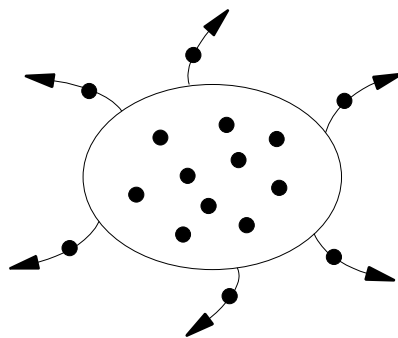


Introduction to Semantic Programming

Oli Sharpe, February 2008

Draft version 1.1



1 Introduction

This paper provides a high level overview of a proposed new paradigm for using computers, called 'Semantic Programming'. Before proceeding further, it is important to note some important practical issues that will not be covered in this paper.

We will not be looking at questions about how a new paradigm could be usefully adopted. Nor will we look in detail at how this paradigm can be usefully connected to programs within existing paradigms. Finally, we will not be looking at how the theoretical ideas presented here could be practically realised.

These are all fundamentally important issues without which a new programming paradigm has no useful purpose, and so I urge the reader to rest assured that much research has gone into exploring these issues and this further work will be reported on in later papers.

The purpose of this paper is to introduce the ideas behind Semantic Programming.

1.1 General Motivations

Over the last number of decades computers have become ever more usefully employed as a fundamental component of both our economic and intellectual endeavours. However, as computers have grown in speed, scale and connectivity our methods of programming computers haven't evolved nearly as fast.

In particular, there are contingent, practical choices that were made throughout this history (such as the hierarchical file system developed in the 1960s) that today are so embedded in our way of using computers that it is at times hard to remember that they are indeed a choice.

Having said that, it is worth stating clearly at this point that adherence to common standards, however contingent those standards may be, has been a fundamental enabler to allow us to build the complex layers of abstraction and cross system interaction that we enjoy today. Therefore any proposal to abandon or replace a given standard needs to have compelling advantages over the way things are done currently.

I believe that Semantic Programming offers such advantages as well as being able to live comfortably alongside existing systems.

2 Concerns with existing approaches

Please note that I do not claim to be the first to either notice or attempt to address the following concerns. This is simply a collection of some of the key motivations that led me to explore the ideas that became Semantic Programming.

2.1 Currently programs are thought of as text we could write with pen and paper

Since the development of compilers in the 1950s the programming of computers has been thought about in terms of writing human readable text. Indeed, the text of a program could be written entirely with pen and paper and only once complete transferred to a computer to be run. This approach to programming has many great features.

As well as the obvious benefit of making computer programs easier to read, it also allowed the processes of teaching and learning about programming to be separable from computers themselves. While computers were expensive, books about computers were cheap. Being able to fully express computer programs in an easy to read fashion within a book is a fantastically useful achievement.

However, as the complexity and size of programs has grown, I believe this approach is reaching the end of its useful life. Text is inherently linear, whereas large programs are more naturally thought of as complex interconnected graphs. As the size and complexity of the program graph has moved further away from the linear structure of a text document, so we have had to make ever more cumbersome efforts to express the full program in a collection of text files.

In many areas of enterprise programming we end up tying together bits of our program graph through hand typed connection tokens. If these tokens are wrong, the system fails to work correctly, so they are every bit as important to the program as the neatly crafted object oriented parts of the program. Often these tokens (or other equivalent) live in XML configuration files that live outside the main program text.

The standard way of thinking about these different groups of text files that make up the final program graph is to see the separation as being justified by the different concerns that they each relate to. For example, you might have Java code as 'core product', XML files as local system configuration details, another set of files to define local business logic and yet another set of files to manage the local look and feel.

However, this 'separation of concerns' explanation for the state of affairs often feels like a post-rationalisation of what is really a breakdown in the way that we currently program computers.

2.1.1 Programs are complex graphs we should manage with computers

We should stop thinking about programming as being equivalent to writing text with pen and paper. It is not. Programming is the creation and manipulation of complex interconnected graphs. We should be using computers to help visualise and manipulate these graphs in meaningful ways. The visualisation may at times include the use of text, but text displayed within a dynamic context that is still linked to the other parts of the graph that we are not currently looking at.

Computers are cheaper now and it is reasonable to suppose that we can always have a computer helping us manage this complex graph while we program and while we teach programming. Sure books with toy examples in text format will remain useful teaching tools, but we should always be aware of the limitations of text (and indeed static two dimensional diagrams) to be able to represent practically useful programs.

2.2 Currently programs manage data in the local memory of one computer

Programming languages since the 1960s until the present mostly concern themselves with the question of how to make programs that will manage data in the memory of one computer. There are two important consequences of this:

1. Long term persistence of data has to be explicitly managed
2. Connectivity with remote data has to be explicitly managed.

