Penelope, A Language for Realizing Context Spaces

Sovrin Tolia
Software Technology Laboratory
HP Laboratories Palo Alto
HPL-2002-240
August 26th , 2002*

E-mail: sovrin.tolia@hpl.hp.com

semantic query, programming languages, resource description framework, meta-data

Context Spaces aim to provide a new class of document management services in which storage, organization and retrieval of information is based on semantically rich and active meta-data. It enhances the access to existing document resources by making it more proactive, mobile and context-aware. We have designed a new language, Penelope, to realize Context Spaces. Penelope defines the basic programming language concepts, including primitive data types as objects. It uses the message passing model for expressing operations over these objects. The current implementation of Penelope is based on the Resource Description Framework (RDF) data model and the Schema specification. The property centric approach and the platform independence provided by RDF makes Penelope suitable for building distributed components. This work on Penelope demonstrates the benefits of expressing data and code in a single unifying language.

Contents

Chapter 1

Introduction

The goal of the Context Spaces project is to design an infrastructure for building meta-data based, semantically rich, document storage, organization and retrieval services. Each Context Space is an active computational entity that interacts with other Context Spaces by exchanging messages. The message itself, can contain both the information and the code required to perform the computation over this information.

The design goals and the architecture of Context Spaces is outlined in [10],[16]. To make documents the first class citizens of the Context Spaces, content of the document is wrapped around by meta-documents. Meta-documents contain the actual content, meta-data describing the content, computations associated with the content and meta-data about the meta-document itself. A Context Space is a collection of such meta-documents.

Context Spaces could be realized using two distinct approaches. The first technique is one in which the data and the computation over that data is expressed using different languages. For example, we could use Resource Description Framework [11] for describing the meta-data and Java for expressing all the computations. This is an API-centric approach and results in a certain degree of indirectedness in building such systems. We advocate the use of Penelope for realizing Context Spaces. Penelope is the new language that provides support for both data and computation over this data. It is based on the Resource Description Framework (RDF) data model and the Schema specification. The flexibility and the platform independence provided by the RDF data model makes it suitable for expressing meta-data about documents. The property-centric approach and the semi-structured nature of the RDF data model makes it suitable for expressing computation over this meta-data. Penelope leverages these properties to demonstrate the benefits of representing both data and code in a single unifying language. 4.3 provide further insights on the benefits and drawbacks of the two alternative approaches.

The design of Penelope drew inspiration from SmallTalk. Just like in SmallTalk, Penelope defines programming language concepts, including primitive data types as objects. The message passing paradigm is used to model operations on objects. The code in Penelope could be expressed in both human-readable form as well as in RDF form. Penelope is an interpreted language and the interpreter takes code in RDF form as the input and outputs results expressed in RDF. The results could be facts or a piece of code that would need to be executed to generate the facts. This property of closure, in which both the computation and the result of the computation is expressed in Penelope, makes it very attractive for building disbributed components like Context Spaces.

In the next chapter we provide an introduction to the Resource Description Framework and its suitability for building Penelope. In Chapter 3 we outline some of the existing work that has been done in building languages over RDF. Chapter 4 provides complete details of Penelope, including the need, basic object definitions, translator, interpreter and benefits and drawbacks of the language. In Chapter 6 we highlight some of the experiences we had during the course of this work. Chapter 7 concludes this report with some of the features that could be implemented in Penelop in future.

Chapter 2

Background

RDF [11] is the Resource Description Framework standard proposed by W3C for processing meta-data. It provides interoperability between applications that exchange machine understandable information. The RDF data model defines Resources, Properties and Statements. Anything described in RDF is a Resource and all resources are described using a URI. Properties are subset of resources that define attributes and relationships between resources. A RDF statement is a triple consisting of a subject, predicate and an object or an object value. For example, the statement "Bob is the brother of Alice" will be represented in RDF as

```
<?xml version="1.0"?>
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:rdfs="http://www.w3.org/TR/1999/PR-rdf-schema-19990303#"
   xmlns:hp="http://www.hpl.hp.com/people/">

   <rdf:Description about="&hp;#bob">
    <hp:brotherOf rdf:resource="&hp;#alice"/>
   </rdf:Description>

  <rdf:Description about="&hp;#alice">
    <hp:Name>Alice</hp:Name>
  </rdf:Description>
  <rdf:Description about="&hp;#bob">
    <hp:Name>Bob</hp:Name>
  </rdf:Description>

  <rdf:Property rdf:about="&hp;#Name"
    <rdfs:range rdf:resource="&rdfs;Literal"/>
  </rdf:Property>
</rdf:RDF
```

The RDF Schema specification provides definitions of basic constructs like Resource, Class, Property that can be used by the ontology designer to conceptualize his domain. The objects in Penelope are defined using concepts from RDF Schema. The operations over Penelope objects are defined using the RDF triple data model.

An interesting aspect in the design of RDF was the adoption of the Property-Centric Approach. In this, a property is defined to be applicable to classes of resources. This is distinctly different from the object-oriented design approach, in which classes are defined to have attributes and methods. For example, in object-orientation, you may define a class Person with attribute author. In RDF, you would define the property author having the class person as its domain and literal as its range. A sample RDF description of this may look like this.

```
<rdf:Property rdf:ID="Author">
    <rdfs:domain rdf:resource="#Person"/>
    <rdfs:range rdf:resource="#Literal"/>
</rdf:Description>
```

This property-centric approach in RDF enables entities to say anything about any existing resources. For example, someone might define a class person and define a property email to identify an individual. The others might use the same definition of the class Person but would define a property SSN number along with the property email. Penelope leverages this property to allow programmers to add methods to classes and instances (without sub-classing) both at design-time and run-time. Section 4.1 outlines the benefits of this in greater detail.

The inherent platform neutraility in RDF makes Penelope platform independent. The semi-structured nature of the RDF data model enables the Penelope programmer to specify arbitary models for both data and computation over data. The RDF specification has been adopted by the W3C as a standard and consequently may be adopted as a defacto language for meta-data. In such a case, Penelope would prove to be extremely useful in expressing computation over the meta-data.

Chapter 3

Related Work

Context File System [14] is a part of the Active Spaces project at University of Illinois Urbana Champaign. In principle, context file system also intends to provide context-based document management services for users. However, their approach is different from ours in the way storage, organization and retrieval is addressed. They use the concept of virtual directories to organize documents based on context and have no notion of meta-data based information retrieval. Further they dont directly address the rich querying facility required for semantic retrieval. Unlike context file system, Context Spaces advocates the use of the same language for both data and computation and makes an initial attempt to build this new language, Penelope.

Haystack [9] is a personal information store that caters to the creation, organization and visualization of information using RDF. An essential component of Haystack is Adenine, a programming language suited for manipulating RDF. Adenine shares the same motivation to realize HayStack as Penelope does to realize Context Spaces. One essential difference between Adenine and Penelope is that Adenine closely models the Lisp and Python notation, whereas Penelope emulates the SmallTalk and Java conventions. Adenine does not use RDF for storing the execution state whereas Penelope uses end-to-end RDF for program execution. This has the obvious advantage for code migration during execution.

Fabl [6] is a computation language designed in RDF. Its code, functions and classes are formalized as RDF resources and it provides a dual programming approach. Like Penelope, in Fabl the programmer can specify code in RDF or javascript type syntax. As with Adenine, Fabl does not store the program execution state in RDF making code migration complex.

The ability to express computation in a data modeling language appears in languages like SMIL [7] and XSLT [5]. SMIL is an XML-based language defined to enable effective layout of multimedia content. It supports the switch construct for conditional execution. XSLT is language for transforming XML documents into other XML documents and allows conditional execution by using the if construct. These languages provide extremely preliminary computations (conditionals) for expressing computation.

The pure object-oriented nature of Penelope and the ability to express computation using messages is motivated from the SmallTalk [3] programming model. The syntactical flavor of Penelope was motivated from Java [13]. The Stack based approach to program interpretation in Penelope was motivated from the internal working of the Java Virtual Machine. Although the RDF Jena API [12] is used for building the interpreter, its use highlights some of the drawbacks in the API-centric approach for manipulating RDF.

