General-Purpose Computing
on a
Semantic Network Substrate*

Marko A. Rodriguez

Digital Library Research and Prototyping Team

Los Alamos National Laboratory

Los Alamos, New Mexico 87545

original: April 16, 2007 revision: October 7, 2007

Abstract

This article presents a model of general-purpose computing on a semantic network substrate. The concepts presented are applicable to any semantic network representation. However, due to the standards and technological infrastructure devoted to the Semantic Web effort, this article is presented from this point of view. In the proposed model of computing, the application programming interface, the run-time program, and the state of the computing virtual machine are all represented in the Resource Description Framework (RDF). The implementation of the concepts presented provides a practical computing paradigm that leverages the highly-distributed and standardized representational-layer of the Semantic Web.

keywords: Resource Description Framework, Web Ontology Language, Virtual Machines, Object-Oriented Programming, Semantic Web Computing

1 Introduction

This article discusses computing in semantic networks. A semantic network is a directed labeled graph (Sowa, 1991). The thesis of this article is that the state of a computing machine, its low-level instructions, and the executing program can be represented as a semantic network. The computational model that is presented can be instantiated using any semantic network representation. However, given the existence of the Resource Description Framework (RDF) (Manola & Miller, 2004) and the popular Web Ontology Language (OWL) (McGuinness & Harmelen, 2004), this article presents the theory and the application in terms of these constructs.

The computing model that is proposed is perhaps simple in theory, but in application, requires a relatively strong background in the computer sciences. This article discusses a wide breadth of concepts including those from computer architecture, the Semantic Web, and object-oriented programming. In order to accommodate all interested readers, each discipline's concepts will be introduced at a tutorial level in this introduction. The remainder of the article presents a more in-depth practical application of the proposed model. The practical application includes a specification for an RDF virtual machine architecture (RVM) called Fhat (pronounced făt, like "fat") and an RDF programming language designed specifically for that architecture called Neno (pronounced nēnō, like "knee-know").

The introduction to this article is split into three subsections. §1.1 provides a brief introduction to the field of computer architecture in order to elucidate those concepts which

will be of primary interest later in the article. §1.2 discusses the Semantic Web and the RDF semantic network data model. Finally, §1.3 provides an overview of object-oriented programming and its relation to OWL.

1.1 General-Purpose Computing and the Virtual Machine

A general-purpose computer is one that can support any known computation. The meaning of computability and what is required to compute was first developed by Alan Turning in the late 1930s (Turing, 1937). Since then, and with the large leaps in the engineering of computing machines, most computers of today are general-purpose computing machines. The two primary components of a computing machine are the central processing unit (CPU) and the main memory (RAM).

The purpose of the CPU is to perform calculations (i.e. execute algorithms on data). Generally, the CPU reads instructions and data from RAM, performs a calculation, and re-inserts its results back into RAM. Most CPUs maintain a relatively small set of instructions that they can execute (Hennessy & Patterson, 2002). Example instructions include `add`, `sub`, `load`, `store`, `branch`, `goto`, etc. However, these primitive instructions can be composed to perform any computing task desired by the programmer.

Currently, the smallest unit of information in a digital computer is the bit. A bit can either be a 0 or a 1. Bits are combined to form bytes (8-bits) and words (machine architecture dependent). Most desktop machines of everyday use have a 32-bit word and are called 32-bit machines. 32-bits can be used to represent 2^{32} different "things". For example, an unsigned 32-bit integer can represent the numbers 0 to 4,294,967,295. While instructions, like data, are represented as a series of 0s and 1s, it is possible to represent the instructions in a more human readable form. The abstraction above binary machine language is called assembly language (Britton, 2003). For example, the following three assembly instructions

```
load 2, 3
load 1, 2
add 1, 2, 3
store 3, 2
```

instruct the CPU to 1) read the word in the memory cell at memory address 2 in RAM and store it in CPU register 3, 2) read the word at memory address 1 and store it in register 2, 3) add the contents of register 1 and 2 and store the result in register 3, and finally 4) store the word in register 3 into memory address 2 of RAM.

Modern day computer languages are written at a much higher level of abstraction than both machine and assembly language. For instance, the previous instructions could be represented by a single statement as

```
z = y + x
```

where register 1 holds the value of variable `y`, register 2 holds the value of variable `x` and memory address 3 holds the value of variable `z`.

To the modern-day programmer, the low-level CPU instructions are hidden. It is the role of the language compiler to translate the human readable/writeable source code into the machine code of the CPU. Simply stated, a compiler is a computer program that translates information written in one language to another language (Aho, Sethi, & Ullman, 1986). In practice, the compiler translates the human code to a list of CPU instructions that are stored in RAM and executed by the CPU in sequential order as pointed to by the CPU's program counter (PC). In some instances, the compiler will translate the human code to the native language of the CPU (the instruction set of the CPU). In other cases, the compiler will translate the human code to another language for a virtual machine to compute (Craig, 2005). Virtual machine language is called byte-code. A virtual machine is, for all practical purposes, a CPU represented in software, not hardware. However, depending on the complexity of the implementation, a virtual machine can either do a hard implementation (where an exact software replica of hardware is instantiated) or a soft implementation (where more of the hardware components are black-boxed in software).

The virtual machine computes its byte-code instructions by using the underlying hardware CPU's instruction set. Finally, to complete the computation stack, the CPU relies on the underlying physical laws of nature to compute its instructions. The laws of physics drive the CPU from state to state. The evolution of states and its effects on the world is computing.

Perhaps the most popular virtual machine is the Java virtual machine (JVM) (Lindholm & Yellin, 1999) of the Java programming language (Eckel, 2002). The JVM is a piece of software that runs on a physical machine. The JVM has its own instruction set much like a hardware CPU has its own instruction set. The JVM resides in RAM and requires the physical CPU to compute its evolution. Thus, the JVM translates its instructions to the instruction set of the native CPU. The benefit of this model is that irrespective of the underlying hardware CPU, a JVM designed for that CPU architecture can read and process any Java software (i.e. any Java byte-code). The drawback, is that the computation is slower than when instructions are represented as instructions for the native CPU.

1.2   The Semantic Web and RDF

The previous section described the most fundamental aspects of computing. This section presents one of the most abstract levels of computing: the Semantic Web. The primary goal of the Semantic Web effort is to provide a standardized framework for describing resources (both physical and conceptual) and their relationships to one another (Fensel, Hendler, Lieberman, & Wahlster, 2003). This framework is called the Resource Description Framework (RDF) (Manola & Miller, 2004).[1]

RDF maintains two central tenets. The first states that the lowest unit of representation is the Universal Resource Identifier (URI) (Berners-Lee, Fielding, Software, Masinter, & Systems, 2005) and the literal (i.e. strings, integers, floating point numbers, etc.).[2] A URI unambiguously identifies a resource. When two resources share the same URI, they are the same resource. However, note that when two resources do not share the same URI, this does not necessarily mean that they are not the same resource. In practice, URIs are namespaced to ensure that name conflicts do not occur across different organizations (Bray, Hollander, Layman, & Tobin, 2006). For example, the URI `http://www.newspaper.org/Article` can have a different meaning, or connotation, than `http://www.science.net/Article` because they are from different namespaces.[3]

The second tenet of RDF states that URIs and literal values are connected to one another in sets of triples, denoting edges of a directed labeled graph. A triple is the smallest relational fact that can be asserted about the world (Sowa, 1999). For instance, the statement "I am", can be denoted (I, am, I) in triple form. The first element of the triple is called the subject and can be any URI. The second element is called the predicate, and it can also be any URI. Finally, the third element is called the object, and it can be any URI or literal. If U denotes the set of all URIs and L denotes the set of all literals, then an RDF network denoted G can be defined as

$$G \subseteq (U \times U \times (U \cup L)).$$

RDF has attracted commercial and scholarly interest, not only because of the Semantic Web vision, but because RDF provides a unique way of modeling data. This enthusiasm has sparked the development and distribution of various triple-store applications dedicated to the storage and manipulation of RDF networks (Lee, 2004). Some triple-stores can support computations on RDF networks that are on the order of 10^{10} triples (Aasman, 2006). A triple-store is analagous to a relational database. However, instead of representing data in relational tables, a triple-store represents its data as a semantic network. A triple-store provides an interface to an RDF network for the purpose of reading from and writing to the RDF network. The most implemented query language is the SPARQL Protocol and RDF Query Language (SPARQL) (Prud'hommeaux & Seaborne, 2004). SPARQL, loosely, is a hybrid of both SQL (a relational database language) and Prolog (a logic programming language) (Louden, 2003). As an example, the following SPARQL query returns all URIs that are both a type of `CognitiveScientist` and `ComputerScientist`.

---

[1]Note that RDF is a data model, not a syntax. RDF has many different syntaxes like RDF/XML (Manola & Miller, 2004), Notation 3 (N3) (Berners-Lee, 1998), the N-TRIPLE format (Beckett, 2001), and TRiX (Carroll & Stickler, 2004).

[2]There also exist blank or anonymous nodes. There will be no discussion of blank nodes in this article.

[3]For the sake of brevity, prefixes are usually used instead of the full namespace. For instance, `http://www.w3.org/1999/02/22-rdf-syntax-ns#` is prefixed as `rdf:`.

```
SELECT ?x
WHERE {
   ?x <rdf:type> <ComputerScientist> .
   ?x <rdf:type> <CognitiveScientist> }
```

The example SPARQL query will bind the variable `?x` to all URIs that are the subject of the triples with a predicate of `rdf:type` and objects of `ComputerScientist` and `CognitiveScientist`. For the example RDF network diagrammed in Figure 1, `?x` would bind to `Marko`. Thus, the query above would return `Marko`.[4]
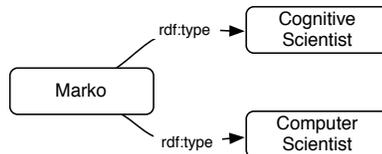

Figure 1: An example RDF network.

The previous query can be represented in its more set theoretic sense as

$$X = \{?x \mid (?x, \texttt{rdf:type}, \texttt{ComputerScientist}) \in G$$
$$\land (?x, \texttt{rdf:type}, \texttt{CognitiveScientist}) \in G\},$$

where X is the set of URIs that bind to `?x` and G is the RDF network represented as an edge list. The above syntax's semantics is "X is the set of all elements `?x` such that `?x` is the head of the triple ending with `rdf:type`, `ComputerScientist` and the head of the triple ending with `rdf:type`, `CognitiveScientist`, where both triples are in the triple list G". Only recently has there been a proposal to extend SPARQL to support writing and deleting triples to and from an RDF network. SPARQL/Update (Seaborne & Manjunath, 2007) can be used to add the fact that `Marko` is also an `rdf:type` of `Human`.

```
INSERT { <Marko> <rdf:type> <Human> . }
```

In a more set theoretic notation, this statement is equivalent to

$$G = G \cup (\texttt{Marko}, \texttt{rdf:type}, \texttt{Human}).$$

The semantics of the previous statement is: "Set the triple list G to the current triple list G unioned with the triple (`Marko`, `rdf:type`, `Human`). Finally, it is possible to remove a triple using SPARQL/Update. For instance,

```
DELETE { <I> <am> <I> . }
```

In set theoretic notation, this is equivalent to

$$G = G \setminus (\texttt{I}, \texttt{am}, \texttt{I}),$$

where the semantics are "set the triple list G to the current triple list G minus the triple (`I`, `am`, `I`)".

---

[4]Many triple-store applications support reasoning about resources during a query (at run-time). For example, suppose that the triple (`Marko`, `rdf:type`, `ComputerScientist`) does not exist in the RDF network, but instead there exist the triples (`Marko`, `rdf:type`, `ComputerEngineer`) and (`ComputerEngineer`, `owl:sameAs`, `ComputerScientist`). With OWL reasoning, `?x` would still bind to `Marko` because `ComputerEngineer` and `ComputerScientist` are the same according to OWL semantics. The RDF computing concepts presented in this article primarily focus on triple pattern matching and thus, beyond direct URI and literal name matching, no other semantics are used.