Many programs today (if not the majority) have to do both of these: persist data and connect to remote data¹. There have been many ways in which the required explicit management has been made easier within modern programming frameworks, however these approaches often feel like they are papering over a fundamental problem with the underlying programming paradigms.

As the programming languages natively treat all variable references as referring to local memory, the typical solution is to create a local memory proxy for any remote data. Indeed, this is the only practical solution and is not itself the problem. The problem is that most existing programming paradigms allow data in local memory to retain a greater status (and thereby superior behaviour) than remote data.

For example, remote data will always have the risk of latency delays or even a complete disconnection that results in failure of whatever action was being taken. Therefore handling of remote data always requires extra error handling code in case these unwanted events occur. This extra code is often ugly and complex when written in linear text form. Sometimes the latency concerns can be so great that remote data access is best managed asynchronously.

In contrast local data can be trusted to behave much better, and therefore code written exclusively for local data is simpler to write in text and therefore simpler to read, understand and debug. It is also true that when writing high performance programs explicit control over the local / remote distinction is essential in order to allow the programmer to optimise the behaviour of the program.

Indeed, the programming language C was written for developing system programs that need great performance and therefore it benefits from keeping the local / remote distinction. C++, Java, C# and many other contemporary programming languages have kept this influence from C.

However, for most general programming the distinction leaves the programmer in the perilous position of having to plan with foresight which data will be held locally and which remotely. Given the temptation of the easier to read code and greater performance for local data, most programmers naturally gravitate to handling most of their data as explicitly local data. Thus persistence and connectivity to other programs and other computers remains a peripheral concern, one that is often tacked on in later stages of the development project.

Indeed, it's often the need to link these peripheral concerns of persistence and connectivity to the core program that lead to the spaghetti mess of text files with linking tokens discussed in the previous section.

2.2.1 All data should be managed as if it's persistent, remote data

The reality of programming today is that most new programs written will not only need to persist their data but will also need to connect to other programs across the internet. Indeed, we often do not know up front which data will be local or which will be remote. It therefore feels like now is an appropriate time to suggest that our programming languages should treat all variable references as potentially being for remote data that resides somewhere else on the internet.

¹ Clearly the persistence of and access to data is invariably mediated by other programs on either the local computer or a remote computer so for 'remote data' you could instead read 'data managed by another program'.

Sure, under the hood our language implementations may take advantage of knowing that some specific data happens to be held locally at a given moment in time, but the programmer shouldn't have to care any more about this distinction.

Similarly we should switch around to the assumption that data should be persisted unless explicitly deleted. This, again, can leave the general programmer free to focus on the basic logic of what needs to happen rather than having to care about the underlying 'plumbing'.

Given that interactions with remote data have more chance of failing mid way through a procedure, it also, therefore, makes sense to suggest that we should always be programming with transaction management in place. This will ensure that the persisted data remains consistent whatever happens.

Today, programming in text files, the extra code for transaction management and exception management is so ugly and cumbersome that it is only regularly used for large scale enterprise software. However, the natural pressure is to be able to provide enterprise level robustness to all levels of computer user. To achieve this flexibly we need to change our approach to programming.

If, as suggested in 2.1.1 above, we now use computers to help us manipulate the complex interconnected graphs that define our programs, then we can manage more sensibly and aesthetically the extra error catching and transaction management code that is needed to safely handle all data as if it's remote.

2.3 Currently we're tempted to think about ever larger buckets of data

One damaging consequence of the two previous concerns is that we end up thinking about computer systems in terms of buckets that hold a specific set of data.

This arises because building connections between systems currently requires extra work that has to be repeated for each connection. Indeed this work is usually treated as additional work (both from a technical and budgetary perspective) rather than as being fundamental. Therefore it always looks simpler and more convenient to either put more data into an existing bucket or to buy a separate, disconnected bucket.

The invention of the relational data model (1969) leading to the SQL standard for modern databases was another important milestone towards the successes of modern computing. However, it has also become the technology of choice for building isolated buckets of data.

Indeed, the temptation is to see 'building one big bucket for everything' as a strategically sensible option for an organisation because on the surface this looks like an easy and powerful approach. However, this is an illusion because most modern organisations do not have a clearly defined boundary. Collaboration between organisations is an increasingly important part of what companies, public bodies and charities do. Indeed, the working and private lives of individuals are also becoming increasingly fluid as well. So, there is no longer a clear place where to draw the boundary of the big data bucket.

At some stage systems will have to connect beyond any line that is drawn.