Chapter 4

Penelope, A Language for Realizing Context Spaces

4.1 Motivation

The Context Spaces project envisions that the document content will be supplemented with active and expressive meta-data. Active meta-data implies that it will have both information and the code to act on that information. The active property of documents can be extremely useful in complex information sharing environments. For example, the department secretary may receive an email having the unsorted list of employees who are attending the baseball game. She may want to store this list in the sorted manner. It would be extremely flexible if the document itself contained methods to sort the list based on various keys, rather than she having to feed this information to self-written or custom sorting software. Arguably, the content provider cannot possibly imagine all the possible uses of the content and provide methods for manipulations but generic computational methods along with data can prove to be handy.

Active documents can be useful in distributed environments where semantically similar but syntactically different pieces of information can come from various sources. For example, a user may receive a Mastercard credit statement that expresses the amount in two separate fields (dollars and cents) and Visa credit statement that expresses the amount in a single field. Automated processing of this information requires conversion of amounts to one consistent representation. This would be facilitated if both these documents had methods to convert the amount into a standard representation.

Expressivity implies that meta-data will have inherent semantics attached to it. Document management systems will need to take into account the availability of such rich and active meta-data to provide semantically grounded, storage, organization and retrieval services. To provide a unifying and homogeneous mechanism for building such systems, the Context Spaces project designs a new language Penelope.

Context Spaces are essentially distributed components. For providing the most effective document management services, these components would need to query and interact with each other. The request-response paradigm in such distributed environments becomes extremly powerful, if the sender is able to specify code for computing the queries, apart from the standard information passing. The receiver would in turn, process the code and return the result as facts or other pieces of code. The need for such loosely coupled, distributed components and the support for its messaging infrastructure further motivates the design of Penelope.

Chen et.al. [4] outline a technique called Rule Shipping Technique in which the sender specifies the rules to be executed against the knowledge-base of the receiver. This approach had a certain degree of indirectedness as the data was specified in RDF and rules were specified in prolog style predicates. Penelope intends to serve as an end-to-end solution for both data and computation.

Further in traditional object-oriented languages, it is not possible to supplement the class definition with methods and attributes during runtime (Compile-time extensions are indeed possible with inheritence and overriding). The requirement becomes more interesting when we want to add methods and properties to instances. Penelope intends to demonstrate this flexibility of runtime-time extensions to classes and instances. The capability to dynamically add methods to classes and instances at runtime has several advantages. It breaks away from traditionally static nature of object oriented designs allowing objects to grow and shrink during their lifetime. We eliminate the need to recompile the enhanced object definitions for the enhancements to take effect. The programmer is freed from the burden of creating a sub-class hierarchy for incorporating new and existing object definitions. It enables loosely coupled distributed components to involve in more meaningful negotiations.

Domain specific programming languages have received a lot of attention in the past and some of them have been very successful. An inherent characteristic of domain specific programming languages is that programmers are provided with an intrinsic syntactical and conceptual support for the underlying data model. An example of this would be MATLAB. There is a learning curve associated with such languages, but once that barrier is overcome, programmers find the use of such languages extremely attractive. Penelope intends to target the domain of meta-data for documents that is expressed using rich semantic languages.

4.2 Choice of RDF for Penelope

When it comes to designing a new set of markups with inherent semantics attached to it, three technologies come to light, XML [2] , RDF [11] and DAML [1]. XML Schemas and its associated toolsets are widely used in the industry today. XML documents tend to be very structured and clear for the reader to follow. On the other hand, RDF or DAML based documents are much more verbose and often quite difficult for the reader to comprehend. However, XML was intended primarily to provide structure to documents rather than semantics. The use of XML in designing Penelope would require us to reinvent the wheel in defining base relationships like subClassOf, subPropertyOf, instanceOf etc. We would have had to define the concepts of classes, properties and its associated model for representation. Further, since there is no inherent data-model associated with XML, reasoning over information expressed in XML would not be as sophisticated as it would be in the case of RDF and DAML.

RDF has gained popularity as one of the emerging defacto standards for describing meta-data on the web. The interoperability achieved by applications using RDF and the platform neutrality obtained from its XML serialization syntax makes it particularly suitable for describing and exchanging information on the web. Realizing the benefits of RDF, W3C has adopted RDF as a standard for describing meta-data.

RDF Schemas provide many of the base relationships required to model real-world concepts and express semantic queries over meta-data. It's associated triple data-model provides a simple but powerful way to make statements about entities in the world. This semi-structured nature of the data model makes it suitable for building concrete models for data and computation. The property-centric approach in RDF enables Penelope to provide the flexibility to add methods and attributes to classes and instances at runtime. The design of Penelope requires it to be platform independent. Further, it should not prove to be an hindrance in building distributed components that aim for interoperability. RDF seems to be a reasonable choice as far as these two design goals are concerned.

Though the RDF toolset is not quite as mature as the XML toolset, it is quite sufficient for the present exercise. The XML serialization of RDF could become quite tedious for the programmer to express his code, but we envision that with the availability of more sophisticated toolkits, these intricate low-level details could be easily hidden from the programmer. DAML builds over RDF with additional relationship properties but the DAML toolset is not quite sufficient to rapidly prototype systems.

The researchers in the RDF and DAML [1] working groups have identified some of the deficiencies in these specifications and are working towards providing a language called Web Ontology Language(OWL) [8] that has more expressivity than what RDF has presently. We envision that Penelope could be extended to incorporate new semantic markups quite flexibly, since its already based on RDF.

4.3 API-Centric Approach Vs Penelope

In the API-centric approach, programmers try to make the language's syntax support the underlying data model. This becomes an obvious use-case for the extensible nature of object-oriented languages. For example, we could have the underlying data model in RDF and use Java for manipulating RDF expressions. In this case, a programmer would create class libraries for encapsulating the information expressed in the RDF data model. Thereafter provide methods over these classes to compute over the information. Our claim is that API-centric approach adds more complexity for performing computation over information expressed in RDF as opposed to computational code expressed in RDF itself.

Program Goal: Given a list of employees working in the company, find out all the employees who are above the age of 50 (to quality for the early retirement program).

In this example, we assume that the company maintains employee information in the RDF form and may look like this:

```
<rdf:Description about="&hp;#592013">
  <rdf:type rdf:resource="&hp;#HPEmployee"/>
  <hp:Name>John Doe</hp:Name>
  <hp:Age>51</hp:Age>
</rdf:Description>

<rdf:Description about="&hp;#592014">
  <rdf:type rdf:resource="&hp;#HPEmployee"/>
  <hp:Name>Bill Smith</hp:Name>
  <hp:Age>45</hp:Age>
</rdf:Description>
```

A piece of code to carry out the intended goal in Penelope would look like the following.

```
// Assumes that current program holds the RDF information
Variable X = this.query(null,RDF.type,&hp;#HPEmployee);

while(X.hasNext()) {
Variable Y = X.next();
if (Y.Age > 50)
  Y.print();
}
```

If the same functionality is implemented in Java and Jena, the code might look like the following.

```
// Jena stores RDF statements in a model
StmtIterator sit = model.listStatements(new SelectorImpl(null,RDF.type,&hp;#HPEmployee));

while(sit.hasNext()) {
Statement s = (Statement)sit.next();
RDFNode obj = s.getObject();
if (obj instanceof Literal) {
   Integer ageInt = new Integer(obj.toString());
   if (ageInt.intValue() > 50) {
      Statement st = model.getProperty(s.getSubject(),&hp;#Name);
      System.out.println(st.getObject().toString());
           }
        }
}
```

From the above two examples it is clear that the Java based approach has a level of indirection in comparing for the age field as opposed to the Penelope based approach. We envision that as data types become more complex, the API-centric approach would make programming complex. To this end, Penelope advocates the use of a language whose syntax supports the underlying data model.

4.4 Design

To enable computation over meta-data expressed in RDF, we designed Penelope. The design of Penelope was heavily influenced by the design of SmallTalk. In SmallTalk everything is an object. This is in direct contrast to languages like Java, which though being Object-Oriented, provide support for primitive data types. The uniform object representation provides a unified way to model the foundation data objects. The blurring of distinction between objects and primitive data types, removes the additional complexity needed to support them in the interpreting engine. Further, it makes the architecture of the interpreting engine uniform in handling operations and expressions. Thus, being a "Pure Object-Oriented Language" is one of the key strengths of Penelope.