1.3 Object-Oriented Programming and OWL

OWL is an ontology modeling language represented completely in RDF. In OWL, it is possible to model abstract classes and their relationships to one another as well as to use these models and the semantics of OWL to reason about unspecified relationships. In OWL semantics, if `Human` is a class and there exists the triple (`Marko, rdf:type, Human`), then `Marko` is considered an instance of `Human`. The URI `Human` is part of the ontology-level of the RDF network and the URI `Marko` is part of the instance-level (also called the individual-level) of the RDF network. In OWL, it is possible to state that all `Humans` can have another `Human` as a friend. This is possible by declaring an `owl:ObjectProperty` named `hasFriend` that has an `rdfs:domain` of `Human` and an `rdfs:range` of `Human`. Furthermore, it is possible to restrict the cardinality of the `hasFriend` property and thus, state that a `Human` can have no more than one friend. This is diagrammed in Figure 2.[5]
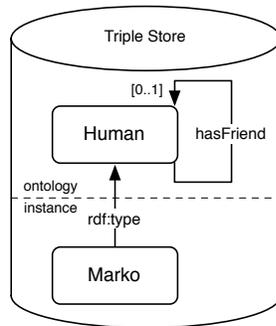

Figure 2: An ontology and an instance is represented in an RDF network.

A class specification in object-oriented programming is called an application programming interface (API) (Sebesta, 2005). OWL ontologies share some similarities to the object-oriented API. However, OWL ontologies also differ in many respects. OWL is a description logic language that is primarily focused on a means by which to reason on RDF data. An object-oriented API is primarily focused on concretely defining classes and their explicit relationships to one another and is thus, more in line with the frames modeling paradigm. Furthermore, OWL ontologies can contain instances (i.e. individuals), allow for multiple inheritance, do not support the unique name assumption, nor the closed world assumption (Oren, Delbru, Gerke, Haller, & Decker, 2007; Wang et al., 2007).

Another aspect of OWL that differs from object-oriented APIs is that object-oriented APIs include the concept of a method. The method is an algorithmic "behavior" that forms the foundation of the evolutionary processes that drive the instances of these classes from state to state. One of the primary purposes of this article is to introduce an OWL ontology for modeling methods and their low-level machine instructions. While process information can be represented in Frame Logic (i.e. F-Logic) (Kifer, Lausen, & Wu., 1995), this article is primarily interested in modeling methods in much the same way that they are represented in modern day object-oriented languages such as Java and C++ and in terms of their syntax, semantics, and low-level representation.

In Java and C++, a method is defined for a class and is used to manipulate the properties (called fields) of an instance of that class. For example,

```
class Human {
  Human hasFriend;
  void makeFriend(Human h) {
    this.hasFriend = h;
  }
}
```

---

[5]In this article, ontology diagrams will not explicitly represent the constructs `rdfs:domain`, `rdfs:range`, nor the `owl:Restriction` anonymous URIs. These URIs are assumed to be apparent from the diagram. For example, the restriction shown as `[0..1]` in Figure 2 is represented by an `owl:Restriction` for the `hasFriend` property where the `maxCardinality` is 1 and `Human` is an `rdfs:subClassOf` of this `owl:Restriction`.

declares that there exists an abstract class called `Human`. A `Human` has one field called `hasFriend`. The `hasFriend` field refers to an object of type `Human`. Furthermore, according to the class declaration, a `Human` has a method called `makeFriend`. The `makeFriend` method takes a single argument that is of type `Human` and sets its `hasFriend` field to the `Human` provided in the argument. The `this` keyword makes explicit that the `hasFriend` field is the field of the object for which the `makeFriend` method was invoked.

In many object-oriented languages, an instance of `Human` is created with the `new` operator. For instance,

```
Human Marko = new Human();
```

creates a `Human` named (referenced as) `Marko`. The `new` operator is analogous to the `rdf:type` property. Thus, after this code is executed, a similar situation exists as that which is represented in Figure 2. However, the ontological model diagrammed in the top half of Figure 2 does not have the `makeFriend` method URI. The relationship between object-oriented programming and OWL is presented in Table 1.

	object-oriented	OWL	example
class specification	API	ontology	Human
object property	field	`rdf:Property`	hasFriend
object method	method		makeFriend
instantiate	new operator	`rdf:type` property	`new`/`rdf:type`

Table 1: The relationship between object-oriented programming, OWL, and the section example.

It is no large conceptual leap to attach a method URI to a class. Currently, there is no strong incentive to provide a framework for representing methods in OWL. RDF was originally developed as a data modeling framework, not a programming environment *per se*. However, in a similar vein, the Web Ontology Language for Services (OWL-S) has been proposed as a web services model to support the discovery, execution, and tracking of the execution of Semantic Web services (Martin et al., 2004; Alesso & Smith, 2005). An OWL-S service exposes a service profile that describes what the service does, a service grounding that describes how to invoke the service, and a service model that describes how the service works. While OWL-S does provide the notion of object-oriented method invocation on the Semantic Web, OWL-S is more at the agent-oriented level and its intended use is for more "client/server" type problems. Another interesting and related idea is to use RDF as a medium for communication between various computing devices and thus, utilize the Semantic Web as an infrastructure for distributed computing (Fensel, 2004). Other object-oriented notions have been proposed within the context of RDF. For instance, SWCLOS (Koide & Kawamura, 2004) and ActiveRDF (Oren et al., 2007) utilize RDF as a medium for ensuring the long-term persistence of an object. Both frameworks allow their respective languages (CLOS and Ruby) to populate the fields of their objects for use in their language environments. Once their fields have been populated, the object's methods can be invoked in their respective programming environments.

1.4 The Contributions of this Article

This article unifies all of the concepts presented hitherto into a framework for computing on RDF networks. In this framework, the state of a computing virtual machine, the API, and the low-level instructions are all represented in RDF. Furthermore, unlike the current programming paradigm, there is no stack of representation. The lowest level of computing and the highest level of computing are represented in the same substrate: URIs, literals, and triples.

This article proposes the concept of OWL APIs, RDF triple-code, and RDF virtual machines (RVM). Human readable/writeable source code is compiled to create an OWL ontology that abstractly represents how instructions should be united to form instruction sequences.[6] When objects and their methods are instantiated from an OWL API, RDF

---

[6]While OWL has many features that are useful for reasoning about RDF data, the primary purpose of OWL

triple-code is created. RDF triple-code is analogous to virtual machine byte-code, but instead of being represented as bits, bytes, and words, it is represented as URIs and triples. In other words, a piece of executable software is represented as a traversable RDF network. The RVM is a virtual machine whose state is represented in RDF. The RVM's stacks, program counter, frames, etc. are modeled as an RDF network. It is the role of the RVM to "walk" the traversable RDF triple-code and compute.

In summary, software is written in human readable/writeable source code, compiled to an OWL API, instantiated to RDF triple-code, and processed by a computing machine whose state is represented in RDF. However, there is always a homunculus. There is always some external process that drives the evolution of the representational substrate. For the JVM, that homunculus is the hardware CPU. For the hardware CPU, the homunculus is the physical laws of nature. For the RVM, the homunculus is some host CPU whether that host CPU is another virtual machine like the JVM or a hardware CPU. Table 2 presents the different levels of abstraction in computing and how they are represented by the physical machine, virtual machine, and proposed RDF computing paradigms.

level	machine paradigm	virtual machine paradigm	RDF paradigm
high-level code	source code	source code	source code
machine code	native instructions	byte-code	triple-code
instruction units	bits	bits	URIs and literals
machine state	hardware	software	RDF
machine execution	physics	hardware	software

Table 2: The various levels of abstraction in current and proposed computing paradigms.

2   A High-Level Perspective

Assume there exists an RDF triple-store. Internal to that triple-store is an RDF network. That RDF network is composed of triples. A triple is a set of three URIs and/or literals. Those URIs can be used as a pointer to anything. This article presents a model of computation that is represented by URIs and literals and their interrelation to one another (triples). Thus, computation is represented as an RDF network. Figure 3 presents a high-level perspective on what will be discussed throughout the remainder of this article. What is diagrammed in Figure 3 is a very compartmentalized model of the components of computing. This model is in line with the common paradigm of computer science and engineering. However, less traditional realizations of this paradigm can remove the discrete levels of representation to support multi-level interactions between the various computing components since all the components are represented in the same RDF substrate: as URIs, literals, and triples.

Figure 3 shows 6 primary components. Two of these components are at the ontological level of the RDF network, two are at the instance level of the RDF network, and two are at the machine level external to the RDF network. While there are many benefits that emerge from this computing model that are currently seen and as of yet unseen, established interesting aspects are enumerated below.

1. The total address space of the RVM is the space of all URIs and literals. In the RVM model of computing, the RVM state has no concept of the underlying hardware CPU's address space because instructions and data are represented in RDF. This idea is discussed in §3.1.

2. The Semantic Web is no longer an information gathering infrastructure, but a distributed information processing infrastructure (the process can move to the data, the data doesn't have to move to the process). An RVM can be "GETed" from a web-server as an RDF/XML document or "SELECTed" from an RDF triple-store. RDF programs and RVM states are "first-class" web-entities. The ramifications of this is that an RVM can move between triple-store environments and can compute on local

---

with respect to the concepts presented in this article is to utilize OWL for its ability to create highly restricted data models. These restricted models form the APIs and ensure that instance RDF triple-code can be unambiguously generated by an RVM.
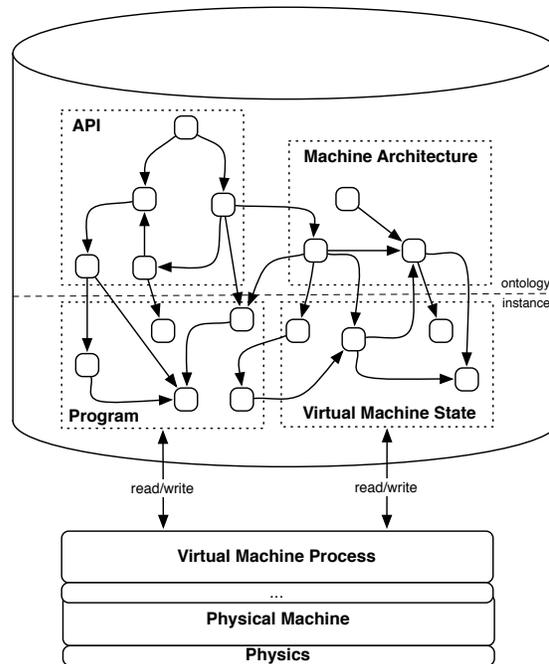
Figure 3: A high-level perspective of the Semantic Web computing environment.

data sets without requiring moving the data to the processor. This idea is discussed in §4.1.

3. This model maintains the "write once, run anywhere" paradigm of the JVM. The RVM model ensures that human readable/writeable source code is compiled down to an intermediate language that is independent of the underlying hardware CPU executing the RVM process.

4. Languages built on a semantic network substrate can have unique constructs not found in other languages (e.g. inverse field referencing, multi-instance fields, field querying, etc.). While it is theoretically possible to add these constructs to other languages, they are not provided in the core of the languages as these languages do not have an underlying semantic network data model. These novel language constructs are discussed in §3.2.

5. Currently, there already exists an infrastructure to support the paradigm (triple-stores, ontology modeling languages, query languages, etc.) and thus, requires very little investment by the community. The primary investment is the development of source-to-OWL API compilers, RVMs, and the standardization of RDF triple-code and RVM distribution/security protocols.

6. An RVM can be engineered at any level of complexity. It is possible to move the complexity to the software implementing the RVM process to ease machine architecture development and speed up computing time. This idea is discussed in §4.3.

7. In this model, language reflection exists at the API, software, and RVM level (everything is represented in RDF). This idea is discussed in §4.2.

2.1   The Ontological Level

The ontological level of the RDF network diagrammed in Figure 3 is represented in OWL. This subsection will discuss the two primary ontological components: the API, and the RVM architecture.

2.1.1   The API

OWL supports the specification of class interactions. However, class interactions are specified in terms of property relationships, not method invocations. OWL has no formal way of specifying class behaviors (i.e. methods). However, in OWL, it is possible to define method and instruction classes and formally specify restrictions that dictate how instructions should be interrelated within a method. The method and instruction ontology presented in this article makes RDF a programming framework and not just a data modeling framework.

2.1.2   The Machine Architecture

The RDF machine architecture is modeled in OWL. The machine architecture ontology is an abstract description of an instance of a particular RVM. Depending on the level of abstraction required, different machine architectures can be implemented at varying levels of detail.

2.2   The Instance Level

The instance level of an RDF network is constrained by the requirements specified in the ontological level of the RDF network. This subsection will present the two components of the instance layer of the diagram in Figure 3.