2.3.1 We should think of data as a graph distributed across many systems

Rather than building ever bigger buckets we should be thinking about data differently. As with programs, our data is really a complex graph and typically we want that graph to connect across and be distributed over many systems. Indeed, ideally users of the data should be able to use the graph of data without having to care (much) about the systems over which it is distributed.

I left a 'much' in brackets in the previous sentence because I believe that users do want to keep some level of control over which systems can and can't hold their data. There are some organisations that

we trust and some that we don't. We should, therefore, be able to specify this kind of control over where our data lives.

However, apart from some such types of control, most users will not otherwise care which systems hold which parts of their data as long as it's being held securely and they can get access to it when they need it.

The service oriented architecture (SOA) approach to connecting existing buckets of data is an important step in the right direction to help make it easier for existing systems to talk with each other. However, in contemporary programming languages the SOA approach has to be added and managed explicitly by the programmer.

If we adopt programming paradigms that treat all data as being remote data on another system (as suggested in 2.2.1) then we start getting some way towards natively treating our data as a graph distributed over many systems. This would put SOA thinking at the heart of the programming paradigm rather than at the periphery.

Instead of buckets of data we'd be thinking about graphs, or networks, of data.

2.4 Currently we mostly think of programs and data separately

As programming languages developed so too did the norm of seeing programs and data as quite separate kinds of things. The rise of software development methodologies further separated the way that programs and data are managed, which also separated the programmers who create and manage programs, from the users who create and manage data. As with many of the issues looked at so far, this separation of concerns has been an important part of the success of computer software development to date.

In some circumstances a strict separation of concerns, with a tightly managed software development process will always be necessary to ensure that the software performs perfectly. However, in many business and personal uses of computing the separation between the user and the programmer can result in costly software that doesn't do quite what anybody wanted it to do.

This separation also means that while our programs may use nicely encapsulated objects, our stored data is often 'naked' data with little context and no security. Conversely, while most organisations (just about) ensure that their data is well backed up, often the local configuration details of the programs are not backed up because they live in a no man's land between the programmer's responsibility for the core program and user's responsibility for the data.

Indeed with many modern pieces of software there is so much configuration that can be achieved by the user that it's worth questioning whether there is a clear boundary where the programming ends and pure 'data management' begins. Users want ever greater control over how their computers manage their data. If everyone is going to be instructing their computers what to do, doesn't that mean that everyone is going to be programming?

It's time to start seeing programs as just another form of user created data and all users as potential programmers.

2.4.1 Programs and data are just different parts of the same distributed graph

As explored earlier, programs and data are both interconnected graphs of information. If programs are just another form of user created data, then there is really just one big interconnected graph of information that we want the computers to help us manage. Some part of the graph we call 'programs' because those are the bits that instruct the computers what to do and when to do it.

Clearly we'll want to use access controls to ensure that only the appropriate people can change the

programs, as indeed only the appropriate people should be able to change any data. Similarly, by keeping the data and programs together this should help build systems that keep sensitive data protected by always keeping it within an active security context. We should try to avoid storing or exchanging 'naked' data.

By seeing programs and data as part of one information graph we can then also ensure that this one graph is stored in a way that is robust to failure, rather than having to think about and plan the backup and recovery plans for each program, each set of configuration details and each bucket of data.

2.5 Currently we're tempted to build one ontology for the world

The last concern that I'll mention here is about a false temptation when trying to connect multiple systems. We've already looked at how connecting systems is seen as hard work within current programming paradigms. The concern I wish to mention here is that even when people do go to the effort of connecting systems they often try to do so by attempting to define a definitive ontology for the relevant domain.

Clearly for two systems to talk with each other meaningfully they need to talk a standard language that they can both understand. They need to both understand a standard ontology. So far, so good.

The problem is how to scale this to multiple connections between multiple systems. All too often it appears that (as with buckets of data) it's just too tempting to think: "if we could just get everyone to agree one standard ontology then this would be easy".

In reality this "one ontology" dream is just as much an illusion as the one big bucket dream. There will always be alternative views on how best to organise an ontology of any given domain as expressed within one natural language, let alone between all world languages.

2.5.1 We should always expect to have to handle multiple ontologies

Rather than dreaming of agreeing the one correct ontology, instead we should always work on the assumption that our systems will have to be able to handle interactions with other systems using multiple different ontologies even within a single domain.

Indeed, these ontologies are not just going to be different, sometimes they will directly contradict each other and we simply must build systems that can 'fail gracefully' when they can't automatically rectify inconsistencies between the multiple ontologies. We cannot hope to iron out all inconsistencies between ontologies that our systems may encounter.

