovvero componenti Java in grado di eseguire programmi in un determinato linguaggio di scripting.

Un'applicazione che supporta questa specifica può essere utilizzata esternamente attraverso i metodi dell'interfaccia standard, nascondendo così la sua reale implementazione.

In pratica è possibile utilizzare tuProlog all'interno di un'applicazione Java utilizzando le interfacce ScriptEngineFactory e ScriptEngine. La prima è utilizzata nel meccanismo di discovery per ricavare istanze di ScriptEngine per il linguaggio considerato mentre la seconda espone i veri e propri metodi per interpretazione e l'esecuzione degli script.

Questo caso d'uso è disponibile unicamente all'interno della versione Java, tuttavia è possibile analizzare le funzionalità che offre .NET a supporto degli script engine e valutare la possibilità di utilizzare tuProlog in questa modalità anche da codice .NET. Questo approccio permetterebbe di recuperare in maniera differente il caso d'suo che prevede l'integrazione di tuProlog come motore Prolog all'interno di applicazioni .NET.

La presenza di un equivalente alla specifica JSR-223 in .NET, ovvero di un'0interfaccia standard che supporti l'utilizzo di engine script semplificherebbe l'utilizzo di soluzioni di Bridging. Infatti in questo caso sarebbe possibile servendosi di strumenti simili a quelli trattati nel capitolo precedente fornire lato .NET un'implementazione di tale interfaccia che nasconderebbe i meccanismi di comunicazione tra i due linguaggi. Purtroppo però .NET non propone nulla di simile.

Microsoft offre la possibilità di implementare engine script attraverso l'interfaccia COM IActiveScript, che però non risulta essere adatta in quanto si tratta di una tecnologia ormai obsoleta e che non ha compatibilità con la piattaforma Java.

4.6 Piattaforma UWP

Una ulteriore possibilità per recuperare le funzionalità di interazione con i linguaggi .NET può essere quella di considerare come possibile piattaforma target la Universal Windows Platform(UWP) di Windows 10.

4.6.1 Piattaforma UWP

Microsoft ha introdotto con Windows 10 la piattaforma UWP(Universal Windows Platform)[14] ovvero una piattaforma per app comune per tutti i dispositivi che eseguono Windows 10. UWP offre le stesse API di base per tutti i dispositivi Windows. Se un app è sviluppata usando solo le API di base può essere utilizzata su un qualsiasi dispositivo Windows 10, indipendentemente che sia destinata a un PC Desktop, Xbox o smartphone.

Le app UWP possono essere sviluppate utilizzando Visual Studio, creando un progetto della categoria Windows Universal, i linguaggi utilizzabili sono C#, C++, VB e Javascript. Ogni app ha accesso alle API Win32 specifiche per la UWP, che sono implementate in tutti i dispositivi Windows 10 e che quindi ne garantiscono la portabilità. Nel caso in cui, invece, un app sia destinata ad un dispositivo specifico, questa può usufruire delle API specifiche del dispositivo attraverso gli SDK di estensione. Per risolvere le problematiche relativa alle le differenze in termini di caratteristiche dei dispositivi e dimensioni dello schermo, per lo sviluppo e la progettazione dell'interfaccia utenteè possibile utilizzare controlli universali e pannelli di layout in grado di adattarsi alla densità DPI e alle dimesioni dello schermo.Dopo essere stata sviluppate le app UWP possono essere distribuite attraverso il Microsoft Store, che fornisce un sistema di gestione dei pacchetti che ne semplifica l'installazione e la disinstallazione.

Per poter utilizzare la piattaforma UWP nel contesto di tuProlog è necessario individuare un processo di *porting*, di fatto rimettendo in gioco tutte le alternative discusse in precedenza come traduzione manuale automatica.
Dal momento che le osservazioni fatte riguardo allo sviluppo manuale rimangono valide indipendentemente, diventa quindi necessario trovare uno strumento che supporti un livello di traduzione.