2.2.1   The Program

An API abstractly defines a software application. When an API is instantiated, instance RDF triple-code is created. Triple-code represents the instructions used by an RVM to compute.

2.2.2   The Virtual Machine

An instance of the machine architecture is an RDF virtual machine (RVM). The purpose of the RVM is to represent its state (stacks, program counter, etc.) in the same RDF network as the triple-code instructions. However, the RDF-based RVM is not a "true" computer. The RVM simply represents its state in RDF. The RVM requires a software implementation outside the triple-store to compute its instructions. This requires the machine level discussed next.

2.3   The Machine Level

The machine level is where the actual computation is executed. An RDF network is a data structure. RDF is not a processor in the common sense—it has no way of evolving itself. In order to process RDF data, some external process must read and write to the RDF network. The reading and writing of the RDF network evolves the RVM and the objects on which it is computing. This section discusses the machine level that is diagrammed in Figure 3.

2.3.1   The Virtual Machine Process

The virtual machine process is represented in software on a particular host machine. The RVM processor must be compatible with both the triple-store interface (e.g. SPARQL/Update) and the underlying host machine. The RVM's host machine can be the physical machine (hardware CPU) or another virtual machine. For instance, if the RVM's machine process is implemented in the Java language, then the machine process runs in the JVM. This is diagrammed in Figure 3 by the . . . component in between the virtual machine process and the physical machine.

2.3.2   The Physical Machine

The physical machine is the actual hardware CPU. The RVM implementation translates the RDF triple-code to the host machine's instruction set. For example, if the RVM process is running on the Intel Core Duo, then it is the role of the RVM process to translate the RDF triple-code to that specified by the Intel Core Duo instruction set. Thus, portability

of this architectural model relies on a per host implementation of the RVM. Finally, to complete the computational stack, the laws of physics compute the hardware CPU. Much like the RDF representation of the RVM is a "snap-shot" representation of a computation, the hardware CPU is a silicon/electron "snap-shot" representation of a computation.

3   The Neno Language

This section presents the specification of a programming language designed to take advantage of a pure RDF computing environment. This language is called Neno. Neno is a high-level object-oriented language that is written in a grammar similar to other object-oriented languages such as Java and C++. However, Neno provides some functionality that is not possible with other languages (i.e. not explicit in the constructs of other object-oriented languages). This functionality is not due to the sophistication of the Neno language, but instead, is due to the fact that it is written for an RDF substrate and thus, can take advantage of the flexibility of RDF and its read/write interfaces. For this reason, Neno is in a class of languages that is coined semantic network programming languages. The Ripple programming language is another such semantic network programming language (Shinavier, 2007). Both Neno and Ripple are Turing complete and thus, can perform any classical (non-quantum) computation.

Neno source code is written in human readable/writeable plain-text like the source code of many other high-level programming languages. Neno source code is compiled by a NenoFhat compiler. The NenoFhat compiler compiles Neno source code to a Fhat OWL API. The Fhat OWL API is analogous to the `jar` file of Java. A Fhat RVM instantiates (loads) aspects of the API into the instance layer of the RDF network. This instantiated aspect of the API is executable RDF triple-code. A Fhat RVM processes the triple-code and thus, computes. The analogies between the Neno and Java components are presented in Table 3.

artifact	Neno	Java
source code	AClass.neno	AClass.java
compiler	nenofhat	javac
API	AClass.owl	AClass.class
virtual machine	fhat	java
program	RDF network	JVM memory

Table 3: The mapping between Neno and Java components.

The following examples will only namespace those entities that are not within the namespace `http://neno.lanl.gov`. Thus, the default namespace is `http://neno.lanl.gov` (prefixed as `neno`). The Neno programming language is engineered to be in compliance with OWL and the XML Schema Definition (XSD) namespaces. OWL provides the concept of classes, inheritance, datatype and class properties, and property restrictions. However, Neno restricts its compiled Fhat OWL APIs to single-parent classes (i.e. multiple-inheritance is not supported) and holds the closed world assumption (i.e. only properties that are stated in the ontology can be computed on in a Neno object). This is similar to what is assumed in Java. XSD provides the specification for the literal data types (e.g. string, integer, float, double, date, time, etc.). The XSD URI namespace prefix is `xsd`.

The lexicon that will be used to express the following concepts is drawn from object-oriented programming, not OWL. OWL parlance will only be used when completely describing the "back-end" of a particular aspect of the language. Table 4 states the relationship between OWL terms and object-oriented programming terms.

3.1   The Universally Unique Identifier Address Space

Throughout the remainder of this article, Universally Unique Identifiers (UUIDs) will be continually used (Leach, 2005). The set of all UUIDs is a subset of the set of all URIs. A UUID is a 128-bit (16-byte) string that can be created in disparate environments with a near zero probability of ever being reproduced. To understand the number of UUIDs that are possible at 128-bits, it would require 1 trillion unique UUIDs to be created every

OWL	object-oriented languages
`owl:Class`	Class
`neno:Method`	Method
`rdf:Property`	Field
subject of `rdf:type`	Object

Table 4: The mapping between the terms in OWL and object-oriented programming.

nanosecond for 10 billion years to exhaust the space of all possible UUIDs.[7] A UUID can be represented as a 36 character hexadecimal string. For example, 6c3f8afe-ec3d-11db-8314-0800200c9a66, is a UUID. The hexadecimal representation will be used in all the following examples. However, for the sake of brevity, since 36 characters is too lengthy for the examples and diagrams, only the first 8 characters will be used. Thus, 6c3f8afe-ec3d-11db-8314-0800200c9a66 will be represented as 6c3f8afe. Furthermore, UUIDs, when used as URIs are namespaced as

$$\texttt{urn:uuid:6c3f8afe-ec3d-11db-8314-0800200c9a66}$$

and for diagrams and examples, is abbreviated as `urn:uuid:6c3f8afe`.

When Neno source code is compiled to Fhat triple-code, a UUID is created for nearly everything; every instruction class and instruction instance is identified by a UUID. When a Fhat is instantiated, a UUID is created for all the `rdfs:Resources` that compose the machine (i.e. stacks, frames, etc.). In typical programming environments, the programming language and its computing machine are constrained by the size of RAM (and virtual memory with most modern day operating systems). For a 32-bit machine, the maximum size of RAM is approximately 4 gigabytes. This means that there are only 2^{32} possible addresses and thus, words in RAM. However, for Neno, no such constraints exist. The space of all UUIDs is the address space of a Fhat RVM (more generally, the space of all URIs and literals is the address space). Fhat does not use RAM for storing its data and instructions, Fhat uses an RDF network. Thus, Fhat does not have any hard constraint on how much memory it "allocates" for its processing.

3.2 Class Declarations in Neno Source Code

Neno source code has a grammar that is very similar to other object-oriented languages. For instance, suppose the following simple class written in the Java programming language:

```
package gov.lanl.neno.demo;

import java.lang.*;
import java.util.*;

public class Human {
  private String hasName;
  private ArrayList<Human> hasFriend;

  public Human (String n) {
    this.hasName = n;
  }

  public void makeFriend(Human h) {
    if(h != this)
      this.hasFriend.add(h);
  }

  public void setName(String n) {
    this.hasName = n;
  }
}
```

---

[7]This fact was taken from Wikipedia at http://en.wikipedia.org/wiki/UUID.

The `Human` class has two fields named `hasName` and `hasFriend`. The field `hasName` takes a value of `String` (or `java.lang.String` to be more specific) and `hasFriend` takes a value of `Human`. The `Human` class has one constructor and one method. A constructor is used to create an object and is a type of method. In Java, a constructor tells the JVM to allocate memory for the object on the heap (i.e. an object "pool") and set the object's field values according to the statements in the body of the constructor. The constructor for `Human` takes a `String` called `n` and creates a new `Human` instance called an object. The `Human` constructor sets that object's `hasName` field to `n`. The `Human` method is called `makeFriend`. This method takes a `Human` with variable name `h` as an argument. If the object referenced by `h` is not the `Human` for which this method was invoked, then the object for which this method was called has `h` added to its `hasFriend` field. Note, that unlike the example in Figure 2, it is possible for a `Human` object to have multiple friends because of the use of the `ArrayList<Human>`.[8]

The Neno programming language is similar to Java. The following source code demonstrates how to declare nearly the same class in Neno.[9]

```
prefix owl: <http://www.w3.org/2002/07/owl>;
prefix xsd: <http://www.w3.org/2001/XMLSchema>;
prefix demo: <http://neno.lanl.gov/demo>;

owl:Thing demo:Human {
  xsd:string hasName[1];
  demo:Human hasFriend[0..*];

  !Human(xsd:string n) {
    this.hasName = n;
  }

  makeFriend(demo:Human h) {
    if(h != this)
      this.hasFriend =+ h;
  }

  setName(xsd:string n) {
    this.hasName = n;
  }
}
```

While the `Human` class declaration in Java and in Neno are nearly identical, there are a few constructs that make the two languages different. For one, instead of "importing" packages, in Neno, namespaces are declared and ontologies are imported.[10] To ease namespace declarations, prefixes are used (e.g. `owl`, `xsd`, and `demo`). All constructors are denoted by the class name prefixed by the ! symbol. Similarly, though not in the above example, all destructors are denoted by the class name prefixed by the ~ symbol. Notice that all datatype primitives (e.g. `xsd:string`) are from the XSD namespace. The Fhat RVM is engineered specifically for these datatypes. Perhaps the most unique aspect of the Neno language is the cardinality restriction specifier in the field declaration (e.g. `[0..1]`). Because Neno was designed for a semantic network substrate, there is nothing that prevents the same property (i.e. field) to point to multiple different URIs. In order to demand that there exist no more than one field, the `[0..1]` notation is used. Note that `demo:Human` is an `rdfs:subClassOf owl:Thing` as specified by the `owl:Thing demo:Human` class description. Class inheritance is specified by the prefix to the declaration of the class name. Note that in Neno, a class can only have a single parent even though OWL supports multiple-inhertiance.[11] Furthermore, note that all class properties have a universal restriction on the class or datatype value. For example, in the above `demo:Human` class, the `hasName` property must have an `xsd:string` value and all `hasFriend` properties must have `demo:Human` values.

---

[8] Java generics as represented by the $< >$ notation is supported by Java 1.5+.

[9] When there are no ambiguities in naming, the class declaration can be written without prefixes.

[10] Note that languages such as Java and C++ do maintain the concept of package namespaces.

[11] This constraint does not apply to `owl:Restrictions` as Neno classes utilize `owl:Restrictions` to make explicit property restrictions. Thus, excluding `owl:Restriction` subclassing, a Neno object class can only be the subclass of a single class.

In order to demonstrate the relationship between Neno source code and its compiled OWL API, the following simple class example is presented. The class

```
prefix owl: <http://www.w3.org/2002/07/owl>;
prefix xsd: <http://www.w3.org/2001/XMLSchema>;
prefix demo: <http://neno.lanl.gov/demo>;

owl:Thing demo:Example {
  xsd:integer t[0..1];

  test(xsd:integer n) {
    for(xsd:integer i=0; i < n; i++) {
      this.t = this.t + 1;
    }
  }
}
```

has the following OWL RDF/XML representation:

```
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#">
  <owl:Ontology rdf:about="http://neno.lanl.gov"/>
  <owl:Ontology rdf:about="http://neno.lanl.gov/demo">
    <owl:imports rdf:resource="http://neno.lanl.gov"/>
  </owl:Ontology>
  ...
  <!-- A PUSHVALUE INSTRUCTION -->

  <owl:Class rdf:about="http://neno.lanl.gov/demo#2271ea72-877c-4090-9f89-...">
    <rdfs:subClassOf rdf:resource="http://neno.lanl.gov#PushValue"/>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty rdf:resource="http://neno.lanl.gov#hasValue"/>
        <owl:allValuesFrom>
          <owl:Class rdf:about="http://neno.lanl.gov/demo#9792cc3c-5600-4660-..."/>
        </owl:allValuesFrom>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty rdf:resource="http://neno.lanl.gov#nextInst"/>
        <owl:allValuesFrom>
          <owl:Class rdf:about="http://neno.lanl.gov/demo#a80ba54c-5344-4df1-..."/>
        </owl:allValuesFrom>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>

  <!-- THE PUSHED VALUE -->

  <owl:Class rdf:about="http://neno.lanl.gov/demo#9792cc3c-5600-4660-bc1f-...">
    <rdfs:subClassOf rdf:resource="http://neno.lanl.gov#LocalDirect"/>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:hasValue rdf:datatype="http://www.w3.org/2001/XMLSchema#integer"
        >1</owl:hasValue>
        <owl:onProperty rdf:resource="http://neno.lanl.gov#hasURI"/>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>

  <!-- THE NEXT INSTRUCTION AFTER THE PUSHVALUE INSTRUCTION: AN ADD INSTRUCTION -->

  <owl:Class rdf:about="http://neno.lanl.gov/demo#a80ba54c-5344-4df1-91a0-...">
    <rdfs:subClassOf rdf:resource="http://neno.lanl.gov#Add"/>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty rdf:resource="http://neno.lanl.gov#hasLeft"/>
        <owl:allValuesFrom rdf:resource="http://neno.lanl.gov/demo#4c715d16-b6e6-..."/>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty rdf:resource="http://neno.lanl.gov#hasRight"/>
        <owl:allValuesFrom rdf:resource="http://neno.lanl.gov/demo#fdde7f6f-b9c0-..."/>
      </owl:Restriction>
```

```
      </rdfs:subClassOf>
      <rdfs:subClassOf>
        <owl:Restriction>
          <owl:onProperty rdf:resource="http://neno.lanl.gov#nextInst"/>
          <owl:allValuesFrom rdf:resource="http://neno.lanl.gov/demo#e3b8a797-849b-..."/>
        </owl:Restriction>
      </rdfs:subClassOf>
    </owl:Class>
  ...
</rdf:RDF>
```