With this in mind I am not convinced that our current approaches to these kinds of issues (like the RDF graph of the Semantic Web) are sufficiently expressive to be able to handle multiple contradictory ontologies in one system where each is only relevant within specific contexts.

3 What is Semantic Programming?

At its core, Semantic Programming (SP from now on) is a proposal for the data model on which the one distributed, interconnected graph of all programs and data should be built. In light of the concerns raised in section 2 above, SP proposes the 'nodedge' as the basic building block for this distributed graph on which higher level computing should be built.

A 'nodedge' is the atomic unit of data that forms both the nodes and the edges of the SP graph. Nodedges are built out of references to other nodedges. The details of this nodedge data model are examined in detail later.

As with the relational model behind SQL databases and the RDF triple behind the Semantic Web, SP not only suggests a specific data model to use, but also defines the basic way that the model should be used and thought about. In the case of SP as much meaning content as possible has been stripped out of the underlying framework. On the other hand SP suggests a new way of working with computers and this opens up the possibility of a whole new class of specific programming languages that are natively consistent with SP thinking and thereby belong to the SP paradigm.

In the research to date work has been started on one such language, Semprola (*Semantic Programming Language*), but a lot of other work has been undertaken by using Java in a way that is consistent with the new SP paradigm, much as you could use C in an object oriented way. In this sense it is still very early days for the SP project.

3.1 What's in a name?

This research project has been on-going as a private interest for many years, and a number of different names have been used at different stages of the research. However, a central theme of the research from the earliest explorations has been the idea that:

Meaning arises from the relationships between things and the behaviours of things.

A longer description of the naïve semantic theory being used follows below. What's important here is that this research project is focused around how the references that build up a semantic graph come to mean what they mean for a given agent². Another way to put this is that the research looks at how do agents know what to do when they come across particular symbols or tokens that make up a reference in the semantic graph? How does the agent 'know' what the reference means?

By using the name 'Semantic Programming' the intention is to highlight this focus, it is obviously not an attempt to suggest that other programming paradigms do not address issues of semantics. After all, 'Functional Programming' is not the only paradigm interested in functions, nor does 'Logic Programming' attempt a primary claim to logic, nor is 'Object Oriented Programming' the only paradigm that uses encapsulated 'objects' and messaging.

Indeed, with the Church Turing thesis in mind it is fairly safe to assume that all programming paradigms are computationally equivalent. What is important about a paradigm is its underlying philosophy of how to program and the resultant types of computation that it thereby makes easy to program.

With a central focus on the meaning of references, it therefore seems appropriate to call this proposed paradigm 'Semantic Programming' (from now on we will use 'SP').

² We'll be talking about how 'agents' come to understand references, rather than 'computers' so that we can keep the term 'computer' to refer to an individual physical machine. By 'agent' we mean a specific type of program running on a given computer. We can therefore imagine multiple agents running on one computer.

3.2 A Naïve Semantic Theory

This research has not attempted to develop an original philosophical semantic theory, but rather it is built on a set of intuitive (and therefore no doubt derivative and slightly naïve) ideas that we can quickly explore here. These ideas are based around how 'agents' understand the meaning of 'references'. In this abstract context the term 'agent' refers to both humans and computer programs.

- A 'reference' has no atomic meaning in and of itself, but has meaning only in so far as an agent is able to manipulate, act upon or in some other way 'understand' the reference.
- An agent understands the meaning of a reference if it is able to use it effectively.
- The meaning of a given reference can depend on the context in which it's being used.
- An agent can come to understand the meaning of a given reference in a given context in one of three ways:
 - Axiomatically, or in other words by having the meaning of the reference 'hard wired' for a given context.
 - Deductively, through the reference's relationships to other known references within the given context (e.g. if the agent knows that 'X is the same as Y' and already understand the meaning of 'Y' then it can derive the meaning of 'X')
 - Behaviourally, through the reference's behaviour in the given context (if we can probe the reference's referent and get responses back then we learn something new about the meaning of the reference)

With this third mode of 'understanding' we can see the importance of another key idea:

- An agent must be embedded within an interactive world if it is to expand its understanding of the references it is using.

This also gives us a slightly hand wavy and dangerously circular handle on what we might have meant by saying that an agent is using a reference 'effectively':

- An agent is using a reference 'effectively' if its usage is understood by other agents in the world.

So, the notion of 'meaning' that we're working with is not purely based in formal logic nor is it based entirely in an embedded behavioural notion of normative meaning. It's a mixture of both.

3.3 A quick caveat

Some readers may at this point be starting to wonder if SP is an attempt to build computer agents capable of cognition comparable to that of humans. It is not. SP is merely an attempt to explore a new way of thinking about and using computers as tools for humans. It is still useful to think of computers as reference understanding agents in the world even if we mean a very muted form of 'understanding' that is (at this stage at least) far from being a true cognitive ability.