4.6 2 Codename One

Codename One[15] permette a sviluppatori Java di scrivere app destinate a tutti i dispositivi mobili (iOS, Android, Windows etc.). E' quindi una alternativa che può essere presa in considerazione per supportare il porting di tuProlog su piattaforma UWP.

Codename One è un progetto open source nato del 2006 da Sun MicroSystem ed è disponibile come plug-in per i diversi IDE per Java.

L'operazione di building verso le diverse architetture è resa disponibile come servizio cloud in versione gratuita o a pagamento. Nella versione gratuita vi sono limiti nel numero di build eseguibili in un mese espressi attraverso un sistema a crediti. Ogni mese vengono forniti 100 crediti, le build verso Android e UWP costano un credito mentre quelle verso IOS ne costano 8. Inoltre è presente un limite sulla dimensione del pacchetto jar, che deve essere inferiore a 1 MB (il calcolo non comprende le librerie di Codename ONE). La versione gratuita permette l'utilizzo commerciale delle app generate.

Il processo di traduzione da bytecode Java a applicazione UWP è disponibile grazie una versione modificata di IKVM resa disponibile in maniera opensource.

Essendo però destinato a dispositivi mobile Codename One non supporta totalmente Java SE. Le motivazioni portate dal team di sviluppo sono che supportare completamente la JVM porterebbe ad un aumento delle dimensioni della distribuzione, eliminerebbe la portabilità completa e impatterebbe negativamente sulla performance. Per questo motivo non sono supportate librerie come java.io o java.NET difficilmente traducibili all'interno di un sistema operativo per dispositivi mobili o java.NET le cui funzionalità non sono significative all'interno di un app. Inoltre non è supportata la Riflessione essendo strettamente legata al bytecode. Anche le librerie grafiche come swing o FX non sono supportate in quanto sostituite da altre librerie. Al fine di non introdurre nel progetto parti di codice non traducibili, non è neanche possibile integrare nel progetto file jar arbitrari. Per quanto riguarda futuri aggiornamenti il team di sviluppo ha come obbiettivo il supporto completo a Java ME[21], ovvero la piattaforma Java destinata a dispositivi mobili e IOT.

Nonostante sia un progetto attivo Codename One non supporta e non prevede di supportare Java 9, essendo le novità introdotte da questa nuova versione, in particolare i moduli, considerate non interessanti per lo sviluppo di app per dispositivi mobili.

Codename One offre un processo di traduzione compatibile con le modalità di sviluppo di tuProlog, permettendo la generazione di applicazioni per piattaforme target differenti a partire dal solo codice Java. Anche i limiti sulle traduzioni imposti della versione gratuita non risultano troppo opprimenti, in quanto i limite massimo di 1Mb per la dimensione del file jar è nettamente superiore alla dimensione attuale di tuProlog ed è possibile fare utilizzo commerciale dell'applicazione generata (tuProlog è utilizzato con licenza LGPL all'interno di applicazioni commerciali).Inoltre se si considera di voler generare esclusivamente l'applicazione destinata alla piattaforma UWP il limite mensile di 100 build può essere considerato sufficientemente alto.
Non vi è invece compatibilità tra Codeame One e tuProlog per quanto riguarda il supporto alle librerie Java, tuProlog, infatti, fa uso di classi appartenenti a librerie che non potrebbero essere tradotte se si decidesse di utilizzare Codename One, come ad esempio Java.net.Socket e java.io.File. Inoltre non è previsto nessun tipo di supporto per Java 9 principale elemento di interesse per la ricerca che si sta effettuando. Paragonando allo stato attuale IKVM e Codename One, non aggiunge nulla in termini di librerie supportate.

Se si volesse ottenere una versione tuProlog UWP attraverso Codename One sarebbe necessario rinunciare a numerose funzionalità ed in particolare a quelle legate alla programmazione multiparadigma. Tale versione quindi non sarebbe adatta a sostituire l'attuale versione .NET.