In Penelope expressions and computations over them are modeled as messages. This message passing paradigm has been directly influenced from the SmallTalk model for evaluating expressions. For example, for evaluating the expression 2+3, a message is sent to Object '2' with method selector '+' and parameter being the object '3'. The message-passing paradigm is ideal for our language, since one of its design goals is to allow distributed components to interact with each other.

Another key design feature of Penelope is that it satisfies the property of "Closure". This means that the Penelope interpreting engine accepts code expressed in RDF and provides the result of the computations also in RDF. The most important benefit of this property is that it makes programs portable. Another key advantage of expressing results in RDF is that no separate processing engine is required for interpreting the results. Using this approach, the interpreted results may be information or code that would need to be run for computing the results. This design feature of Penelope was inspired by P-Code virtual machine execution models. The concept of P-Code is not new. It was used by versions of Pascal in the early 70's. However, with the emergence of Java and Java Virtual Machine Bytecodes, the virtual machines for programming languages have gained popularity. The basic idea is to represent the compiled code in some intermediate form so as to make it portable across different computing architectures. This does make the execution of virtual machine based programs slower than their native counterpart. In Penelope, RDF is used to represent P-Code or bytecodes. The inherent platform neutraility of RDF thus makes Penelope platform independent.

Penelope supports program migration. Seamless program migration has been enabled by the way of using RDF expressions for describing the object model and maintenance of internal state of the program execution also in RDF. This use of RDF for maintaining internal state of program execution makes program migration independent of the platforms of the hosts to which the migration takes place. Further, the internal state maintenance in RDF gives the programmer better control over the persistence of their programs.

The syntax of Penelope closely resembles Java. The present implementation supports only untyped Variables, though we do envision the need to support typed variables for compile-time type-checking and optimization. All parameter passing in Penelope is enabled by references and the reference in this case would be a URI pointing to the object instance. Penelope is an interpreted language and its processing engine is powered by a translator and an interpreter. In Penelope, the programmer could specify code using both Java-type syntax as well as in RDF. The subset of the Penelope vocabulary is dedicated to defining the code constructs. A Penelope program is a collection of statements and each statement is a collection of messages with the statement itself being a message. This abstraction provides a unifying way for the interpreter to handle the processing of code blocks.

Penelope belongs to the class of Domain Specific Programming Languages. The basic purpose of Penelope is to support computations over the RDF data model. As outlined in section 4.3, this relieves the programmer from the additional complexity of providing support for computations over RDF expressions. The inherent use of RDF enables Penelope to support data and meta-data expressed in RDF. Thus, the support for data in Penelope is restricted to the extent to which RDF supports expressions for data.

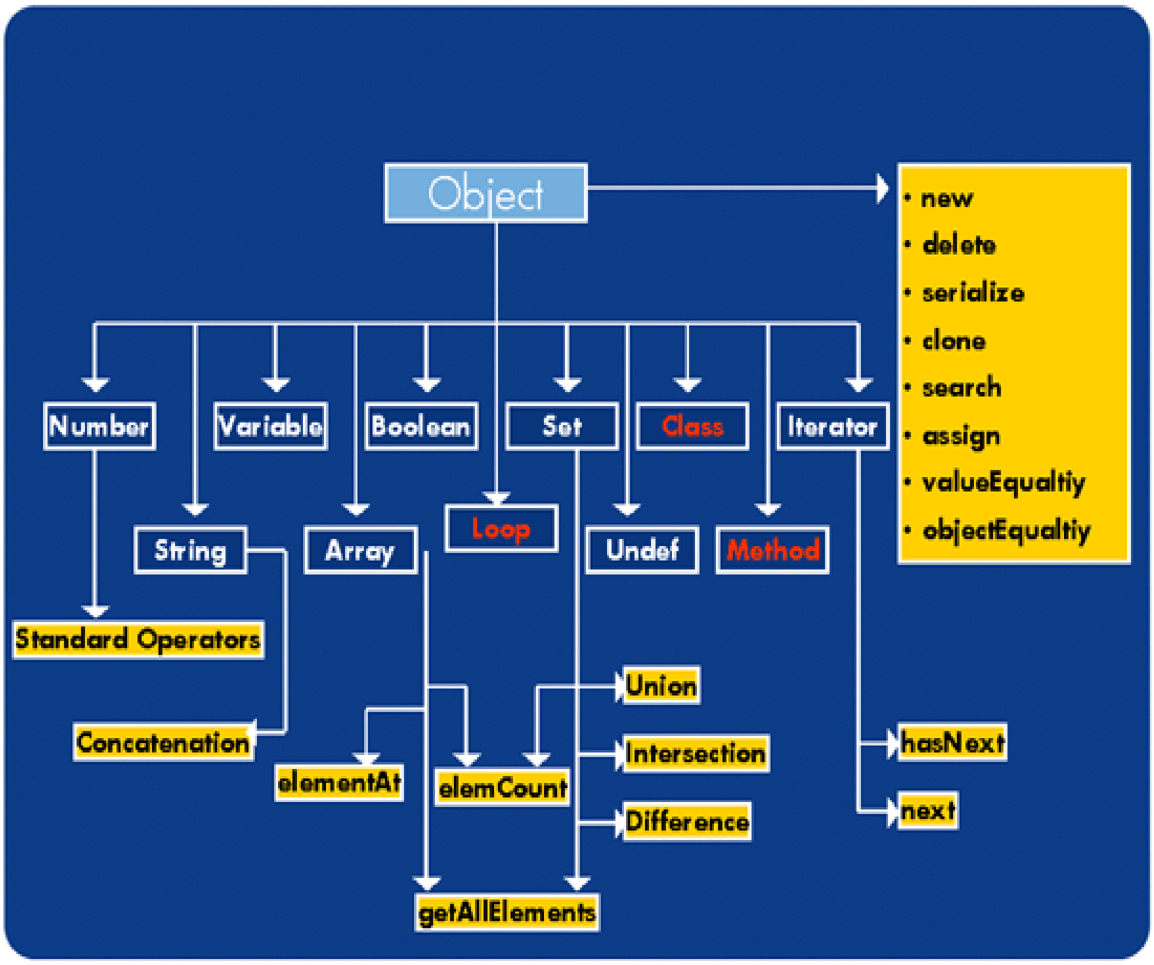


Figure 4.1: Objects and Operations Supported by Penelope Vocabulary

4.5 Object Definitions

There are two distinct object modules in the Penelope vocabulary. The code vocabulary provides constructs with which the programmer structures his code. The other module specifies the foundation objects supported by the Penelope language.

In Penelope, an Object is the base unit of conceptualization. Every object is a RDF resource and is identified by a unique URI. The process of computing these URI's is described in [15]. In essence, the URIs for objects are constructed by computing the hash over all the object properties.

Figure 4.1 outlines all the foundation data objects supported by Penelope. It also provides the operations supported on these objects. As depicted, the following objects are supported by the first version of the language, Number, String Variable, Boolean, Set, Class, Method, Iterator, Undefined. All the objects support base methods like new, delete, serialize, query, value equality, object equality, clone and assignment. For the Collection objects like Array and Set, the union, intersection and difference operations are supported. For the Iterator object, hasNext and next methods are supported. The vocabulary also specifies methods for carrying out standard operations like addition, subtraction, multiplication and division. The conditional constructs provided by Penelope include the while and the if. In both these cases, their restrictive form is used, one in which the condition is a single granular expression. The code constructs provided by the Penelope vocabulary include, the definition for a program, code block, statements and messages (expressions). Penelope Vocabulary does define Class and Methods but the current version of the interpreter does not support the execution of these definitions.