The most important idea to take away from the above Fhat OWL API subset is that the role of the compiler is to generate UUID-named instruction classes that are subclasses of particular Fhat instructions (e.g. `PushValue`). These generated instruction classes have `owl:Restrictions` on them that ensure that instances of these classes are connected to one another in an unambiguous way (e.g. `owl:Restrictions` on their respective `nextInt` property) and that their operand values are made explicit (e.g. `owl:Restrictions` on their respective operand properties). This unambiguous instantiation is the RDF triple-code that is created when a Fhat RVM instantiates the API.

For example, in the above Fhat OWL API snippet, any `demo:2271ea72 PushValue` instruction instance must have one and only one `hasValue` property. The value of that property must be a `demo:9792cc3c LocalDirect` value with a `hasURI` property value of `"1"^^<xsd:integer>`. The instance of `demo:2271ea72` must also have a `nextInst` property that is of `rdf:type demo:a80ba54c`, where `demo:a80ba54c` is an `rdfs:subClassOf Add`. An instance of this `demo:a80ba54c Add` instruction instructs the Fhat RVM to add its `hasLeft` operand and its `hasRight` operands together. This `demo:a80ba54c Add` also has a `nextInst` property value that must be an instance of `demo:e3b8a797`. Though not shown, the `demo:e3b8a797` is an `rdfs:subClassOf Set`. In this way, through strict `owl:Restrictions`, the flow of triple-code can be generated in an unambiguous manner by the Fhat RVM.

The remainder of this section will go over the more salient aspects of the Neno programming language.

3.2.1 Declaring Namespaces

Namespaces promote the distributed nature of the Semantic Web by ensuring that there are no URI name conflicts in the ontologies and instances of different organizations (Bray et al., 2006). The Java language has a similar construct called packaging. The package specification in Java supports organizational namespacing. Neno supports the prefixing of namespaces. For example, `demo:Human` resolves to

<div align="center">

`http://neno.lanl.gov/demo#Human`.

</div>

3.2.2 Datatypes

Fhat is engineered to handle `xsd:anySimpleType` and provides specific support for any of its derived types (Biron & Malhotra, 2004). The XSD namespace maintains, amongst others: `xsd:string`, `xsd:double`, `xsd:integer`, `xsd:date`, etc. Example operations include,

```
"neno"^^xsd:string + "fhat"^^xsd:string
"2007-11-30"^^xsd:date < "2007-12-01"^^xsd:date
"1"^^xsd:integer - "0"^^xsd:integer
```

Neno has low-level support for high-level datatype manipulations such as string concatenation, data and time comparisons, date incrementing, etc. Exactly what operations are allowed with what datatypes will be discussed later when describing the Fhat instruction set.

3.2.3 The `this` Variable

The `this` variable is used in many object-oriented languages to specify the field to be accessed or the method to be invoked. All methods inherently have `this` as a variable they can use. The same construct exists in Neno with no variation in meaning.

3.2.4 Field Cardinality

While Neno is an object-oriented language, it is also a semantic network programming language. Neno is more in line with the concepts of RDF than it is with those of Java and C++. One of the major distinguishing features of an object in Neno is that objects can have multi-instance fields. This means that a single field (predicate) can have more than one value (object). For instance, in Java

```
Human marko = new Human("Marko Rodriguez");
marko.setName("Marko Antonio Rodriguez");
```

will initially set the `hasName` field of the `Human` object referenced by the variable name `marko` to "Marko Rodriguez". The invocation of the `setName` method of `marko` will replace "Marko Rodriguez" with "Marko Antonio Rodriguez". Thus, the field `hasName` has a cardinality of 1. All fields in Java have a cardinality of 1 and are universally quantified for the specified class (though taxonomical subsumption is supported).

In Neno, it is possible for a field to have a cardinality greater than one. In Neno, when a class' fields are declared, the cardinality specifier is used to denote how many properties of this type are allowed for an instance of this class. Thus, in the Neno code at the start of this section,

```
 xsd:string hasName[1];
```

states that any `Human` object must have one and only one field (property) called `hasName` and that `hasName` field points to some `xsd:string`. Therefore, it is illegal for the Fhat RVM to add a new `hasName` property to the class `marko`. The original property must be removed before the new property can be added. The general grammar for field restrictions in Neno is `[# (..(# | *))]`, where # refers to some integer value.

Neno does not adopt any of the OWL semantics regarding cardinality and "semantically distinct" resources. The `owl:sameAs` relationship between resources is not considered when determining the cardinality of a property and thus, only the explicit number of properties (explicit triples) of a particular type (predicate) are acknowledged by the NenoFhat compiler and Fhat RVM.

3.2.5 Handling Fields

Neno provides the following field and local variable operators: `=+`, `=-`, `=/`, and `=`. These operators are called "set plus", "set minus", "set clear", and "set", respectively. The definition of these operators is made apparent through examples that demonstrate their use. For instance, from the class declarations above, the `Human` class has the field `hasFriend`. For the Java example, the `hasFriend` field can have more than one `Human` value only indirectly through the use of the `ArrayList<Human>` class. In Neno, no `ArrayList<Human>` is needed because a field can have a cardinality greater than 1. The cardinality specifier `[0..*]` states that there are no restrictions on the number of friends a `Human` can have. In order to add more friends to a `Human` object, the `=+` operator is used. If the `Human` instance has the URI `urn:uuid:2db4a1d2` and the provided `Human` argument has the URI `urn:uuid:47878dcc` then the `=+` operator instructs Fhat to execute

```
INSERT { <urn:uuid:2db4a1d2> <demo:hasFriend> <urn:uuid:47878dcc> .}
```

on the triple-store. On the other hand, if the `=` operator was used, then Fhat would issue the following commands to the triple-store:

```
DELETE { <urn:uuid:2db4a1d2> <demo:hasFriend> ?x .}
INSERT { <urn:uuid:2db4a1d2> <demo:hasFriend> <urn:uuid:47878dcc> .}
```

For a multi-instance field, the `=` is a very destructive operator. For a `[0..1]` or `[1]` field, `=` behaves as one would expect in any other object-oriented language. Furthermore, for a `[0..1]` or `[1]` field, `=+` is not allowed as it will cause the insertion of more than one property of the same predicate.

In order to control the removal of fields from a multi-instance field, the `=-` and `=/` operators can be used. For example, suppose the following method declaration in Neno

```
makeEnemy(Human h) {
  this.hasFriends =- h;
}
```

The `makeEnemy` method will remove the `Human` object identified by the variable name `h` from the `hasFriend` fields. If the `h` variable is a reference to the URI `urn:uuid:4800e2c2`, then at the Fhat level, Fhat will execute the following command on the triple-store:

```
DELETE { <urn:uuid:2db4a1d2> <demo:hasFriend> <urn:uuid:4800e2c2> .}
```

Finally, assume that there is a rogue `Human` that wishes to have no friends at all. In order for this one man army to sever his ties, the = / operator is used. Assume the following overloaded method declaration for a `Human`.

```
makeEnemy() {
  this.hasFriends =/;
}
```

The above statement statement would have Fhat execute the following delete command on the triple-store:

```
DELETE { <urn:uuid:2db4a1d2> <demo:hasFriend> ?human }
```

3.2.6   Field Querying

In many cases, a field (i.e. property) will have many instances. In computer programming terms, fields can be thought of as arrays. However, these "arrays" are not objects, but simply greater than one cardinality fields. In Java, arrays are objects and high-level array objects like the `java.util.ArrayList` provide functions to search an array. In Neno, there are no methods that support such behaviors since fields are not objects. Instead, Neno provides language constructs that support field querying. For example, suppose the following method

```
boolean isFriend(Human unknown) {
  if(this.hasFriend =? unknown) {
    return true;
  }
  else {
   return false;
  }
 }
```

In the above `isFriend` method, the provided `Human` argument referenced by the variable name `unknown` is checked against all the `hasFriend` fields. Again, the `owl:sameAs` property is not respected and thus, "sameness" is determined by exact URIs. The `=?` operator is a conditional operator and thus, always returns either `"true"`$^{\wedge\wedge}$`xsd:boolean` or `"false"`$^{\wedge\wedge}$`xsd:boolean`. At the Fhat level, if `this` references the UUID `urn:uuid:2d386232` and `unknown` references `urn:uuid:75e05c12`, then the Fhat RVM executes the following query on the triple-store:

```
ASK { <urn:uuid:2d386232> <demo:hasFriend>  <urn:uuid:75e05c12> . }
```

Similarly, imagine the following method,

```
boolean isFriendByName(Human unknown) {
  if(this.hasFriend.hasName =? unknown.hasName) {
    return true;
  }
  else {
   return false;
  }
 }
```

Assuming the same UUID references for `this` and `unknown` from previous examples, the `=?` operation would have the Fhat execute the following query on the RDF network

```
ASK { <urn:uuid:2d386232> <demo:hasFriend>  ?x .
      ?x <demo:hasName> ?y .
      <urn:uuid:75e05c12> <demo:hasName> ?y }
```

Again, there is no reasoning involved in any of these triple-store operations; only "raw" triple and URI/literal matching is used.

3.2.7 Looping and Conditionals

Looping and conditionals are nearly identical to the Java language. In Neno, there exists the `for`, `while`, and `if/else` constructs. For example, a `for` statement is

```
for(xsd:integer i = "0"^^xsd:integer; i<"10"^^xsd:integer; i++)
{ /* for block */ }
```

a `while` statement is

```
while(xsd:integer i < "10"^^xsd:integer)
  { /* while block */ }
```

and an `if/else` statement is

```
if(xsd:integer i < "10"^^xsd:integer)
  { /* if block */ }
  { /* else block */}
```

It is important to note that these statements need not have the literal type specifier (e.g. `xsd:integer`) on every hardcoded literal. The literal type can be inferred from its context and thus, is automatically added by the compiler. For example, since `i` is an `xsd:integer`, it is assumed that 10 is also.

3.2.8 Field Looping

In many cases it is desirable to loop through all the resources of a field for the purposes of searching or for manipulating each resource. For instance, suppose the following `Human` method:

```
namelessFaces() {
  for(Human h : this.hasFriend) {
    h.hasName = "..."^^xsd:string;
  }
  for(xsd:integer i=0; i<this.hasFriend*; i++) {
    Human h = this.hasFriend[i];
    h.hasName = "."^^xsd:string;
  }
 }
```

The above `namelessFaces` method demonstrates two types of field looping mechanisms offered by Neno. The first is analogous to the Java 1.5 language specification. With the first for loop, the variable `h` is set to a single `hasFriend` of `this`. The second for loop uses the index `i` that goes from index 0 to the size of the "array" (`this.hasFriend*`). The `*` notation in this context returns the number of `hasFriend` properties of the `this` object. In other words `*` returns the cardinality of the `this.hasFriend` field.