L'utilizzo di Codename One potrebbe invece essere utile per lo sviluppo di un eventuale tuProlog per dispositivi mobile che potrebbe essere reso disponibile facilmente per diverse piattaforme, tale caso però non rientra nell'obbiettivo di questa tesi.

4.7 Tabelle comparative strumenti

Tabella A - Supporto allo sviluppo tuProlog.NET

Approccio	Strumento	Problematiche	Applicabile
Mantenimento IKVM	IKVM	- Difficilmente standardizzabile - Non sostenibile a lungo terrine - Richiede sviluppo manuale di parti di codice	**No***
Traduzione automatica di codice sorgente	Sharpen	- Obsoleto	**No**
	Java to C#(tangible solution)	- Non traduce la maggior parte delle librerie standard di Java - Non open source	
Traduzione automatica di bytecode	XMLVM	- Mancanza del supporto a Java 9 - Qualità inferiore a IKVM - Obsoleto	**No**
	JANET	- Mancanza del supporto Java 9 - Qualità inferiore a IKVM - Obsoleto	

Tabella B - Recupero casi d'uso con approcci alternativi

Approccio	Strumento	Finalità	Problematiche	Aspetti positivi	Applicabile
Bridging	jni4net	Utilizzo di tuProlog Java all'interno di applicazioni .NET.	- Non attivo - Documentazione disordinata - Presenza di bug - Supporto limitato a Windows - Risultato negativo dei test con Java 9	- Open source - Risultato positivo dei test	**No**
	Javonet	Estensione di tuProlog Java per introdurre supporto la programmazione multilinguaggio tra Prolog-.NET	- Licenza a pagamento con costo non sostenibile - Licenza accademia non compatibile con distribuzione di tuProlog	- Attivo e supportato - Risultato positivo dei test - Compatible la struttura della OOLibrary	**No***
tuProlog come Script Engine	Nessuno	Utilizzo di tuProlog Java all'interno di applicazioni .NET.	- Non ci sono tecnologie che supportano quest'approccio.		**No**
Porting a UWP	Codename ONE	Sviluppo di tuProlog per piattaforma UWP e supporto alla programmazione mutilinguaggio tra Porlog-.NET	- Non traduce alcune librerie standard di Java SE.	- Attivo - Licenza gratuita - Utilizzabile per lo sviluppo di applicazioni per dispositivi mobili	**No**

5 Discussione

Nel capitolo precedente è stata valutata singolarmente l'applicabilità di ciascuno degli approcci proposti, è quindi opportuno discutere in maniera più generale i risultati ottenuti al fine di rispondere alle seguenti domande:

- E' possibile sostenere lo sviluppo tuProlog.NET?
- E' possibile recuperare alcuni dei casi d'suo specifici di tuProlog.NET con approcci alternativi?

5.1 Sviluppo di tuProlog.NET

Al fine di valutare la sostenibilità dello sviluppo di tuProlog.NET sono stati valutati diversi processi di traduzione che potessero sostituire quello fornito da IKVM. In sintesi, in seguito alle ricerche effettuate si è concluso che non è possibile sostenere un processo di traduzione manuale, a causa della complessità del progetto, e che non sono disponibili strumenti in grado di supportare un processo di traduzione automatica. La conclusione che si può quindi trarre è che l'unico strumento su cui è possibile fare riferimento per lo sviluppo di tuProlog.Net è IKVM.

Nel capitolo dedicato all'approccio di soluzione che prevedeva il mantenimento di IKVM come sostegno alla traduzione sono state evidenziate diverse problematiche relative principalmente allo sviluppo a lungo termine di tuProlog.NET. Tuttavia, avendo ora i risultati complessivi della ricerca, è opportuno valutare se proseguire lo sviluppo di tuProlog.Net per qualche altra versione o se cessare il supporto. Come già discusso nel capitolo sopra citato per rispondere a questa questione sarebbe necessario avere una prospettiva completa delle modifiche che verranno introdotte nella nuova versione di tuProlog. Rimangono però valide le seguenti osservazioni:

- Continuare lo sviluppo di tuProlog.Net basandosi su IKVM necessita di un lavoro di scrittura manuale per integrare le parti di codice non traducibili automaticamente
- IKVM non è compatibile né con Java 9 né con i moduli è quindi necessario inserire nel processo di generazione di tuProlog.Net un passaggio intermedio che permetta di compilare un progetto con struttura modulare con target Java 8.