For example, the String Object in Penelope would like the following:

```
<rdf:Description rdf:about='hashedrdf.v1:SHA1=f4e8d44db809ac747ae20ae74482a3375f82d9db'>
  <rdf:type rdf:resource='http://hpl.hp.com/Penelope/Primitive#String'/>
  <NS0:string_value>Sovrin</NS0:string_value>
</rdf:Description>
```

An Array object in Penelope would look like the following:

```
** This describes an array having a sequence of elements and an
element count associated with it
<rdf:Description rdf:about='hashedrdf.v1:SHA1=71500bd290571f3c8590ebce286ce17217ee8240'>
    <rdf:type rdf:resource='http://hpl.hp.com/Penelope/Primitive#Array'/>
    <NS0:element-count rdf:resource='hashedrdf.v1:SHA1=a4564059376886426c4dbd08d358de409a2a60ee'/>
    <NS0:array-elements rdf:resource='hashedrdf.v1:SHA1=ce2e1f35833c4225716466fa6ebfb98bbaff3959'/>
</rdf:Description>

** Defines the Sequence Container to hold references to all elements
of the array
<rdf:Description rdf:about='hashedrdf.v1:SHA1=ce2e1f35833c4225716466fa6ebfb98bbaff3959'>
    <rdf:_1 rdf:resource='hashedrdf.v1:SHA1=7647e707d99da04c58c25f42b2b53f3235f735d2'/>
    <rdf:type rdf:resource='http://www.w3.org/1999/02/22-rdf-syntax-ns#Seq'/>
</rdf:Description>

** Number object, denoting the element of the array
<rdf:Description rdf:about='hashedrdf.v1:SHA1=7647e707d99da04c58c25f42b2b53f3235f735d2'>
    <rdf:type rdf:resource='http://hpl.hp.com/Penelope/Primitive#Number'/>
    <NS0:numeric_value>5</NS0:numeric_value>
</rdf:Description>

** Number object, denoting the element count of the array
```

```
<rdf:Description rdf:about='hashedrdf.v1:SHA1=a4564059376886426c4dbd08d358de409a2a60ee'>
    <rdf:type rdf:resource='http://hpl.hp.com/Penelope/Primitive#Number'/>
    <NS0:numeric_value>1</NS0:numeric_value>
</rdf:Description>
```

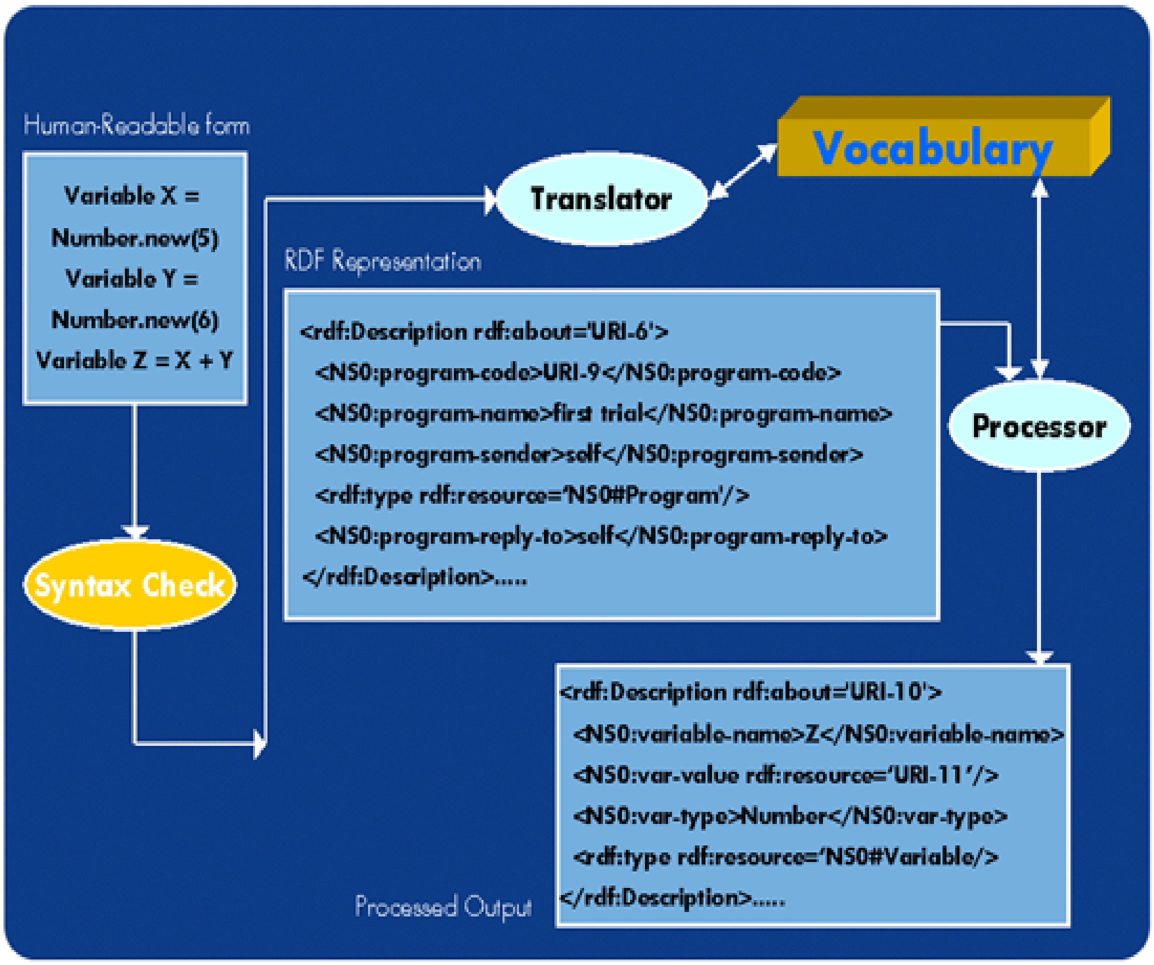
4.6 Translation and Interpretation

The Penelope programmer could write code in Java-based syntax or in RDF syntax. The human-readable form of the code will be really simple for the programmer to model his computations. We envision that RDF form of the code would be useful for distributed components to query and interact with each other. This two-level indirection is introduced in writing code to maintain the closure property within the interpreter - it takes in code that is expressed in RDF and outputs results in RDF. The translator takes care of mapping the human-code to rdf-code. The schema for the code constructs is kept independant of the underlying schema for the language. This is to ensure that programs written need not have to change if the underlying schema changes.

The translator converts the human-code to rdf-code using the message-paradigm. Using this, Penelope code is expressed as a Program. A program is modeled as a collection of statements. A Statement is modeled as a collection of messages. This implies that a statement, in effect, is also a message. The translator prepares the code stack for the interpreter to process. The code stack is a collection of statements with each statement holding the pointer to the first message. Messages follow left to right associativity during evaluation.

RDF essentially does not impose any ordering on the statements about resources. When the RDF processing engine encounters an RDF description, it does not guarantee the order in which triples are generated. However, in our case, we need to know the sequential order for the execution of statements. Rather than expecting the programmer to explicitly specify line numbers, we impose an implicit restriction on the code expressed in RDF. The programmer is mandated to use the Container Sequence to express all the statements in the program. This gives the interpreter a precise notion of the order of execution of statements.

The interpreter is the processing unit for executing code expressed in Penelope language constructs. It



```
Variable X =
Number.new(5)
Variable Y =
Number.new(6)
Variable Z = X + Y
```

```
<rdf:Description rdf:about='URI-6'>
  <NS0:program-code>URI-9</NS0:program-code>
  <NS0:program-name>first trial</NS0:program-name>
  <NS0:program-sender>self</NS0:program-sender>
  <rdf:type rdf:resource='NS0#Program'/>
  <NS0:program-reply-to>self</NS0:program-reply-to>
</rdf:Description>.....
```

```
<rdf:Description rdf:about='URI-10'>
  <NS0:variable-name>Z</NS0:variable-name>
  <NS0:var-value rdf:resource='URI-11'/>
  <NS0:var-type>Number</NS0:var-type>
  <rdf:type rdf:resource='NS0#Variable/>
</rdf:Description>.....
```

Figure 4.2: Anatomy of Interpreting

supports two modes of execution, one in which command line parameters could be passed to it for initiating the request and the other being operating in the wrapper mode. In this mode, it is wrapped up by an Distributed Component Interface and predominantly processes all incoming messages to the component. The initial prototype of the interpreter is single-threaded. Thus when the interpreter is being used in the wrapper mode, queing of messages and response synchronization has to be done by the wrapper container.