Neither is SP an attempt to explain human cognition on the basis of this naïve semantic theory. This theory is simply a useful sketch to underpin the thinking behind a practical approach for working with computers.

4 A Simple Formal Framework

In section 2 we looked at various concerns that lead to the suggestion that both programs and data should be thought of as part of one distributed graph. In section 3 we looked at a naïve theory for how the references that build up such a graph could come to have useful meaning for an agent.

To translate these ideas into a computational setting we need to develop a simple formal framework for the distributed graph and the agents that understand it. (Note that what is given here is not a tight technical specification of the framework, but rather a descriptive explanation of the framework).

There are many examples in computing where graphs are used to represent semantic relations, most notably the Semantic Web, Topic Maps and more generally any kind of semantic network. Many such existing approaches were looked at as part of this research to determine whether they could form the basis for what this project was trying to achieve.

Unfortunately, none of the existing approaches looked at satisfied the key requirements that this project was trying to address and therefore SP is based on its own framework for building a graph of semantic relations.

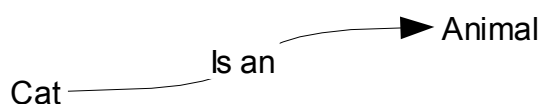
However, while the framework within SP is new, there are clear connections (and therefore potential synergies) between SP and other existing approaches to model semantics with graphs.

4.1 Introducing the 'Nodedge'

The classic graph is constructed out of a set of nodes and a separate set of edges. However, the graph in SP is built out of a single set of 'nodedges'. As the name implies these nodedge entities can all be referenced as a 'node' in the graph and each such nodedge in fact defines an 'edge' between various other nodedges.

So, what structure should these nodedges take?

The classic kind of edge within a semantic network graph can be thought of as a triplet of the form $\langle A, r, B \rangle$ expressing the directed relationship 'r' between 'A' and 'B'. In the Semantic Web RDF framework these are thought of as $\langle \text{subject}, \text{predicate}, \text{object} \rangle$ triplets. So for example the triple $\langle \text{'Cat'}, \text{'is an'}, \text{'Animal'} \rangle$ would be a typical edge within an RDF graph.



However, in SP we are going to require that each nodedge can be identified with an ID that is a globally unique ID (a GUID of some kind) so that at the very least each nodedge can be referred to by this GUID. This ensures that each nodedge can potentially be referenced as a 'node' within another nodedge.

So, in SP we at least need to use a quadruple of the form say: $\langle \text{'123'}, \text{'Cat'}, \text{'is an'}, \text{'Animal'} \rangle$

However, in SP we also want to be able to say much more about the context in which a given statement is said to be true. For example, we may wish to know who is making the statement? So, we could then end up with quintuple of the form $\langle \text{'Peter said'}, \text{'123'}, \text{'Cat'}, \text{'is an'}, \text{'Animal'} \rangle$. Or should the order be $\langle \text{'123'}, \text{'Peter said'}, \text{'Cat'}, \text{'is an'}, \text{'Animal'} \rangle$?

It is soon clear that using ever expanding n-tuples raises the problem of having to specify meaning by the order within the n-tuple. Not only is this a problem for continued extensibility, but it also goes against a natural desire to remove as much meaning as possible from the underlying data

model on which SP is built.

So, instead, the 'edge' information of a nodedge is held within a set of 'key / value' pairs. Within SP this collection is known as the *typed references* of the nodedge and are talked about as being a set of '*reference type / reference*' pairs. The *reference type* part is meant to help the agent understand the meaning of the related *reference* part.

In Java-like notation this part of the nodedge structure could be written as:

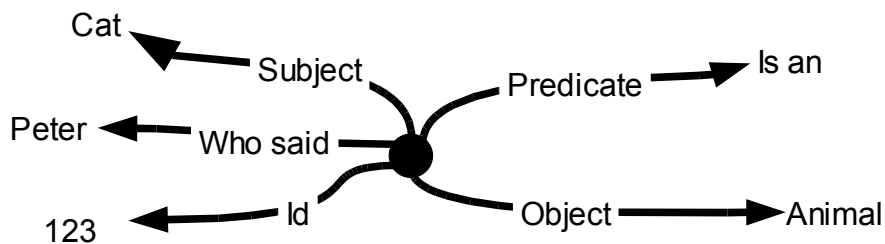
```
Map<Reference, Reference> typedReferences
```

Where *Reference* is obviously a class defining a reference to another nodedge. We'll come back to the structure of references later.