Sostenere lo sviluppo di tuProlog.Net, per qualche altra versione, significa quindi introdurre degli accorgimenti che permettano di sopperire alle mancanze di IKVM, rimane quindi da valutare se valga la pena o meno investire tempo e lavoro.

Trattandosi però di una soluzione che non permette di risolvere il problema in maniera definitiva, la conclusione a cui di giunge è che seppur rimanga da valutare quale potrà essere l'ultima versione rilasciata lo sviluppo di tuProlog.NET non è sostenibile e dovrà quindi essere interrotto.

5.2 Approcci alternativi

Per ampliare il campo di ricerca sono valutati diversi approcci alternativi che permettessero il recupero dei seguenti casi d'uso.

- Uso di tuProlog Java come motore Prolog all'interno di un'applicazione .NET

- Estensione di tuProlog Java per introdurre supporto alla programmazione multiparadigma Prolog-.NET

Il capitolo precedente è esaustivo per quanto riguarda l'analisi degli approcci tuProlog come Script Engine e porting di tuProlog su piattaforma UWP, è infatti già stato motivato il perché questi approcci siano impraticabili. Dalla analisi delle soluzioni basate su *bridging* sono invece emersi alcuni aspetti che necessitano di approfondimento.

In base ai test preliminari effettuati jni4net non sembra essere una soluzione adeguata per implementare il primo caso d'uso, tuttavia, a causa della documentazione disordinata, risulta complicato valutare a priori quali possono essere i limiti di questa soluzione. E' quindi possibile effettuare dei test ulteriori al fine di ottenere una valutazione più completa, in ogni caso è opportuno considerare che dato che jni4net non riceve aggiornamenti da diversi anni, questo approccio non sarà sostenibile a lungo termine.

Per completezza è però necessario osservare per implementare questo caso d'uso non è necessario appoggiarsi a un supporto per il Briding, sarebbe infatti possibile utilizzare tuProlog Java da applicazione .NET utilizzando delle architetture distribuite. Sarebbe ad esempio possibile fornire i servizi di motore Prolog implementando un server Web o un server TCP basato su Socket, inoltre per entrambi gli approcci possono essere realizzati con riferimento all' architettura basata su Proxy proposta in *figura 5*. E' possibile osservare un'implementazione simile nella tesi "*Windows 10 IoT su Raspberry Pi 2: multiparadigm programming tra Java e C#*"[24] di Luca Marzaduri dove viene reso disponibile per applicazioni .NET un servizio Java attraverso un Proxy realizzato con Socket TCP.

Per quanto riguarda il secondo caso d'uso, gli approcci possibili erano stati sintetizzati nella *figura 8*, che distingueva due alternative. La seconda richiede la specificità di Javonet, che però non più essere utilizzato per i limiti imposti dalla licenza, ed è quindi non applicabile, mentre la prima mostra delle analogie con l'architettura della figura 5. E' quindi lecito domandarsi se sia possibile utilizzare, anche per questo caso d'suo gli approcci che prevedono un'architettura distribuita. In questo caso però si presentano delle problematiche differenti, infatti implementare una libreria tuProlog come Server comporterebbe un forte impatto sulla struttura del progetto. Per questo motivo un approccio simile per quanto teoricamente possibile, è probabilmente impraticabile.

6 Conclusioni

In questa tesi a partire dalle premesse iniziali sull'interruzione dello sviluppo di IKVM si è tentato di definire più formalmente i termini del problema al fine di poter effettuare una corretta ricerca della soluzione. L'obbiettivo che si è cercato di raggiungere è stato quello di riformulare il concetto iniziale di "sviluppo di tuProlog.NET" al fine di includere il maggior numero di alternative possibili. Nonostante questo sforzo lo studio di fattibilità effettuato non ha evidenziato nessuno strumento che permetta di ottenere risultati particolarmente significativi, tuttavia con le informazioni ottenute è ora possibile disporre di un quadro completo, relativamente ai temi affrontati, sulle opportunità di sviluppo di tuProlog.