The interpreter uses Jena's [12] memory model as it's RDF statements store. Upon careful examination of the Jena's Internal In-memory storage model, we find that it trades off storage costs for search costs. However, we anticipate that in future, this storage model might have to be optmized for better performance. Jena's support for an abstract Model for a collection of statements and extensible plug-in modules for serializing the model are two of the most prominent features that make it attractive to use.

The interpreter maintains the complete state of program execution in the RDF memory model provided by Jena. This ensures the property of persistence (if required) and enables code migration quite easily. It has an associated model for code, for tracking the statements that have been executed in the program. Figure-4.2 outlines flow of execution of code. The code expressed in the figure is partial and complete view of the code can be found in section-4.7. For all the basic operations on objects that are supported by Penelope, the interpreter maintains a mapping between the methods and the actual interpreter method that would perform the operation. The operation is carried out by using Java Reflection over objects. This mapping is maintained in the RDF memory model provided by Jena. An example of the mapping entry in the model would look like:

```
<rdf:Description rdf:about='hashedrdf.v1:SHA1=cd1cdeef62888e8db2befe3b5f572cecd20491ae'>
    <rdf:type rdf:resource='http://hpl.hp.com/Penelope/Primitive#Method'/>
    <NS0:method-type>Java</NS0:method-type>
    <NS0:method-name>new</NS0:method-name>
    <NS0:method-code>newPenObject</NS0:method-code>
    <NS0:method-impl-class>Interpreter</NS0:method-impl-class>
    <NS0:method-class>String</NS0:method-class>
</rdf:Description>
```

Current version of the syntax checking for the language is simple and does checking for code expressed in java-based syntax. However, we aim to provide syntax and type-checking for the rdf-form of the code, so that syntax checking has to be done for just one form.

4.6.1 Error Handling

Both the translator and the interpreter support preliminary form of error handling. The Penelope language defines some basic forms of errors like null pointer exception in the interpreter, invalid arguments and invalid RDF code. These errors messages are expressed in RDF to facilitate the process of seamless error identification between distributed components. For example, A null pointer exception in the interpreter would look like the following

```
<rdf:RDF
  xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
  xmlns:NS0='http://hpl.hp.com/Penelope/Primitive#'
>
<rdf:Description rdf:about='hashedrdf.v1:SHA1=608e5e8edfe8c858b1f069b0d8ea5b62df2c0142'>
    <rdf:type rdf:resource='http://hpl.hp.com/Penelope/Primitive#Error'/>
    <NS0:error-message>Null Pointer Exception</NS0:error-message>
  </rdf:Description>
</rdf:RDF>
```

4.7 Illustrative Example

We provide a very simple example for illustrating the basic way the code is structured and executed in Penelope.

Suppose, we want to add two numbers and print the result. In the code below ** marked lines is used to denote comments for the reader.

Human Form of the Code will look like the following:

```
** Human readable syntax of Penelope is closely modelled on Java syntax
Variable X = Number.new(5);
Variable Y = Number.new(6);

** Expressions in Penelope are modelled as messages. In the next
** statement, a message is sent to object X with method selector + and
** argument as Y
Variable Z = X + Y;

** Serializes the RDF description of the object
Z.print();
```

Its Corresponding RDF Form would look like the following (To avoid redundancy, we provide a particular representation of a statement in the program):

```
** Namespace declaration for the program
<rdf:RDF
  xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
  xmlns:NS0='http://hpl.hp.com/Penelope/Primitive#'
 >

** This is the program block that serves as a collection of code blocks
<rdf:Description rdf:about='URI-6'>
    <NS0:program-code>URI-9</NS0:program-code>
    <NS0:program-name>firsttrial</NS0:program-name>
    <NS0:program-sender>self</NS0:program-sender>
    <rdf:type rdf:resource='http://hpl.hp.com/Penelope/Primitive#Program'/>
    <NS0:program-reply-to>self</NS0:program-reply-to>
</rdf:Description>


** This is the Code block that serves as a collection of statements
<rdf:Description rdf:about='URI-9'>
    <NS0:code-element>URI-3</NS0:code-element>
    <rdf:type rdf:resource='http://hpl.hp.com/Penelope/Primitive#Code'/>
</rdf:Description>

** A sequence holding reference to a collection of statements
<rdf:Description rdf:about='URI-3'>
    <rdf:_1 rdf:resource='URI-4'/>
    <rdf:type rdf:resource='http://www.w3.org/1999/02/22-rdf-syntax-ns#Seq'/>
</rdf:Description>

** The basic statement type in the Penelope vocabulary
<rdf:Description rdf:about='URI-4'>
    <NS0:statement-element>URI-1</NS0:statement-element>
    <rdf:type rdf:resource='http://hpl.hp.com/Penelope/Primitive#Statement'/>
</rdf:Description>

** This is the statement block that serves as a collection of Messages
** Note that the statement itself is a message
<rdf:Description rdf:about='URI-1'>
    <rdf:type rdf:resource='http://hpl.hp.com/Penelope/Primitive#Message'/>
    <NS0:receiver>Variable X</NS0:receiver>
    <NS0:message-argument>URI-2</NS0:message-argument>
    <NS0:operator>=</NS0:operator>
</rdf:Description>

** Sequence denoting the collection of arguments to the message
<rdf:Description rdf:about='URI-2'>
    <rdf:_1 rdf:resource='URI-7'/>
    <rdf:type rdf:resource='http://www.w3.org/1999/02/22-rdf-syntax-ns#Seq'/>
</rdf:Description>

** The basic argument type from the penelope vocabulary
<rdf:Description rdf:about='URI-7'>
    <rdf:type rdf:resource='http://hpl.hp.com/Penelope/Primitive#Argument'/>
    <NS0:argument-type rdf:resource='NS0;#Message'/>
    <NS0:argument-value rdf:resource='URI-8'/>
</rdf:Description>
```

```
** This represents a message that is passed to object Number with
** method selector new and argument as Number value 5
<rdf:Description rdf:about='URI-8'>
    <NS0:operator>new</NS0:operator>
    <NS0:receiver>Number</NS0:receiver>
    <NS0:message-argument>URI-12</NS0:message-argument>
    <rdf:type rdf:resource='http://hpl.hp.com/Penelope/Primitive#Message'/>
</rdf:Description>

<rdf:Description rdf:about='URI-12'>
    <rdf:type rdf:resource='http://www.w3.org/1999/02/22-rdf-syntax-ns#Seq'/>
    <rdf:_1 rdf:resource='URI-5'/>
</rdf:Description>

<rdf:Description rdf:about='URI-5'>
    <NS0:argument-value>5</NS0:argument-value>
    <NS0:argument-type rdf:resource='NS0;#Number'/>
    <rdf:type rdf:resource='http://hpl.hp.com/Penelope/Primitive#Argument'/>
</rdf:Description>

** This denotes the print operation on the variable Y
<rdf:Description rdf:about='URI-31'>
    <NS0:operator>print</NS0:operator>
    <NS0:receiver-object>Z</NS0:receiver-object>
    <NS0:message-argument rdf:resource='URI32'/>
    <NS0:receiver>Z</NS0:receiver>
    <rdf:type rdf:resource='http://hpl.hp.com/Penelope/Primitive#Message'/>
</rdf:Description>

** This denotes that the print operation does not take any arguments
<rdf:Description rdf:about='URI32'>
    <rdf:type rdf:resource='http://www.w3.org/1999/02/22-rdf-syntax-ns#Seq'/>
</rdf:Description>


</rdf:RDF>
```

The result of the Interpretation process would look like the following.

```
** A Number instance with value 11
<rdf:Description rdf:about='hashedrdf.v1:SHA1=uri1'>
    <rdf:type rdf:resource='http://hpl.hp.com/Penelope/Primitive#Number'/>
    <NS0:numeric_value>11</NS0:numeric_value>
</rdf:Description>

** A Variable Instance for Z
<rdf:Description rdf:about='hashedrdf.v1:SHA1=uri2'>
    <rdf:type rdf:resource='http://hpl.hp.com/Penelope/Primitive#Variable'/>
    <NS0:var-name>Z</NS0:var-name>
    <NS0:var-value rdf:resource='hashedrdf.v1:SHA1=uri1'/>
</rdf:Description>
```

4.8 Advantages of Penelope

Though Penelope is in the preliminary state of implementation, it provides several advantages.