Now, with our developing nodedge structure, we are in the position to represent the fact that Peter said that a cat is an animal by using a structure like:

```
{ 'Who said':'Peter', 'Subject':'Cat', 'Predicate':'Is an', 'Object':'Animal', 'Id':'123' }
```

Or diagrammatically:

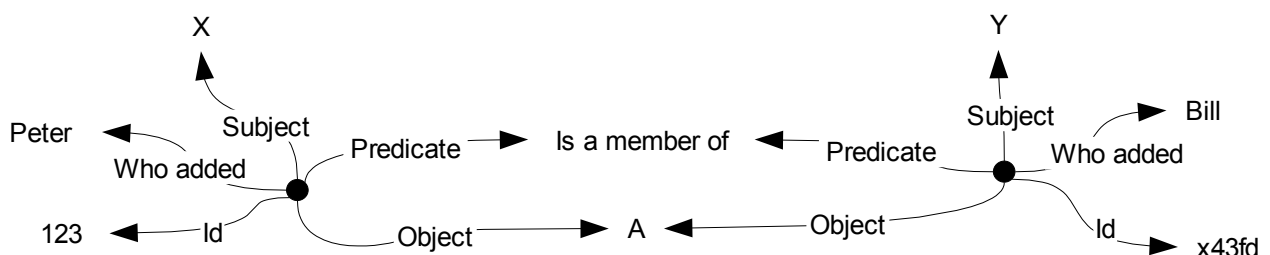


This is not quite complete for SP. We want everything within the SP graph to be a referenceable nodedge and we want there to be *only one global graph*. By 'one graph' we mean only one graph distributed across all computers involved with SP.

As things stands this 'one graph' requirement would lead to unreasonable search implications when trying to determine the meaning of certain parts of the graph. For example, if we want nodedge *A* to represent a set of things and we decide to use nodedges describing '*is a member of*' relationships to define the membership of *A*, then we could try to express the fact that *X* and *Y* are members of *A* with the two nodedges:

```
{ 'Who added':'Peter', 'Subject':'X', 'Predicate':'Is a member of', 'Object':'A', 'Id':'123' }
{ 'Who added':'Bill', 'Subject':'Y', 'Predicate':'Is a member of', 'Object':'A', 'Id':'x43df' }
```

Which could diagrammatically be viewed as:



The problem is that in order to know for certain the full membership of set A we would need to examine the nodedges held on every computer participating in the distributed SP graph to check for nodedges defining such '*is a member of*' relations. This is clearly impossible.

The solution that SP uses is to provide a way to limit the extent of such searches by allowing each nodedge to also hold a set of references. Again, in Java-like notation this makes the completed SP nodedge have the following structure:

```
class Nodedge {
    Map<Reference, Reference> typedReferences;
    Set<Reference> setOfReferences;
}
```

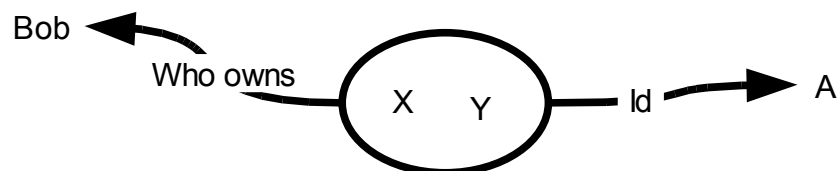
There is no default meaning of what membership of this *set of references* set is meant to mean. One way to look at this is that the typed references form the 'edge' like meta data that defines the context of the 'node' like set of references. It's also worth noting that the set of references to other nodedges defines a sub-graph of the entire SP graph.

Now to implement the set A with two members X and Y we have two basic approaches. One is to use the set of references of the nodedge representing A to directly hold the membership of A :

We could textually represent this as something like:

```
{ 'Who owns':'Bob', 'Id':'A' } & { 'X' , 'Y' }
```

Or we could represent this diagrammatically as:

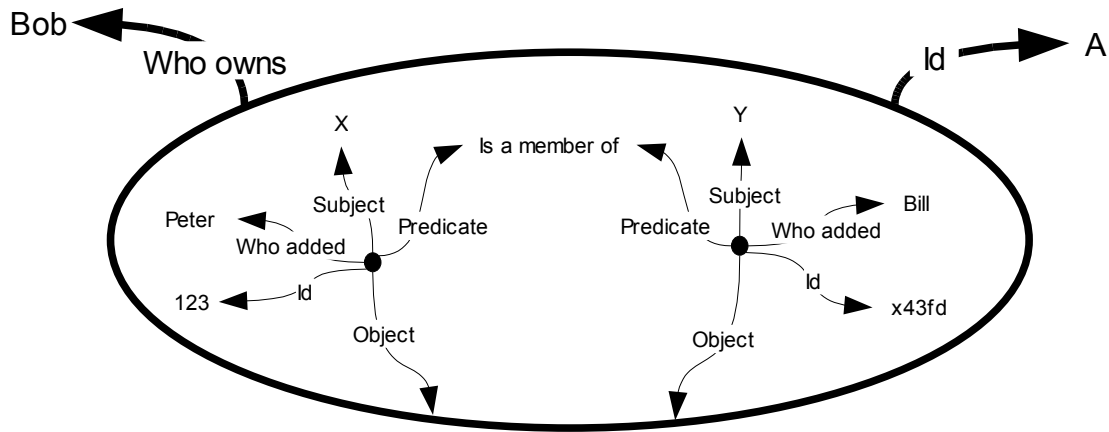


The second approach would be to have the set of references held by nodedge A determine a sub-graph that defines the set A . This second approach is more cumbersome, but does give the extra benefit of being able to hold extra information about each set membership, such as who added the given member to the set.

So, a textual representation of this could look like:

```
{ 'Who owns':'Bob', 'Id':'A' } & { '123' , 'x43df' }
{ 'Who added':'Peter', 'Subject':'X', 'Predicate':'Is a member of', 'Object':'A', 'Id':'123' } & { }
{ 'Who added':'Bill', 'Subject':'Y', 'Predicate':'Is a member of', 'Object':'A', 'Id':'x43df' } & { }
```

Or, diagrammatically:

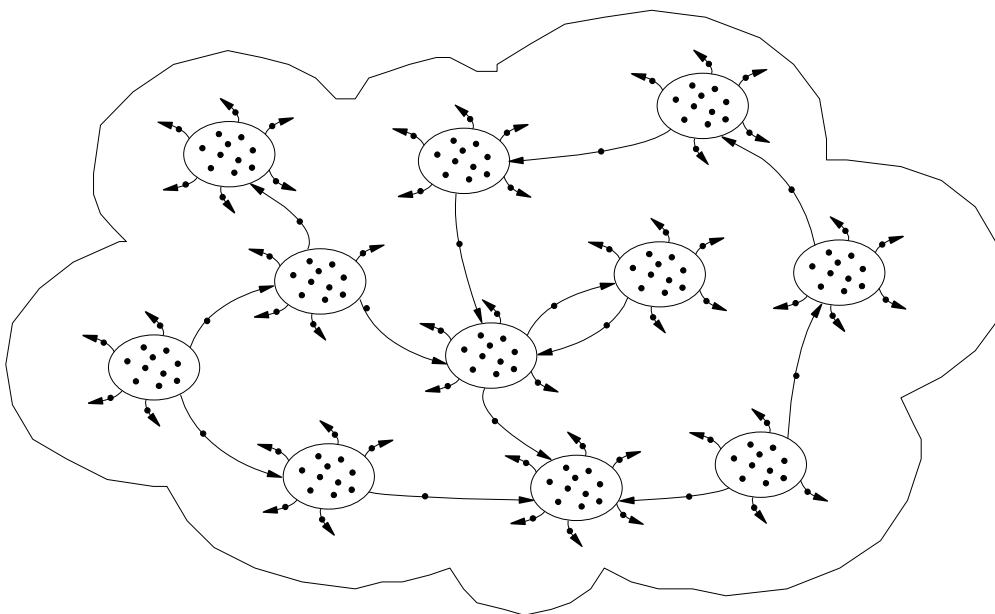


Finally, there are a couple of important points to note about the nodedge structure:

- All references should be thought of as references to other nodedges.
- However, the fact that a nodedge holds a particular reference does not imply that the supposedly referenced nodedge exists. So, the references are decoupled references.
- There is no default meaning to membership of a given nodedge's set of references.
- Any nodedge can hold references to any other nodedge that may exist globally.
- Conversely, each nodedge can be referenced by multiple other nodedges.
- Nodedge states are persistent over time (not just held in memory).
- Nodedges can hold a reference to their GUID as a typed reference by using a reference type that the agent recognises (as has been illustrated roughly in the examples above).

4.2 Only One SP Graph

As every nodedge can refer to any other nodedge we can think of there existing a single global cloud of nodedges that together form a single graph, which we will refer to as the 'SP nodedge graph' or simply the 'SP graph'.



4.3 What about the References?

Now that we have built up the nodedge in terms of references, what do the references themselves look like?