Finally, as field values are not stored in a vector, but instead as an unordered set, the field "arrays" in Neno are not guaranteed to be ordered. Thus, `this.hasFriend[1]` may not be the same value later in the code. Ordering is dependent upon the triple-store's indexing algorithm and stability of a particular order is dependent upon how often re-indexing occurs in the triple-store. It is worth noting that higher-order classes can be created such as specialized `rdf:Seq` and `rdf:List` classes to provided ordered support for arrays.

3.2.9   Type Checking

The `typeof` operator can be used to determine the class type of a URI. For instance, the following statement,

```
xsd:boolean isType = urn:uuid:2db4a1d2 typeof Human
```

would return `true` if `urn:uuid:2db4a1d2` is `rdf:type` `Human` or `rdf:type` of some class that is an `rdfs:subClassOf` `Human`. Also,

```
xsd:boolean isType = urn:uuid:2db4a1d2 typeof rdfs:Resource
```

always returns `true`. Thus, RDFS subsumption semantics are respected and thus, Neno respects the subclassing semantics employed by modern objected-oriented languages. Similarly the `typeOf?` operator returns the type of the resource. For instance,

```
xsd:anyURI type = urn:uuid:2db4a1d2 typeof?
```

returns `http://neno.lanl.gov/demo#Human`.

3.2.10   Inverse Field Referencing

In object-oriented languages the "dot" operator is used to access a method or field of an object. For instance, in `this.hasName`, on the left of the "dot" is the object and on the right of the "dot" is the field. Whether the right hand side of the operator is a field or method can be deduced by the compiler from its context. If `this` resolves to the URI `urn:uuid:2db4a1d2`, then the following Neno code

```
Human h[0..*] = this.hasFriend;
```

would instruct Fhat to execute the following query:

```
SELECT ?h
  WHERE { <urn:uuid:2db4a1d2> <demo:hasFriend> ?h . }
```

According to the previous query, everything that binds to `?h` will be set to the variable h. The above query says "locate all `Human` `hasFriend`s of `this` object." However, Neno provides another concept not found in other object-oriented languages called the "dot dot" operator. The "dot dot" operator provides support for what is called inverse field referencing (and inverse method invocation discussed next). Assume the following line in some method of some class,

```
Human h[0..*] = this..hasFriend;
```

The above statement says, "locate all `Human`s that have `this` object as their `hasFriend`." At the Fhat level, Fhat executes the following query on the triple-store:

```
SELECT ?h
  WHERE { ?h <demo:hasFriend> <urn:uuid:2db4a1d2> .}
```

Furthermore, if the statement is

```
Human h[0..3] = this..hasFriend;
```

Fhat would execute:

```
SELECT ?h
  WHERE { ?h <demo:hasFriend> <urn:uuid:2db4a1d2> .} LIMIT 3
```

3.2.11   Inverse Method Invocation

Like inverse field referencing, inverse method invocation is supported by Neno. Inverse method invocation will invoke all the methods that meet a particular requirement. For instance,

```
this..hasFriend.makeEnemy(this);
```

will ensure that all objects that have `this` as their friend are no longer friends with `this`.

3.2.12   Variable Scoping

Variable scoping in Neno is equivalent to Java. For example, in

```
xsd:integer a = "11"^^xsd:integer;
if(a < "10"^^xsd:integer) {
  xsd:integer b = "2"^^xsd:integer;
}
else {
  xsd:integer c = "3"^^xsd:integer;
}
```

the true and false block of the `if` statement can read the variable `a`, but the true block can not read the `c` in the false block and the false block can not read the `b` in the true block. Also, methods are out of scope from one another. The only way methods communicate are through parameter passing, return values, and object manipulations.

3.2.13   Constructors and Destructors

Constructors and destructors are used in object-oriented languages to create and destroy object, respectively. The concept of a constructor in Neno is similar to that of Java and C++. The concept of a destructor does not exist in Java, but does in C++. It is very important in Neno to provide the programmer an explicit way of performing object destruction. Again, unlike Java, Neno is intended to be used on a persistent semantic network substrate. Thus, when a Fhat stops executing or an object is no longer accessible by a Fhat, that object should not be automatically removed. In short, Fhat does not provide automatic garbage collection (Louden, 2003). It is the role of the programmer to explicitly remove all unwanted objects from the RDF network.

In order to create a new object, the constructor of a class is called using the `new` operator. For example,

```
Human marko = new Human("Marko"^^xsd:string);
```

will generate a sub-network in the RDF network equivalent to Figure 4.
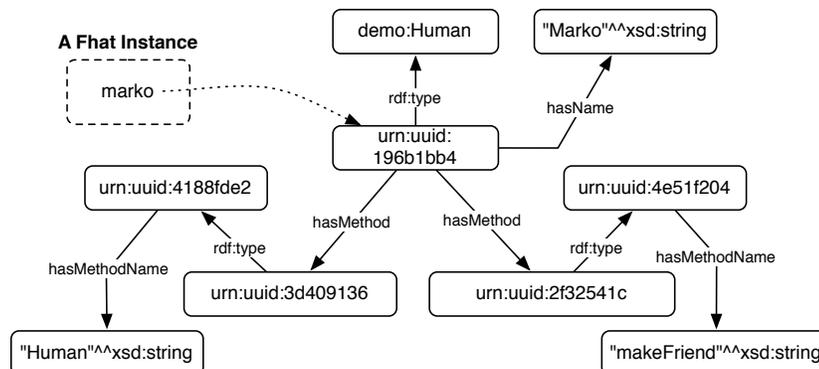
Figure 4: A Fhat instance maintains a variable reference to an object.

The algorithm by which Fhat creates the RDF sub-network will be discussed in the next section. For now, understand that in the variable environment of a Fhat instance there exists a variable named `marko` that points to the newly created `Human` instance (e.g. ⟨`marko`, `rdf:type`, `Human`⟩).

A destructor will instruct Fhat to destroy an object. A destructor is specified in the class declaration. For instance, suppose the following specification for `demo:Human`:

```
Thing Human {
  string hasName[1];
  Human hasFriend[0..*];

  !Human(string n) {
    this.hasName = n;
  }

  ~Human() {
    this.hasName =/
    this.hasFriend =/
    this..hasFriend =/
  }
}
```

In the above class declaration `!Human(string n)` is a constructor and `~Human()` is a destructor. A destructor is called using the `delete` operator. For instance,

```
delete marko;
```

calls `marko`'s `~Human()` destructor.

A class can only have at most one destructor and the destructor takes no arguments. The `~Human()` destructor removes the reference to the object's name, removes all the references to the object's friends, and removes all `hasFriend` references to that object. Thus, if the `Human` object has the URI `urn:uuid:55b2a3b0`, Fhat would execute the following commands on the triple-store:

```
DELETE { <urn:uuid:55b2a3b0> <demo:hasName> ?name .}
DELETE { <urn:uuid:55b2a3b0> <demo:hasFriend> ?human .}
DELETE { ?human <demo:hasFriend> <urn:uuid:55b2a3b0> .}
```

Behind the scenes, Fhat would also remove all the method references of `urn:uuid:55b2a3b0`, internal variable references to `urn:uuid:55b2a3b0`, and the `rdf:type` relationships that relate the object to the ontological-layer. When an object is properly destroyed, only its instance is removed from the RDF network. The object's class specification still exists in the ontological-layer.

3.2.14   General Query

In many instances, Fhat will not have a reference to a particular object. Again, the environment anticipated is one in which objects persist in the RDF network. Thus, when code is executed, it is necessary to locate the URI of a particular object for processing. In order to make this easy for the programmer, a query operator is defined called the "network query" operator and is denoted by the symbol `<?`. For example,

```
xsd:string x = "Marko Antonio Rodriguez"^^xsd:string;
xsd:string query =
   "SELECT ?x WHERE { ?x <demo:hasName> <" + x + "> }
     LIMIT 1"^^xsd:string;
Human h[0..1] <? query;
```

will query the RDF network for at most one `Human` named "Marko Antonio Rodriguez". Note that three statements above could have been written as one. However, to demonstrate string concatenation and variable use, three were used.

3.3 Starting a Program in Neno

In Neno, there are no static methods. Thus, there does not exist something like the `public static void main(String[] args)` method in Java. Instead, Fhat is provided a class URI and a method for that class that takes no arguments. The class is automatically instantiated by Fhat and the specified no-argument method is invoked. For example, if Fhat is pointed to the following `Test` class and `main` method, then the `main` method creates a `Human`, changes its name, then exits. When `main` exits, Fhat halts.

```
owl:Thing demo:Test {
  main() {
    demo:Human h = new Human("Marko Rodriguez");
    h.setName("Marko Antonio Rodriguez");
  }
}
```

3.4 Typical Use Case

This section describes how a developer would typically use the Neno/Fhat environment. The terminal commands below ensure that the NenoFhat compiler translates Neno source code to a Fhat OWL API, loads the Fhat OWL API into the triple-store, instantiates a Fhat RVM, and points the RVM to the `demo:Test` class with a `main` method. Note that the third command is broken into four lines for display purposes. Do not assume that there is a newline character at the end of the first three lines of the third statement.

```
> nenofhat Human.neno -o ntriple -t http://www.triplestore.net/sparql
> nenofhat Test.neno -o xml -t http://www.triplestore.net/sparql
> fhat -vmc http://neno.lanl.gov/neno#Fhat
        -c http://neno.lanl.gov/neno/demo#Test
        -cm main
        -t http://www.triplestore.net/sparql
```

The first terminal command compiles the `Human.neno` source code into a Fhat OWL API represented in N-TRIPLE format and then inserts the `Human.ntriple` triples into the triple-store pointed to by the "-t" URL. The second terminal command compiles the `Test.neno` source code and generates a Fhat OWL API in RDF/XML called `Test.xml`. That RDF/XML file is then loaded into the triple-store. The `nenofhat` compiler can produce any of the popular RDF syntaxes. While in most cases, one or another is chosen, two different syntaxes are shown to demonstrate what is possible with the compiler. Finally, a `Fhat` processor is initiated. The virtual machine process (`fhat`) is called with a pointer to an ontological model of the desired machine architecture. The machine architecture is instantiated. The instantiated `Fhat` then instantiates a `Test` object and calls its `main` method. The instantiated `Test main` method is executable RDF triple-code.

In some instances, a Fhat RVM state may already exist in the triple-store. In such cases, the following command can be invoked to point the Fhat RVM process to the stored RVM state. In the example below, assume that `urn:uuid:60ab17c2` is of `rdf:type Fhat`.

```
> fhat -vmi urn:uuid:60ab17c2 -t http://www.triplestore.net/sparql
```

When the `Fhat` RVM state is located, `fhat` processes the current instruction pointed to by its `programLocation`.

The following list outlines the flags for the `nenofhat` compiler,

- -o : output type (ntriple | n3 | xml)
- -t : triple-store interface

and the `fhat` RVM process,

- -vmi : virtual machine instance URI
- -vmc : virtual machine class URI
- -c : start class URI
- -cm : start class no-argument method
- -t : triple-store interface.

4   The Fhat Virtual Machine Architecture

Fhat is an RVM that was specifically designed for RDF-based semantic network languages. Fhat is a semi-hard implementation of a computing machine. Table 5 presents an explanation of the various levels of virtual machine implementations. The concept of soft, semi-hard, and hard implementations are developed here and thus, are not part of the common lexicon. In the JVM, all of the "hardware" components are represented in software and the state of the machine is not saved outside the current run-time environment. For VHSIC Hardware Description Language (VHDL) machines, the hardware components are modeled at the level of logic gates (AND, OR, XOR, NOT, etc.) (Coelho & Stanculescu, 1988). In Fhat, the hardware components are modeled in RDF (the state), but component execution is modeled in software (the process).

implementation type	requirements	example
soft	hardware methods	Java Virtual Machine, r-Fhat
semi-hard	high-level components	Fhat
hard	low-level components	VHDL designs

Table 5: Different VM implementation types, their requirements, and an example.

There are many reasons why a semi-hard implementation was desired for Fhat and these reasons will be articulated in the sections discussing the various components of the Fhat architecture. However, while this section presents the semi-hard implementation, a soft implementation of Fhat called reduced Fhat (r-Fhat) will be briefly discussed. In short, r-Fhat is faster than the Fhat virtual machine, but does not support run-time machine portability and machine-level reflection. In other words, r-Fhat does not support those functions that require an RDF representation of the machine state.