- It satisfies the property of closure. This preserves the intended semantics of operations conceived during design time in the runtime environment.
- Dynamic addition of methods and attributes to classes and object instances at runtime. This is one of the strongest feature of Penelope and would greatly help distributed components interact with each other for performing different computations.
- Portability and Extensibility. Portability in Penelope comes directly from the platform neutrality of RDF. Extensibility in Penelope could be achieved by extending the existing ontology for the foundation objects and/or by writing ontological libraries for supporting different operations in Penelope.
- Simplicity of usage. We believe that once RDF based meta-data proliferates the web, languages like Penelope would make the processing of this information simple.
- Language like Penelope, blur the boundary between code and data. This would greatly simplify the interaction between distributed components.

4.9 Drawbacks of Penelope

From our experience it is quite clear that manipulating RDF expressions is difficult for the programmer. However, we envision that more sophisticated toolkits and development environments will emerge that will take away this complexity from the programmer. The current implementation of Penelope does not support garbage collection. Memory management in Penelope would require further research especially since, in this case, the memory is organized as a collection of statements.

RDF does not support datatypes. The only datatype that is supported in RDF is literal. Consequently, precision-based numerical computations cannot be done in Penelope in a unified way. From our experience of building Penelope, it seems that the process of debugging the program could become quite tedious. We would need RDF based debuggers that provide simple yet sophisticated view of the current program state.

In the current state of RDF technology, the adoption of Penelope would be difficult. However, we believe that more sophisticated RDF based intergrated development environments will emerge that will greatly simplify the process of writing programs in languages like Penelope.

Chapter 5

Realizing Context Spaces using Penelope - An Example

We take a very simple example here to illustrate how Context Spaces would use Penelope for interaction. Lets assume that the department secretary has sent out emails asking for people who would be joining the company picnic(This is done by the Picnic Context Space, created for the event). Individuals respond with email messages confirming their participation. The email messages may have meta-data expressed in RDF that contain employee identification number.

The picnic Context Space will collate all the responses to prepare a final list of attendees and sent out reminders to employees who did not respond. For performing this kind of an operation, the code in Penelope would look like the following.

```
{

** global declaration for the list of attendees
Variable Z = Set.new();

** Receives confirmation response for participation
** The variable X holds a Number object having the employee number
Variable X = (get-message(this.query(null,EMPno,null)));
Z.addElement(X);

** Once the deadline for the response has passed, the picnic context
** space decides to send out reminders to employees who did not respond.
** Assume Variable A is a set containing all the employees in the
** department
Variable B = A.difference(Z);
sendReminders(B);
```

We have seen in the previous examples, of how objects look like in Penelope. Here we outline how the difference operation is represented in RDF.

```
** Message Representation
<rdf:Description rdf:about='hashedrdf.v1:SHA1=df8c8d1c9a633773585401264416d029fffb54cb'>
    <NS0:message-argument rdf:resource='mesgargs'/>
    <NS0:receiver>A</NS0:receiver>
    <NS0:operator>union</NS0:operator>
    <NS0:receiver-object>A</NS0:receiver-object>
    <rdf:type rdf:resource='http://hpl.hp.com/Penelope/Primitive#Message'/>
</rdf:Description>

** Sequence of Arguments for the message
<rdf:Description rdf:about='mesgargs'>
    <rdf:type rdf:resource='http://www.w3.org/1999/02/22-rdf-syntax-ns#Seq'/>
    <rdf:_1 rdf:resource='hashedrdf.v1:SHA1=9ece1735d7e5ea28ff65294c4923056a76528acf'/>
</rdf:Description>

** The Argument instance
<rdf:Description rdf:about='hashedrdf.v1:SHA1=9ece1735d7e5ea28ff65294c4923056a76528acf'>
    <NS0:argument-type rdf:resource='http://hpl.hp.com/Penelope/Primitive#Variable'/>
    <NS0:argument-value>Z</NS0:argument-value>
    <rdf:type rdf:resource='http://hpl.hp.com/Penelope/Primitive#Argument'/>
</rdf:Description>
```

The key processing element in this code is the difference operation. The Penelope Vocabulary defines Sets and the difference operation over sets. The interpreter maintains information about sets as a RDF Bag. Internally, when the difference operation is called, examines the contents of both these Bags and creates a new Bag having the list of employees. This Bag gets assigned to a new Set object and is assigned to the variable B.

Chapter 6

Experiences

The use of RDF for building Penelope identified some of the key deficiences in the present RDF Schema. Currently, RDF Schema supports Literals as its only datatype. As the use of RDF becomes more versatile, we would need standard datatypes to be modelled in the schema specification. Penelope models arguments in the method declarations and calls as a RDFS Sequence. However, this modelling is slightly imprecise, as there is no way in RDFS to specify that a method takes exactly 'x' parameters. Although Penelope provides both object equality and value equality, there is no explicit way in RDFS to state this equivalence.

Multiple inheritence in traditional programming languages is specified using the sub-classing concept. However, it should be possible for the programmer to define his classes as unions and/or intersection of other pre-defined classes. If Penelope intends to support this feature, the present RDF Schema does not give explicit constructs to state this. Arguably, Sub-Classing may be sufficient to provide all the inheritence functionalities that a programmer needs (Is there is patent in this, specifying classes not just by subclassing but by union and intersection of class definitions present on the web :-)).

During the course of initial prototyping, we felt the need for selective inheritence among classes in RDF. For example, if we have a class Person with firstname, lastname and email properties, we might want to inherit from this class to define a new class with only the property email and add the name property to this inherited class. Upon careful examination of inheritence in Java, we find that in java also all the members are inherited by default and access modifiers control the use of inherited members. We envision that the subClass relationship in RDF should be supplemented with constraints letting the user specify the properties to be inherited or avoided.

We also encountered the need to inherit from instances. This is required to inherit default values. It seems that RDF Schema should provide constructs for specifying default values for properties so that the inherited classes and/or instances would be able to use these default values rather than specifying these repeatedly.

In designing the first pass of the interpreter, we investigated two techniques based on where the RDF representation of the standard data objects are stored. In the first one, the idea was to investigate the "self rendering property" of objects. It implies that every object would provide means to render itself in different formats. In this approach the triples were embedded in the object prototype instances. This approach might have been useful in case we wanted to provide the programmers the ability to extend the base-schema itself. However, we envision that the programmers would define additional constructs using the standard constructs that are provided by the language. Moreover, the code of the interpreter in this case became somewhat unclean.

The second approach was to use the ontology as a visual graph while coding the interpreter. This resulted in the interpreter having more object-based code and was thus clean. In this case all the language defined constructs are defined in the interpreter.

Chapter 7

Conclusion and Future Work

Context-Spaces provide an interesting paradigm in building document management services. In this document, we have outlined the design of the new language Penelope that is being developed to realize Context Spaces. The current implementation of Penelope is in the early stages but it did provide some promising insights into the development of RDF based languages.

The salient features of Penelope are being a pure object oriented language, satisfying the property of closure for expressing and processing computation and its ability to support addition of methods to classes and instances at runtime. Usage of Penelope to manipulate RDF data model would greatly relieve the programmer from writing verbose object-oriented application programming interfaces.

Penelope belongs to the class of domain programming languges. We believe that as meta-data proliferates the web, such meta-data processing languages will come into existence. Penelope goes a step forward in being a meta-data programming language. It provides support for semantic operations, derived from the use of RDF. It may be possible that standards other than RDF is accepted as a defacto for meta-data. Penelope would be easily extensible to support any standard that is derived from RDF.

To extend Penelope to become a complete usable programming language, extensive syntax checking will need to be done. Further, support for typed variables should be introduced with effective compile-time type checking. The foundation data objects in Penelope could be extended to support more methods and different representations. For example, the object Number could be expanded to represent real, double, float and other types of numeric objects. The object String could be enhanced with more sophisticated string manipulation functions. We have implemented a very basic version of the language Penelope and expect that with more strong use-cases being developed, the language could be strengthened to support a large programmer community.