Appendice A – Post in cui viene annunciata l'interruzione dello sviluppo di IKVM

The End of IKVM.NET
After almost fifteen years I have decided to quit working on IKVM.NET. The decision has been a long time coming. Those of you that saw yesterday's Twitter spat, please don't assume that was the cause. It rather shared an underlying cause. I've slowly been losing faith in .NET. Looking back, I guess this process started with the release of .NET 3.5. On the Java side things don't look much better. The Java 9 module system reminds me too much of the generics erasure debacle.

I hope someone will fork IKVM.NET and continue working on it. Although, I'd appreciate it if they'd pick another name. I've gotten so much criticism for the name over the years, that I'd like to hang on to it ☺

I'd like to thank the following people for helping me make this journey or making the journey so much fun: Brian Goetz, Chris Brumme, Chris Laffra, Dawid Weiss, Erik Meijer, Jb Evain, John Rose, Mads Torgersen, Mark Reinhold, Volker Berlin, Wayne Kovsky, The GNU Classpath Community, The Mono Community.

And I want to especially thank my friend Miguel de Icaza for his guidance, support, inspiration and tireless efforts to promote IKVM.

Thank you all and goodbye.

Venerdì 21 aprile 2017 06:26:31 (W. Europe Daylight Time, UTC+02:00)

Bibliografia

- [1] tuProlog - http://tuProlog.apice.unibo.it.
- [2] tuProlog – Manuale pdf
- [3] IKVM- https://www.ikvm.NET/
- [4] Licenza IKVM-http://www.ikvm.NET/license.html
- [5] Fine IKVM- http://weblog.ikvm.NET/2017/04/21/TheEndOfIKVMNET.aspx
- [6] Oracle https://www.oracle.com/it/index.html
- [7] JigSaw Project -http://openjdk.java.NET/projects/jigsaw/
- [8] Bridging https://en.wikipedia.org/wiki/Bridging_(programming)
- [9] Ja.NET - https://sourceforge.NET/projects/janetdev/
- [10] XMLVM http://www.xmlvm.org/overview/http://www.xmlvm.org/overview/
- [11] Sharpen https://github.com/mono/sharpen
- [12] Java Language Conversion Assistant - https://support.microsoft.com/it-it/help/819018/the-java-language-conversion-assistant-jlca-is-now-available
- [13] Java to C# Converter - https://www.tangiblesoftwaresolutions.com/product_details/java_to_csharp_converter.html
- [14] Microsoft uwp https://docs.microsoft.com/it-it/windows/uwp/get-started/universal-application-platform-guide
- [15] Codename One https://www.codenameone.com/
- [16] CodeName One Java support - https://www.codenameone.com/blog/why-we-dont-support-the-full-java-api.html
- [17] JNI Java - https://docs.oracle.com/javase/8/docs/technotes/guides/jni/hj
- [18] jni4net http://jni4net.com/
- [19] Proxygen - https://github.com/jni4net/jni4net/wiki/ProxyGen
- [20] Javonet -https://www.javonet.com/about/what-is-javonet/

- [21] JavaME - http://www.oracle.com/technetwork/java/embedded/javame/index.html
- [22] Java Scripting API- https://docs.oracle.com/javase/8/docs/technotes/guides/scripting/prog_guide/api.html
- [23] Porting e processo di mantenimento di applicazioni Java su piattaforma .NET - Alessandro Montanari http://www.alice.unibo.it/xwiki/bin/view/Theses/JavaNet10
- [24] Windows 10 IoT su Raspberry Pi 2: multiparadigm programming tra Java e C# -- http://apice.unibo.it/xwiki/bin/view/Theses/MarzaduriWin10iOTRaspPi2