Any high-level language can be written to take advantage of the Fhat architecture. While Neno and Fhat were developed in concert and thus, are strongly connected in their requirements of one another, any language that compiles to Fhat RDF triple-code can use a Fhat RVM. This section will discuss the Fhat RVM before discussing the Fhat instruction set. Figure 5 presents the Fhat machine architecture. This machine architecture is represented in OWL and is co-located with other resources in the ontology layer of the RDF network.
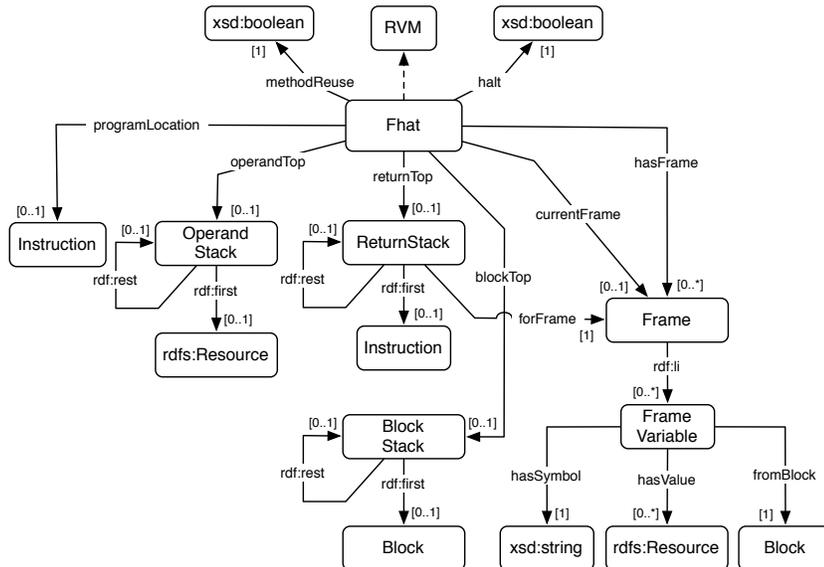

Figure 5: The ontological model of the Fhat virtual machine.

There are 8 primary components to the Fhat RVM. These are enumerated below for ease of reference. Each component will be discussed in more detail in the following subsections.

1. `Fhat`: the CPU that interprets instructions and uses its various components for processing those instructions.

2. `halt`: suspends Fhat processing when false, and permits processing when true.

3. `methodReuse`: determines whether or not method triple-code is reused amongst object instances.

4. `programLocation`: a pointer to the current instruction being executed (i.e. a PC).

5. `BlockStack`: an `rdf:List` that can be pushed and popped for entering and exiting blocks.

6. `OperandStack`: an `rdf:List` that can be pushed and popped for arithmetic computations.

7. `Frame`: a `Method` unique environment for storing local variables.

8. `ReturnStack`: an `rdf:List` that provides a reference to the instruction that called a method and the frame of that method.

While all of these components are represented in RDF, only `Fhat` has an external software component. The software implementation of `Fhat` is called the "virtual machine process" in Figure 3.[12]

4.0.1 **Fhat**

`Fhat` is the primary component of the Fhat RVM. `Fhat` is the most complicated component in the entire Fhat architecture. The high-level Neno pseudo-code for the `Fhat` component is

```
Thing Fhat {
  execute() {
    while(!this.halt && this.programLocation != null) {
      Instruction i = this.programLocation
      if(i typeof Block) { ... }
      else if(i typeof If) { ... }
      else if(i typeof Expression) { ... }
      else if(i typeof Set) { ... }
      ...
      /* update programLocation */
    }
  }
}
```

The above pseudo-code should be implemented in the language of the virtual machine process and thus, for the executing hardware CPU.

It is worth noting that a `Fhat` virtual machine process can be written in Neno as demonstrated in the Neno code above. For example, assume a Neno implemented Fhat instance called Fhat1. In such cases, another Fhat, called Fhat2, is processing Fhat1. Fhat2 can be run on yet another Fhat, called Fhat3, or grounded into some other language that is translating code to the native machine language. This is possible because Neno/Fhat is Turing complete and thus, can run a simulation of itself. When a simulation of itself is run, a complete RDF virtual machine is created. In this simulation environment, both the state and process of the Fhat RVM are represented in RDF.

The current version of Fhat supports most common uses of the `xsd:anySimpleType` and a few of these uses are summarized below:

- `xsd:boolean`: Not, Equals
- `xsd:integer, xsd:float, xsd:double`: Arithmetic, Compare
- `xsd:string`: Add, Compare
- `xsd:date, xsd:dateTime`: Add, Subtract, Compare
- `xsd:anyURI`: Compare.

---

[12]When the term Fhat is used, it is referring to the entire virtual machine, when the teletyped term `Fhat` is used, it is referring to the virtual machine process identified by the URI `Fhat`.

4.0.2 `halt`

At any time, Fhat can be forced to halt by setting the `halt` property of Fhat to true$^{\wedge\wedge}$xsd:boolean. Multi-threading can be simulated in this way. A Neno program can be engineered to run a master Fhat that has a reference to the `halt` property of all its slave Fhats. By setting the `halt` property, the Fhat master can control which Fhat slaves are able to process at any one time. In essence, the master Fhat serves as an operating system.

4.0.3 `methodReuse`

When `methodReuse` is set to true$^{\wedge\wedge}$xsd:boolean, Fhat will instantiate new objects with unique instructions for each method. When `methodReuse` is set to false$^{\wedge\wedge}$xsd:boolean, Fhat will reuse method triple-code amongst the same methods for the different objects. This will be discussed in more detail in §5.2.

4.0.4 `programLocation`

The `programLocation` is a pointer to the current instruction being executed by Fhat. Fhat executes one instruction at a time and thus, the `programLocation` must always point to a single instruction. The "while" loop of Fhat simply moves the `programLocation` from one instruction to the next. At each instruction, Fhat interprets what the instruction is (by its rdf:type "opcode") and uses its various components appropriately. When there are no more instructions (i.e. when there no longer exists a `programLocation` property), Fhat halts.

4.0.5 `BlockStack`

The `BlockStack` is important for variable setting. When a new variable is created in a block of code, it is necessary to associate that variable with that block. When the thread of execution exits the block, all variables created in that block are dereferenced (i.e. deallocated).

4.0.6 `OperandStack`

The `OperandStack` is a LIFO (i.e. "last in, first out") stack that supports any rdfs:Resource. The `OperandStack` is used for local computations such as x = 1 + (2 * 3). For example, when x = 1 + (2 * 3) is executed by Fhat, Fhat will

1. push the value 1 on the OperandStack

2. push the value 2 on the OperandStack

3. push the value 3 on the OperandStack

4. pop both 2 and 3 off the OperandStack, multiply the two operands, and push the value 6 on the OperandStack

5. pop both 1 and 6 the OperandStack, add the two operands, and push the value 7 on the OperandStack

6. set the current Frame FrameVariable x to the value 7 popped off the OperandStack.

The Neno statement x = 1 + (2 * 3) is actually multiple instructions when compiled to Fhat triple-code. The NenoFhat compiler would translate the statement to the triple-code represented in Figure 6.

It is very important to represent such components as the OperandStack component in RDF and not simply in the memory of the host CPU. Suppose that a Fhat instance is to move to another physical machine or, by chance, lose its process "back-end". If any of these two scenarios were the case, the state of the machine is always saved in RDF and thus, would simply "freeze" to await another virtual machine process to continue its execution. If the OperandStack was represented in software and thus, in RAM, then when the software halted, the OperandStack would be lost and the state of the machine would be inconsistent with its programLocation. With an RDF state, the RAM representation of the virtual machine process has a negligible effect on the consistency of the machine.
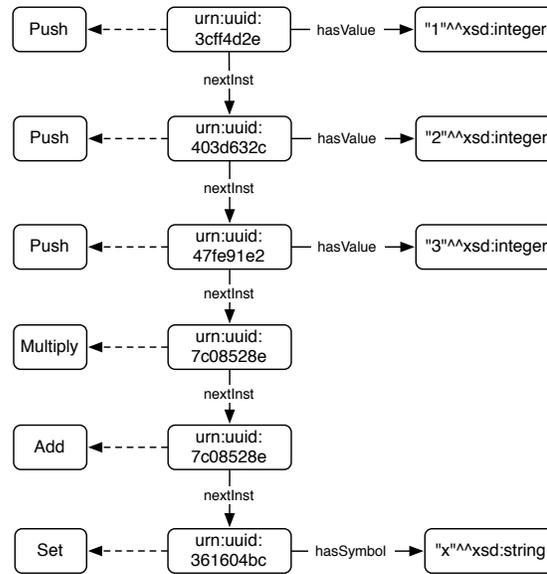
Figure 6: The triple-code representation of the statement x = 1 + (2 * 3).

4.0.7 `Frame`

Fhat is a frame-based processor. This means that each invoked method is provided a `Frame`, or local environment, for its variables (i.e. `FrameVariables`). Due to how variables are scoped in object-oriented languages and because Neno does not support global variables, each method can only communicate with one another through parameter (i.e. method arguments) passing, return value passing, or object manipulations. When method A calls method B, the parameters passed by method A are stored in method B's `Frame` according to the variable names in the method description. For example, assume the following method,

```
xsd:integer methodB(xsd:integer a) {
  return a + "1"^^xsd:integer;
}
```

If method A calls method B, with the statement,

```
xsd:integer x = marko.methodB("2"^^xsd:integer);
```

the value 2 is placed into the `Frame` of method B with the associated variable a. Method B adds 1 to the value and pushes the value 3 on the `OperandStack`. Method A pops one value off the `OperandStack` and sets the local variable x to the value 3. The `OperandStack` is used for the placement of method return values.

4.0.8 `ReturnStack`

The `ReturnStack` is a LIFO stack that maintains pointers to the return location of a method and the method `Frame`. To support recursion, the `ReturnStack` maintains a pointer to the specific `Frame` that is being returned to.

In order to explain how the `ReturnStack` is used, an example is provided. When method A calls method B, the next instruction of method A following the method invocation instruction is pushed onto the `ReturnStack`. When method B has completed its execution (e.g. a `return` is called), Fhat pops the instruction off the `ReturnStack` and sets its `programLocation` to that instruction. In this way, control is returned to method A to complete its execution. When `return` is called in method B, Fhat will delete (i.e. deallocate) all triples associated with the method B `Frame`. If `return` has a value (e.g. `return` 2), that value is pushed onto the `OperandStack` for method A to use in its computation.

4.1 Migrating Fhat Across Different Host CPUs

An interesting aspect of Fhat is the ability to migrate a Fhat process across various host CPUs. A Fhat implementation has two primary components: an RDF state representation and a software process. Because both the RDF triple-code and the complete state of a Fhat instance is represented in the RDF network, it does not matter which Fhat process is executing a particular Fhat state. The Fhat RDF state representation ensures that there are no global variables in the software process. The only variables created in the software process are local to the instruction being executed. Because there are no global variables in the software process, any software process can execute the Fhat RDF state without requiring inter-software process communication. For example, one host CPU can be running the Fhat software process and halt. Another CPU can then start another Fhat software process that points to the URI of the originally halted Fhat RDF state and continue its execution. This concept is diagrammed in Figure 7 where $n = 1$ refers to instruction 1.
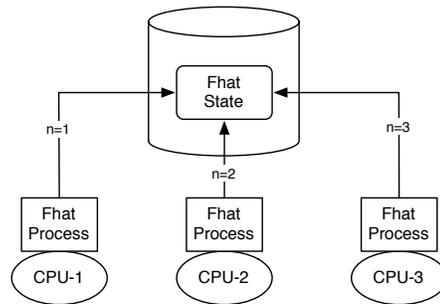
Figure 7: Migrating a Fhat execution across multiple host CPUs.

In principle, each CPU can execute one instruction and then halt. In this way, it is possible to migrate the Fhat RVM across different host CPU's. Thus, if a portion of the Semantic Web is needed for a particular computation, it may be best to have the physical computer supporting that RDF sub-network host the Fhat RVM. Once the Fhat RVM has completed computing that particular RDF sub-network, it can halt and another CPU can pick up the process on a yet another area of the Semantic Web that needs computing by the Fhat RVM. In this model of computing, data doesn't move to the process, the process moves to the data. This idea is diagrammed in Figure 8, where both triple-store servers have Fhat process implementations.
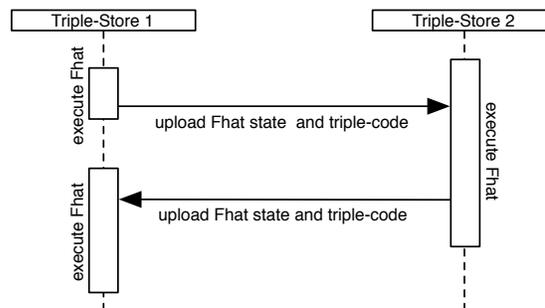

Figure 8: Migrating a Fhat state across different triple-stores.