Appendix A

Penelope Vocabulary

Here we provide an RDF Schema based view of the Penelope Vocabulary. All the Penelope foundation data objects are treated as RDFS Classes. They were modeled as Classes instead of Resources, as Classes are Resources and are intended to define a concept. This view of the Penelope Vocabulary also gives an idea of the Property-Centric approach followed in RDF. In the current implementation, Penelope Vocabulary is maintained in the form of Java Objects and could easily be serialized into other formats, like DAML.

```
<?xml version='1.0'?>
<rdf:RDF
    xmlns:rdfs='http://www.w3.org/2000/01/rdf-schema#'
    xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'>
    <rdfs:Class rdf:about='http://hpl.hp.com/Penelope/Primitive#Iterator'/>
    <rdfs:Class rdf:about='http://hpl.hp.com/Penelope/Primitive#Code'/>
    <rdfs:Class rdf:about='http://hpl.hp.com/Penelope/Primitive#Error'/>
    <rdfs:Class rdf:about='http://hpl.hp.com/Penelope/Primitive#Boolean'/>
    <rdfs:Class rdf:about='http://hpl.hp.com/Penelope/Primitive#Number'/>
    <rdfs:Class rdf:about='http://hpl.hp.com/Penelope/Primitive#Message'/>
    <rdfs:Class rdf:about='http://hpl.hp.com/Penelope/Primitive#Program'/>
    <rdfs:Class rdf:about='http://hpl.hp.com/Penelope/Primitive#Set'/>
    <rdfs:Class rdf:about='http://hpl.hp.com/Penelope/Primitive#Method'/>
    <rdfs:Class rdf:about='http://hpl.hp.com/Penelope/Primitive#Argument'/>
    <rdfs:Class rdf:about='http://hpl.hp.com/Penelope/Primitive#Variable'/>
    <rdfs:Class rdf:about='http://hpl.hp.com/Penelope/Primitive#Array'/>
    <rdfs:Class rdf:about='http://hpl.hp.com/Penelope/Primitive#Class'/>
    <rdfs:Class rdf:about='http://hpl.hp.com/Penelope/Primitive#Void'/>
    <rdfs:Class rdf:about='http://hpl.hp.com/Penelope/Primitive#Nil'/>
    <rdfs:Class rdf:about='http://hpl.hp.com/Penelope/Primitive#UnDefined'/>
    <rdfs:Class rdf:about='http://hpl.hp.com/Penelope/Primitive#Statement'/>
    <rdfs:Class rdf:about='http://hpl.hp.com/Penelope/Primitive#String'/>
    <rdf:Property rdf:about='http://hpl.hp.com/Penelope/Primitive#array-elements'>
        <rdfs:domain rdf:resource='http://hpl.hp.com/Penelope/Primitive#Array'/>
        <rdfs:range rdf:resource='http://www.w3.org/2000/01/rdf-schema#Resource'/>
    </rdf:Property>
    <rdf:Property rdf:about='http://hpl.hp.com/Penelope/Primitive#method-type'>
        <rdfs:domain rdf:resource='http://hpl.hp.com/Penelope/Primitive#Method'/>
        <rdfs:range rdf:resource='http://www.w3.org/2000/01/rdf-schema#Literal'/>
    </rdf:Property>
    <rdf:Property rdf:about='http://hpl.hp.com/Penelope/Primitive#operator'>
        <rdfs:domain rdf:resource='http://hpl.hp.com/Penelope/Primitive#Message'/>
        <rdfs:range rdf:resource='http://hpl.hp.com/Penelope/Primitive#Method'/>
    </rdf:Property>
    <rdf:Property rdf:about='http://hpl.hp.com/Penelope/Primitive#argument-value'>
        <rdfs:domain rdf:resource='http://hpl.hp.com/Penelope/Primitive#Argument'/>
        <rdfs:range rdf:resource='http://www.w3.org/2000/01/rdf-schema#Resource'/>
    </rdf:Property>
    <rdf:Property rdf:about='http://hpl.hp.com/Penelope/Primitive#var-type'>
        <rdfs:range rdf:resource='http://www.w3.org/2000/01/rdf-schema#Class'/>
        <rdfs:domain rdf:resource='http://hpl.hp.com/Penelope/Primitive#Variable'/>
    </rdf:Property>
    <rdfs:Resource rdf:about='http://hpl.hp.com/Penelope/Primitive#Object'/>
    <rdf:Property rdf:about='http://hpl.hp.com/Penelope/Primitive#code-element'>
        <rdfs:domain rdf:resource='http://hpl.hp.com/Penelope/Primitive#Code'/>
        <rdfs:range rdf:resource='http://hpl.hp.com/Penelope/Primitive#Statement'/>
    </rdf:Property>
    <rdf:Property rdf:about='http://hpl.hp.com/Penelope/Primitive#method-parameters'>
        <rdfs:domain rdf:resource='http://hpl.hp.com/Penelope/Primitive#Method'/>
        <rdfs:range rdf:resource='http://www.w3.org/2000/01/rdf-schema#Resource'/>
    </rdf:Property>
    <rdf:Property rdf:about='http://hpl.hp.com/Penelope/Primitive#program-code'>
        <rdfs:domain rdf:resource='http://hpl.hp.com/Penelope/Primitive#Program'/>
        <rdfs:range rdf:resource='http://hpl.hp.com/Penelope/Primitive#Code'/>
    </rdf:Property>
    <rdf:Property rdf:about='http://hpl.hp.com/Penelope/Primitive#iterator-value'>
        <rdfs:domain rdf:resource='http://hpl.hp.com/Penelope/Primitive#Iterator'/>
        <rdfs:range rdf:resource='http://www.w3.org/2000/01/rdf-schema#Resource'/>
    </rdf:Property>
    <rdf:Property rdf:about='http://hpl.hp.com/Penelope/Primitive#class-member'>
        <rdfs:domain rdf:resource='http://hpl.hp.com/Penelope/Primitive#Class'/>
        <rdfs:range rdf:resource='http://hpl.hp.com/Penelope/Primitive#Variable'/>
    </rdf:Property>
    <rdf:Property rdf:about='http://hpl.hp.com/Penelope/Primitive#bool-value'>
        <rdfs:domain rdf:resource='http://hpl.hp.com/Penelope/Primitive#Boolean'/>
        <rdfs:range rdf:resource='http://www.w3.org/2000/01/rdf-schema#Literal'/>
    </rdf:Property>
    <rdf:Property rdf:about='http://hpl.hp.com/Penelope/Primitive#var-value'>
        <rdfs:range rdf:resource='http://www.w3.org/2000/01/rdf-schema#Resource'/>
        <rdfs:domain rdf:resource='http://hpl.hp.com/Penelope/Primitive#Variable'/>
    </rdf:Property>
    <rdf:Property rdf:about='http://hpl.hp.com/Penelope/Primitive#argument-type'>
        <rdfs:domain rdf:resource='http://hpl.hp.com/Penelope/Primitive#Argument'/>
        <rdfs:range rdf:resource='http://www.w3.org/2000/01/rdf-schema#Resource'/>
    </rdf:Property>
    <rdf:Property rdf:about='http://hpl.hp.com/Penelope/Primitive#method-class'>
        <rdfs:domain rdf:resource='http://hpl.hp.com/Penelope/Primitive#Method'/>
        <rdfs:range rdf:resource='http://www.w3.org/2000/01/rdf-schema#Class'/>
```

```
</rdf:Property>
<rdf:Property rdf:about='http://hpl.hp.com/Penelope/Primitive#method-return-type'>
    <rdfs:domain rdf:resource='http://hpl.hp.com/Penelope/Primitive#Method'/>
    <rdfs:range rdf:resource='http://www.w3.org/2000/01/rdf-schema#Resource'/>
</rdf:Property>
<rdf:Property rdf:about='http://hpl.hp.com/Penelope/Primitive#error-message'>
    <rdfs:domain rdf:resource='http://hpl.hp.com/Penelope/Primitive#Error'/>
    <rdfs:range rdf:resource='http://www.w3.org/2000/01/rdf-schema#Literal'/>
</rdf:Property>
<rdf:Property rdf:about='http://hpl.hp.com/Penelope/Primitive#set-elements'>
    <rdfs:domain rdf:resource='http://hpl.hp.com/Penelope/Primitive#Set'/>
    <rdfs:range rdf:resource='http://www.w3.org/2000/01/rdf-schema#Resource'/>
</rdf:Property>
<rdf:Property rdf:about='http://hpl.hp.com/Penelope/Primitive#numeric_value'>
    <rdfs:domain rdf:resource='http://hpl.hp.com/Penelope/Primitive#Number'/>
    <rdfs:range rdf:resource='http://www.w3.org/2000/01/rdf-schema#Literal'/>
</rdf:Property>
<rdf:Property rdf:about='http://hpl.hp.com/Penelope/Primitive#class-name'>
    <rdfs:domain rdf:resource='http://hpl.hp.com/Penelope/Primitive#Class'/>
    <rdfs:range rdf:resource='http://www.w3.org/2000/01/rdf-schema#Literal'/>
</rdf:Property>
<rdf:Property rdf:about='http://hpl.hp.com/Penelope/Primitive#method-name'>
    <rdfs:domain rdf:resource='http://hpl.hp.com/Penelope/Primitive#Method'/>
    <rdfs:range rdf:resource='http://www.w3.org/2000/01/rdf-schema#Literal'/>
</rdf:Property>
<rdf:Property rdf:about='http://hpl.hp.com/Penelope/Primitive#var-name'>
    <rdfs:range rdf:resource='http://www.w3.org/2000/01/rdf-schema#Literal'/>
    <rdfs:domain rdf:resource='http://hpl.hp.com/Penelope/Primitive#Variable'/>
</rdf:Property>
<rdf:Property rdf:about='http://hpl.hp.com/Penelope/Primitive#receiver'>
    <rdfs:domain rdf:resource='http://hpl.hp.com/Penelope/Primitive#Message'/>
    <rdfs:range rdf:resource='http://www.w3.org/2000/01/rdf-schema#Resource'/>
</rdf:Property>
<rdf:Property rdf:about='http://hpl.hp.com/Penelope/Primitive#program-reply-to'>
    <rdfs:domain rdf:resource='http://hpl.hp.com/Penelope/Primitive#Program'/>
    <rdfs:range rdf:resource='http://www.w3.org/2000/01/rdf-schema#Literal'/>
</rdf:Property>
<rdf:Property rdf:about='http://hpl.hp.com/Penelope/Primitive#comment'>
    <rdfs:domain rdf:resource='http://hpl.hp.com/Penelope/Primitive#UnDefined'/>
    <rdfs:range rdf:resource='http://www.w3.org/2000/01/rdf-schema#Literal'/>
</rdf:Property>
<rdf:Property rdf:about='http://hpl.hp.com/Penelope/Primitive#statement-element'>
    <rdfs:domain rdf:resource='http://hpl.hp.com/Penelope/Primitive#Statement'/>
    <rdfs:range rdf:resource='http://hpl.hp.com/Penelope/Primitive#Message'/>
</rdf:Property>
<rdf:Property rdf:about='http://hpl.hp.com/Penelope/Primitive#program-sender'>
    <rdfs:domain rdf:resource='http://hpl.hp.com/Penelope/Primitive#Program'/>
    <rdfs:range rdf:resource='http://www.w3.org/2000/01/rdf-schema#Literal'/>
</rdf:Property>
<rdf:Property rdf:about='http://hpl.hp.com/Penelope/Primitive#method-code'>
    <rdfs:domain rdf:resource='http://hpl.hp.com/Penelope/Primitive#Method'/>
    <rdfs:range rdf:resource='http://hpl.hp.com/Penelope/Primitive#Code'/>
</rdf:Property>
<rdf:Property rdf:about='http://hpl.hp.com/Penelope/Primitive#class-method'>
    <rdfs:domain rdf:resource='http://hpl.hp.com/Penelope/Primitive#Class'/>
```

```
        <rdfs:range rdf:resource='http://hpl.hp.com/Penelope/Primitive#Method'/>
    </rdf:Property>
    <rdf:Property rdf:about='http://hpl.hp.com/Penelope/Primitive#string_value'>
        <rdfs:domain rdf:resource='http://hpl.hp.com/Penelope/Primitive#String'/>
        <rdfs:range rdf:resource='http://www.w3.org/2000/01/rdf-schema#Literal'/>
    </rdf:Property>
    <rdf:Property rdf:about='http://hpl.hp.com/Penelope/Primitive#message-argument'>
        <rdfs:domain rdf:resource='http://hpl.hp.com/Penelope/Primitive#Message'/>
        <rdfs:range rdf:resource='http://hpl.hp.com/Penelope/Primitive#Argument'/>
    </rdf:Property>
    <rdf:Property rdf:about='http://hpl.hp.com/Penelope/Primitive#element-count'>
        <rdfs:domain rdf:resource='http://hpl.hp.com/Penelope/Primitive#Array'/>
        <rdfs:domain rdf:resource='http://hpl.hp.com/Penelope/Primitive#Set'/>
        <rdfs:range rdf:resource='http://hpl.hp.com/Penelope/Primitive#Number'/>
    </rdf:Property>
    <rdf:Property rdf:about='http://hpl.hp.com/Penelope/Primitive#method-impl-class'>
        <rdfs:domain rdf:resource='http://hpl.hp.com/Penelope/Primitive#Method'/>
        <rdfs:range rdf:resource='http://www.w3.org/2000/01/rdf-schema#Class'/>
    </rdf:Property>
    <rdf:Property rdf:about='http://hpl.hp.com/Penelope/Primitive#program-name'>
        <rdfs:domain rdf:resource='http://hpl.hp.com/Penelope/Primitive#Program'/>
        <rdfs:range rdf:resource='http://www.w3.org/2000/01/rdf-schema#Literal'/>
    </rdf:Property>
</rdf:RDF>
```