4.2 Fhat Reflection

A Fhat RVM and the triple-code that it is executing are in the same address space and thus, can reference one another. It is the UUID address space of Neno/Fhat that makes it a unique programming environment in that Neno is not only a completely reflective language, but also that it removes the representational stack found in most other programming environments.

Language reflection means that the program can modify itself during its execution. Many scripting languages and even Java (through the `java.lang.reflect` package) support language reflection. However, not only does Neno/Fhat support language reflection, it also supports machine reflection. A Fhat can modify itself during its execution. There are no true boundaries between the various components of the computation. This idea is represented in Figure 9, where a Fhat RVM has its program counter (`programLocation`) pointing to a `Push` instruction. The `Push` instruction is instructing Fhat to push a reference to itself on its operand stack. With a reference to the Fhat instance in the Fhat operand stack, Fhat can manipulate its own components. Thus, the Fhat RVM is executing triple-code that is manipulating itself.
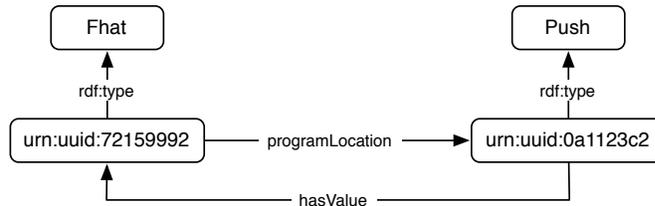
Figure 9: A Fhat processor can process itself.

4.3 r-Fhat

What has been presented thus far is a semi-hard implementation of Fhat. The semi-hard implementation explicitly encodes the state of a Fhat instance in RDF. While this has benefits such as fault tolerance due to virtual machine process failures, support for distributed computing in the form of processor migration, and support for machine-based evolutionary algorithms, it requires a large read/write overhead. Each instruction requires the virtual machine process to explicitly update the virtual machine state. A faster Fhat virtual machine can be engineered that does not explicitly encode the state of the machine in the RDF network. In such cases, the only read/write operations that occur are when an object is instantiated, destroyed, or a property manipulated. This faster Fhat is called reduced Fhat (r-Fhat). In r-Fhat, the operand stack, return stack, etc. are data structures in the implementing language. r-Fhat does not have an OWL machine architecture nor an RDF state.

5 The Fhat Instruction Set

In order for Neno software to run on a Fhat machine instance, it must be compiled to a Fhat OWL API that is compliant with the Fhat instruction set (the Fhat OWL API `owl:imports` the Fhat instruction set ontology). A Fhat RVM uses the Fhat OWL API as a "blueprint" for constructing the instance-level representation of the RDF triple-code. It is the instance-level triple-code that the Fhat RVM "walks" when a program is executing.

5.1 The `Method`

In Neno, the only process code that exists is that which is in a `Method` `Block`. Figure 10 defines the OWL ontology of a `Method`.

A `Method` has an `ArgumentDescriptor` that is of `rdfs:subClassOf` `rdf:Seq` and a return descriptor that is of type `rdfs:Resource`. The sequence of the `ArgumentDescriptor` `Argument` denotes the placement of the `Method` parameter in the method declaration. For instance,

```
xsd:integer exampleMethod(xsd:string n, Human h) { ... }
```

would set the object of the `hasReturnDescriptor` property to the URI `xsd:integer` and the `ArgumentDescriptor` to the `Arguments` n (`rdf:_1`) and h (`rdf:_2`).
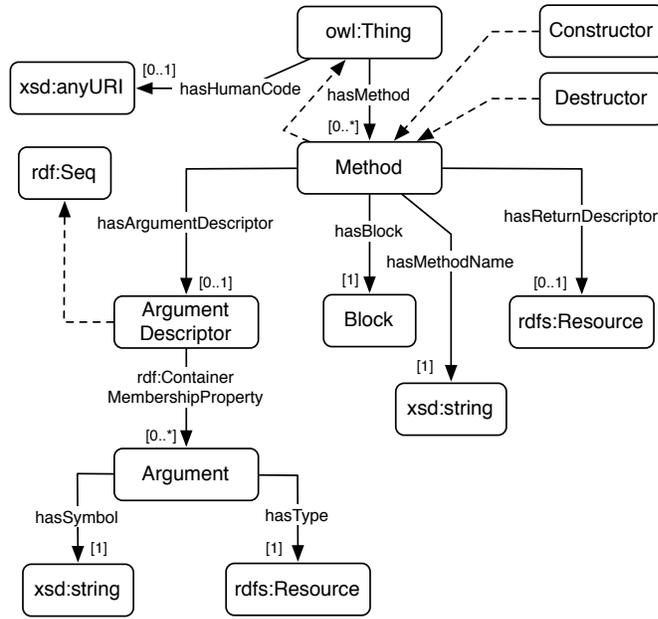
Figure 10: The OWL `Method` ontology.

The `hasHumanCode` property can be used, if desired, to point to the original human readable/writeable source code that describes that class and its methods. By using the `hasHumanCode` property, it is possible for "in-network" or run-time compiling of source code. In principle, a Neno compiler can be written in Neno and be executed by a Fhat RVM. The Neno compiler can compile the representation that results from resolving the URI that is the value of the `xsd:anyURI`.

5.1.1 A `Block` of Fhat Triple-Code

A `Method` has a single `Block`. A `Block` is an `rdfs:subClassOf Instruction` and is composed of a sequence of `Instructions`. The `Instruction` sequence is denoted by the `nextInst` property. The `Instruction` `rdf:type` is the "opcode" of the `Instruction`. The set of all `Instructions` is the instruction set of the Fhat architecture. Figure 11 provides a collection of the super class `Instructions` that can exist in a `Block` of code and their relationship to one another.

Examples of these super classes are itemized below.[13]

- `Arithmetic`: `Add`, `Divide`, `Multiply`, `Not`, `Subtract`.
- `Condition`: `Equals`, `GreaterThan`, `GreaterThanEqual`, `LessThan`, `LessThanEqual`.
- `Setter`: `NetQuery`, `Set`, `SetClear`, `SetMinus`, `SetPlus`, `SetQuery`.
- `Invoke`: `Construct`, `Destruct`.

The `Value` class has a set of subclasses. These subclasses are itemized below.

- `Direct`: `LocalDirect`, `PopDirect`.
- `Variable`: `LocalVariable`, `FieldVariable`, `ObjectVariable`.

When a Fhat instance enters a `Method` it creates a new `Frame`. When a `Variable` is declared, that `Variable` is specified in the `Frame` and according to the current `Block` of the `Fhat` instance as denoted by `Fhat`'s `blockTop` property. A `Block` is used for variable scoping. When Fhat leaves a `Block`, it destroys all the `FrameVariables` in the current `Frame` that have that `Block` as their `fromBlock` property (refer to Figure 5). However, entering a new `Block` is not exiting the old `Block`. Parent `Block` `FrameVariables` can be accessed by child `Blocks`. For instance, in the following Neno code fragment,

---

[13]`Conditions` are unique in that they have a `trueInst` and a `falseInst` property. If the `Condition` is true, the next `Instruction` is the one pointed to by the `trueInst` property, else the next `Instruction` is one pointed to by the `falseInst` property.
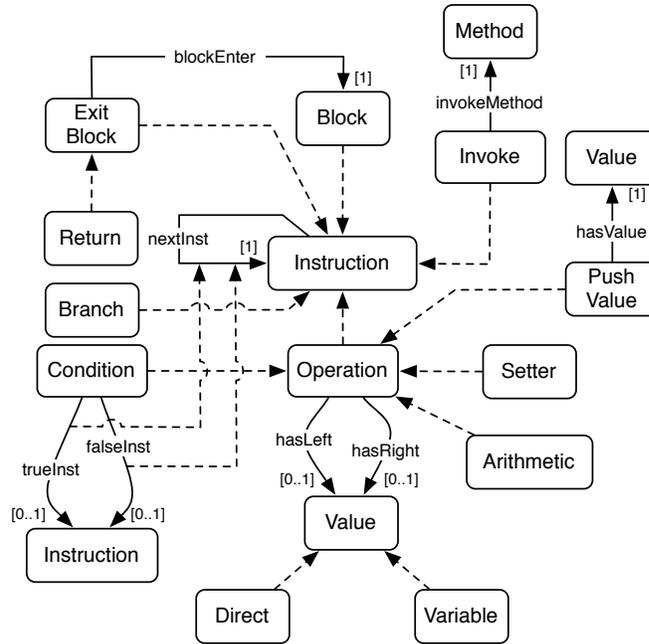
Figure 11: The OWL ontology for a `Block` of `Instructions`.

```
xsd:integer x = "1"^^xsd:integer;
if(x > 2) {
   xsd:integer y = x;
}
else{
   xsd:integer y = x;
}
```

the two `y` `Variables` in the if and else `Blocks` are two different `FrameVariables` since they are from different `Blocks`. Furthermore, note that both the if and else `Blocks` can access the value of `x` since they are in the child `Block` of the `Block` declaring the variable `x`. When Fhat leaves a `Method` (i.e. returns), its `Frame` and its `FrameVariables` are destroyed through dereferencing.

5.2 A `Method` Instance

There are two ways in which a `Method` instance is handled by Fhat: global and local instance models. In the global instance model, when a new object is instantiated, its methods are also instantiated. However, if the instantiated `Method` already exists in the RDF network, the newly created object points its `hasMethod` property to a previously created `Method` of the same `hasMethodName` and UUID. Thus, only one instance of a `Method` exists for all the objects of the same class type. While it is possible to have a unique `Method` instance for each object, by supporting method reuse amongst objects, Fhat limits the growth (in terms of the number of triples) in the RDF network. Furthermore, this increases the speed of the Fhat RVM since it does not need to create a new `Method` from the Fhat OWL API of that `Method`. The global instance model is diagrammed in Figure 12. To ensure global instances, the `methodReuse` property of the `Fhat` instance is set to `"true"`$^{\wedge\wedge}$`xsd:boolean` (refer to Figure 5).

In the local instance model, the `methodReuse` property of a `Fhat` instance is set to `"false"`$^{\wedge\wedge}$`xsd:boolean`. In such cases, a new `Method` instance is created with each new instance of an `owl:Thing`. The benefit of this model is that method reflection can occur on a per-object basis. If an object is to manipulate its `Method` triple-code at run-time, it can do so without destroying the operation of its fellow `owl:Things`. The drawback of the local
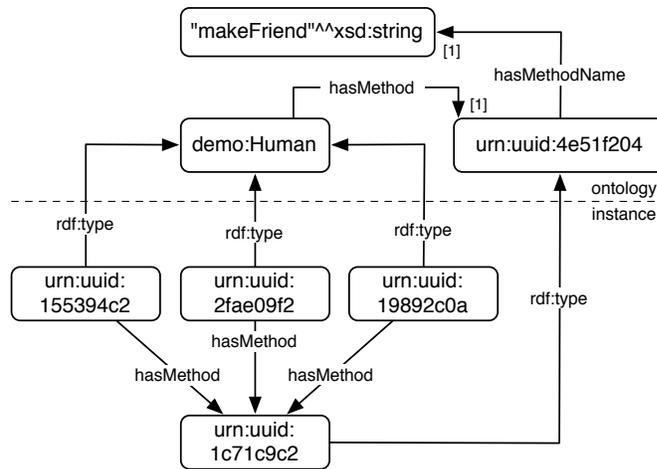
Figure 12: Multiple object's of the same type will share the same `Method` instance.

instance model is triple-store "bloat" and an increase in the time required to instantiate an object relative to the global instance model. The local instance model is diagrammed in Figure 13.
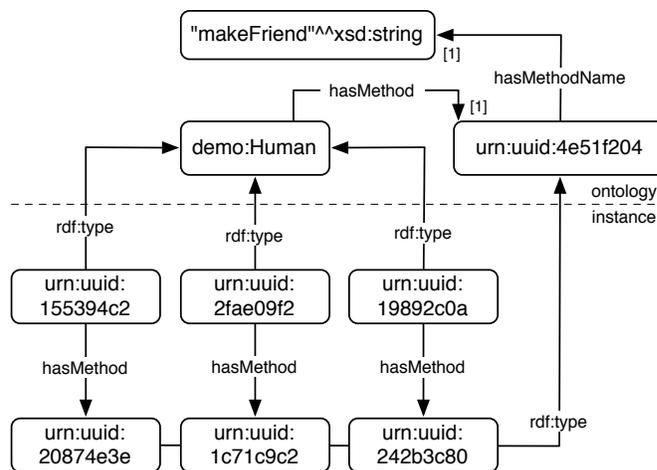

Figure 13: Multiple object's of the same type each have a unique `Method` instance.

5.2.1 An Example **Method** Instance

Suppose the following code,

```
owl:Thing demo:Human
{
  xsd:int example(xsd:string a) {
    if(a == "marko"^^xsd:string) {
      return "1"^^xsd:int;
    }
    else {
      return "2"^^xsd:int;
    }
  }
}
```

When this code is compiled, it compiles to a Fhat OWL API. When an instance of `demo:Human` is created, the Fhat RVM will start its journey at the URI `demo:Human` and move through the ontology creating instance UUID URIs for all the components of the `demo:Human` class. This includes, amongst its hard-coded properties, its `Methods`, their `Blocks`, and their `Instructions`. When the `demo:Human` class is instantiated, an instance will appear in the RDF network as diagrammed in Figure 14.
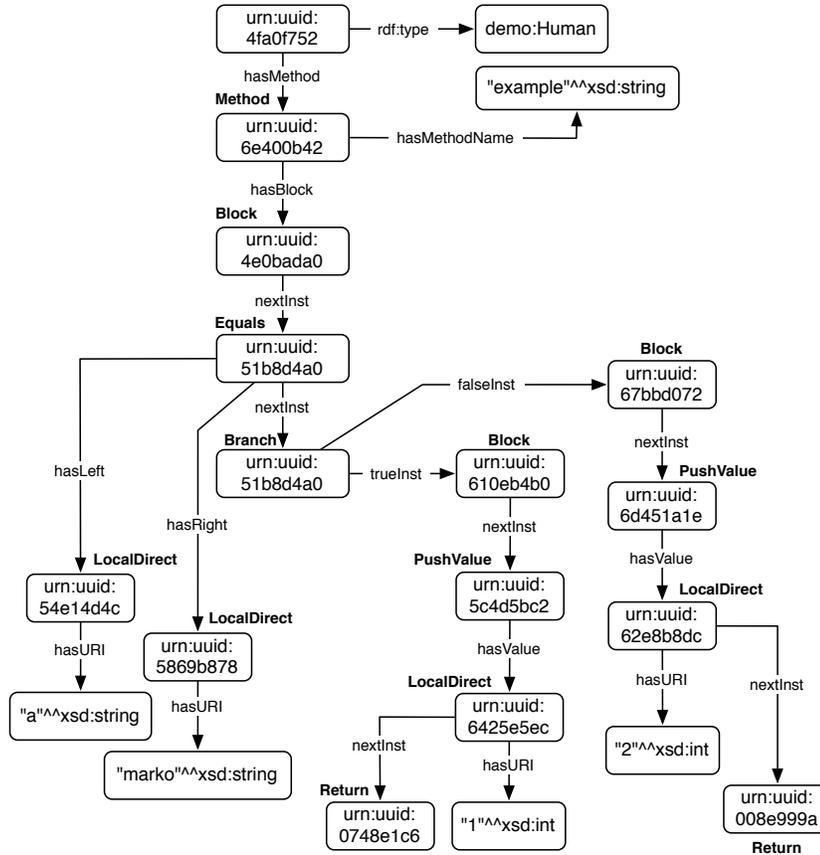

Figure 14: The RDF triple-code for the `example(xsd:string)` method.

6   Conclusion

The primary drawback of the RVM is computing time because a virtual machine must not only read the RDF network to interpret the program instructions, it must also read/write to the RDF network to manipulate data (i.e. instance objects) in the RDF network. Moreover, the virtual machine must read and write to that sub-network of the RDF network that represents the virtual machine's state (e.g. program counter, operand stack, etc.). In doing so, many read/write operations occur in order for the virtual machine to compute. However, much of this issue can be resolved through the use of r-Fhat.

   Imagine a world where virtual machines are as easy to distribute as an HTML document (e.g. an RDF/XML encoding of the virtual machine sub-network). Given that a virtualized machine encodes its state in the RDF network, think about how RVMs can "move" between physical machines in mid-execution. There is complete hardware independence as no physical machine maintains a state representation. Physical machines compute the RVM by reading its program location, its operand stack, its heap, etc. and update those data structures. In such situations, a personal computer can be encoded in the RDF network and be accessed anywhere. Thus, the underlying physical machine is only a hardware shell for the more "personal" machine encoded in the RDF network. These ideas are analogous

to those presented in (Satyanarayanan et al., 2007).

In the RDF network, RVMs, APIs, and triple-code are "first-class" web entities. What happens when archiving services such as the Internet Archive, search engine caches, and digital libraries archive such RDF programs and "snap-shot" states of the executing RVMs (Lorie, 2001; Nelson, McCown, Smith, & Klein, 2007)? In theory, the state of computing world-wide, can be saved/archived and later retrieved to resume execution. The issues and novelties that archiving computations presents are many and are left to future work in this area.

Much of the Semantic Web effort is involved in the distribution of knowledge between organizational boundaries (Davies, Fensel, & Harmelen, 2003). This is perhaps the primary purpose of the ontology. In this respect, organizations of a similar domain should utilize shared ontologies in order to make their information useable between their respective organizations. Procedural encodings support the distribution of not only the knowledge models, but also the algorithms that can be applied to compute on those models. In a non-disjoint manner, data and code are easily exchanged between organizational boundaries (Rodriguez, Watkins, Bollen, & Gershenson, 2007).

Given that the RDF network is composed of triples and triples are composed of URIs and literals, the address space of any virtual machine in the RDF network is the set of all URIs and literals. Given that there are no bounds to the size of these resources, there are no realistic space limitations on the RVM. In other words, the amount of disk-space provided world-wide to support the Semantic Web is the actual memory constraints of this model. However, the success of this distributed computing paradigm relies on the consistent use of such standards as the Link Data specification (Berners-Lee, 2006). With further developments in Linked Data models and the RVM model of computing, the Semantic Web can be made to behave like a general-purpose computer.

Acknowledgments

References

Aasman, J. (2006). *Allegro graph* (Tech. Rep. No. 1). Franz Incorporated.

Aho, A. W., Sethi, R., & Ullman, J. D. (1986). *Compilers: Principles, techniques, and tools.* Addison-Wesley.

Alesso, H. P., & Smith, C. F. (2005). *Developing Semantic Web services.* Wellesey, MA: A.K. Peters LTD.

Beckett, D. (2001). *N-Triples* (Tech. Rep.). University of Bristol.

Berners-Lee, T. (1998). *Notation 3* (Tech. Rep.). World Wide Web Consortium.

Berners-Lee, T. (2006). *Linked data* (Tech. Rep.). World Wide Web Consortium.

Berners-Lee, T., Fielding, R. T., Software, D., Masinter, L., & Systems, A. (2005, January). *Uniform Resource Identifier (URI): Generic Syntax.*

Biron, P. V., & Malhotra, A. (2004). *XML schema part 2: Datatypes second edition* (Tech. Rep.). World Wide Web Consortium.

Bray, T., Hollander, D., Layman, A., & Tobin, R. (2006). *Namespaces in XML 1.0* (Tech. Rep.). World Wide Web Consortium.

Britton, R. (2003). *MIPS assembly language programming.* Prentice Hall.

Carroll, J. J., & Stickler, P. (2004). RDF triples in XML. In *Extreme markup languages.* Montréal, Québec: IDEAlliance.

Coelho, D., & Stanculescu, A. (1988). A state-of-the-art VHDL simulator. In *Thirty-third IEEE computer society international conference* (p. 320-323). San Francisco, CA.

Craig, I. D. (2005). *Virtual machines.* Springer.

Davies, J., Fensel, D., & Harmelen, F. van. (2003). *Towards the semantic web: Ontology-driven knowledge management.* Wiley.

Eckel, B. (2002). *Thinking in Java.* Prentice Hall.

Fensel, D. (2004). Triple-space computing: Semantic web services based on pesistent publication of information. In *Proceedings of the International Conference on Intelligence in Communication Systems* (pp. 43–53). Bangkok, Thailand.

Fensel, D., Hendler, J. A., Lieberman, H., & Wahlster, W. (Eds.). (2003). *Spinning the Semantic Web: Bringing the World Wide Web to its full potential.* MIT Press.

Hennessy, J. L., & Patterson, D. A. (2002). *Computer architecture: A quantitative approach.* Morgan Kaufmann.

Kifer, M., Lausen, G., & Wu., J. (1995). Logical foundations of object-oriented and logical foundations of object-oriented and. *Journal of the ACM*, *42*(4), 741-843.

Koide, S., & Kawamura, M. (2004). SWCLOS: A Semantic Web processor on Common Lisp object system. In *Proceedings of the International Semantic Web Conference.* Hiroshima, Japan.

Leach, P. J. (2005). *A Universally Unique IDentifier (UUID) URN Namespace* (Tech. Rep.). Network Working Group.

Lee, R. (2004). *Scalability report on triple store applications* (Tech. Rep.). Massachusetts Institute of Technology.

Lindholm, T., & Yellin, F. (1999). *The Java virtual machine specification.* Addison-Wesley.

Lorie, R. A. (2001). Long term preservation of digital information. In *Proceedings of the 1st ACM/IEEE-CS joint conference on digital libraries* (pp. 346–352). New York, NY, USA: ACM Press.

Louden, K. C. (2003). *Progamming languages: Principles and practice.* Brooks/Cole - Thomson Learning.

Manola, F., & Miller, E. (2004, February). *RDF primer: W3C recommendation.*

Martin, D., Burstein, M., Hobbs, J., Lassila, O., McDermott, D., McIlraith, S., Narayanan, S., Paolucci, M., Parsia, B., Payne, T., Sirin, E., Srinivasan, N., & Sycara, K. (2004). *OWL-S: Semantic markup for web services* (Tech. Rep.). World Wide Web Consortium.

McGuinness, D. L., & Harmelen, F. van. (2004, February). *OWL web ontology language overview.*

Nelson, M. L., McCown, F., Smith, J., & Klein, M. (2007). Using the web infrastructure to preserve web pages. *International Journal on Digital Libraries.*

Oren, E., Delbru, R., Gerke, S., Haller, A., & Decker, S. (2007). ActiveRDF: Object-oriented semantic web programming. In *Proceedings of the International World Wide Web Conference WWW07.* Banff, Canada.

Prud'hommeaux, E., & Seaborne, A. (2004). *SPARQL query language for RDF* (Tech. Rep.). World Wide Web Consortium.

Rodriguez, M. A., Watkins, J. H., Bollen, J., & Gershenson, C. (2007). Using RDF to model the structure and process of systems. *InterJournal of Complex Systems*, *2131.*

Satyanarayanan, M., Gilbert, B., Toups, M., Tolia, N., Surie, A., O'Hallaron, D. R., Wolbach, A., Harkes, J., Perrig, A., Farber, D. J., Kozuch, M. A., Helfrich, C. J., Nath, P., & Lagar-Cavilla, H. A. (2007). Pervasive personal computing in an internet suspend/resume system. *IEEE Internet Computing*, *11*(2), 16-25.

Seaborne, A., & Manjunath, G. (2007). *SPARQL/Update: A language for updating RDF graphs* (Tech. Rep.). Hewlett-Packard.

Sebesta, R. W. (2005). *Concepts of programming languages.* Addison-Wesley.

Shinavier, J. (2007). Functional programs as Linked Data. In *3rd Workshop on Scripting for the Semantic Web.* Innsbruck, Austria.

Sowa, J. F. (1991). *Principles of semantic networks: Explorations in the representation of knowledge.* San Mateo, CA: Morgan Kaufmann.

Sowa, J. F. (1999). *Knowledge representation: Logical, philosophical, and computational foundations.* Course Technology.

Turing, A. M. (1937). On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society, 42*(2), 230–265.

Wang, H. H., Noy, N., Rector, A., Musen, M., Redmond, T., Rubin, D., Tu, S., Tudorache, T., Drummond, N., Horridge, M., & Sedenberg, J. (2007). Frames and OWL side by side. In *10th International Protégé Conference.* Budapest, Hungary.