Bibliography

[1] The Darpa Agent Markup Language. http://www.daml.org/.

[2] The Extensible Markup Language Second Edition. http://www.w3.org/TR/REC-xml.

[3] *On To SmallTalk*. Addion and Wesley, 1998.

[4] Harry Chen, Sovrin Tolia, Craig Sayers, Tim Finin, and Anupam Joshi. Creating Context-Aware Software Agents. *Workshop on Radical Agents Concepts*, January 2002.

[5] James Clarke. XSL Transformations. http://www.w3.org/TR/xslt.

[6] Chris Goad The Behavior Engine. Describing Computation within RDF. *In Proceedings of the International Semantic Web Working Symposium*, 2001.

[7] Synchronized Multimedia Working Group. Synchronized Multimedia Integration Language. http://www.w3.org/TR/REC-smil/.

[8] Jeff Heflin, Raphael Volz, and Jonathan Dale. The Requirements for the Web Ontology Language. http://www.w3.org/TR/webont-req/.

[9] David Huynh, David Karger, and Dennis Quan. Haystack: A Platform for Creating, Organizing and Visualizing Information Using RDF. *In Proceedings of the International Workshop on the Semantic Web*, 2002.

[10] Harumi Kuno, Craig Sayers, and Kevin Wilkinson. Personal Communications.

[11] Ora Lassila and Ralph R. Swick. *Resource Description Framework (RDF) Model and Syntax Specification.* W3C, February 1999.

[12] Brian Mcbride. Jena: Implementing the RDF Model and Syntax Specification. *In Proceedings of the Second International Workshop on the Semantic Web (SemWeb 2001)*, May 2001.

[13] Sun MicroSystems. The Java Programming Language. www.java.sun.com.

[14] M. Rom, a Roy, and H. Campbell. Gaia: Enabling Active Spaces. *In Proceedings of the 9th ACM SIGOPS European Workshop, Kolding, Denmark*, September 2000.

[15] Craig Sayers and Kevin Eshghi. The Case for Generating URI by hashing RDF Content. Hewlett Packard Laboratories, Technical Report HPL-2002-216.

[16] Craig Sayers, Harumi Kuno, and Kevin Wilkinson. Personal Communications.