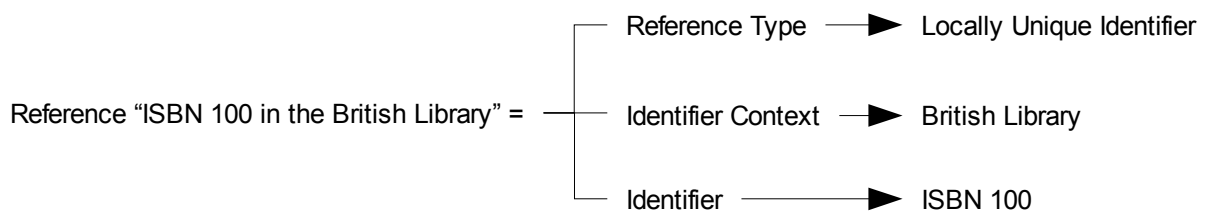
The question of how to identify things in the digital world is an issue that has been explored already by computer science, and the most familiar contemporary form of identifier used in relation to the internet is the URI (Universal Resource Identifier). None of the existing approaches to digital identifiers addresses the question of building a reference in the way that SP requires, although there are large areas of overlap and synergy to be taken advantage of.

Most importantly, we are looking for a data model of a reference, not a textual format for an identifier (although text identifier tokens will be needed). Furthermore, within SP, every reference must in theory refer to another nodedge.

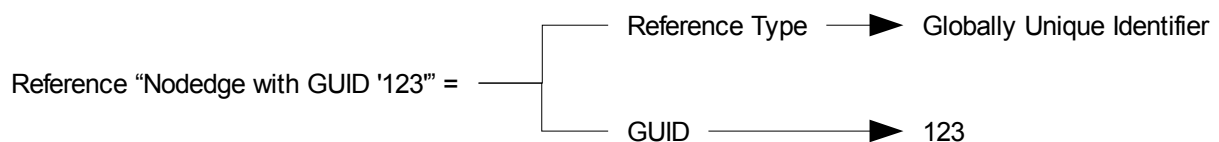
We can imagine (for starters) the following types of reference that we may wish to use:

- References to a nodedge using its GUID.
- References to the GUIDs themselves (in theory as a nodedge representation).
- References to a nodedge using a locally unique ID within a given context (such as for the nodedge representing the book ISBN 100 in the British Library)
- Logical references to a given nodedge given the context in which the reference happens to be used (such as a reference to 'my email account' where 'my' is to be understood in context at the time of usage)

Clearly the ability to represent these kinds of references will require at times more than just a single identifier token. Indeed, sometimes there will be a need for nested references (e.g. When a reference to a context is needed). For example, we will want to be able to say things like:



Where the 'British Library' on the right of the diagram is itself a nested reference to the nodedge representing the British Library. Similarly, if '123' were a globally unique ID (GUID) then we could also have a reference such as:



With these two examples in mind our References are starting to look like *key / value* mappings where the value can be either a Token or a nested Reference. But what about the keys? The meaning of a value can be determined by its key and the other key / value pairs, but how can the meaning of a given key be determined?

The solution to this potential infinite recursion is to stipulate that the keys can only be GUIDs that explicitly identify a given nodedge in a global, context independent manner. Indeed, these GUID keys are a form of simplified explicit reference, whereas the overall Reference structure is a more complex and versatile form of reference.

With all of this in mind we can again look at a Java-like notation for a Token, a GUID and a Reference:

```
interface ReferenceValue { }

class Token implements ReferenceValue {
    String token;
}

class GUID extends Token {}

class Reference extends Map<GUID, ReferenceValue> implements ReferenceValue {}
```

In the above notation the Token has been implemented as a String, but we could imagine binary tokens as well. The GUID extension of Token is there just to represent the extra requirement for the keys to be globally unique tokens. In implementation terms these GUIDs could be, for example, unique URIs, or XRI's or UUIDs.

As with the structure of the nodedge, this structure for a generic reference allows us to make almost no commitments at this stage to the way in which this structure should be used to define references. The references are built out of tokens and nested references and it is up to the agent using a given reference to 'understand' how to handle a given reference construction.

As the GUID tokens are known to be the unique identities of other nodedges, the agent can use these GUIDs to 'understand' the structure of the reference in one of the three ways mentioned earlier:

- axiomatically through having the meaning of some GUIDs hard wired
- deductively, through knowing relationships between the nodedge with the given GUID to other nodedges
- behaviourally, by probing the nodedges referenced by the known GUIDs

Most importantly, from the perspective of SP the GUIDs should not have any internal decomposable elements with meaning that needs to be understood by the agent. Traditional URIs often have specific hierarchical internal structures to the string that have specific meaning. Within SP this is completely discouraged, and any such structure should be ignored by the agent. A GUID is simply meant to be globally unique.

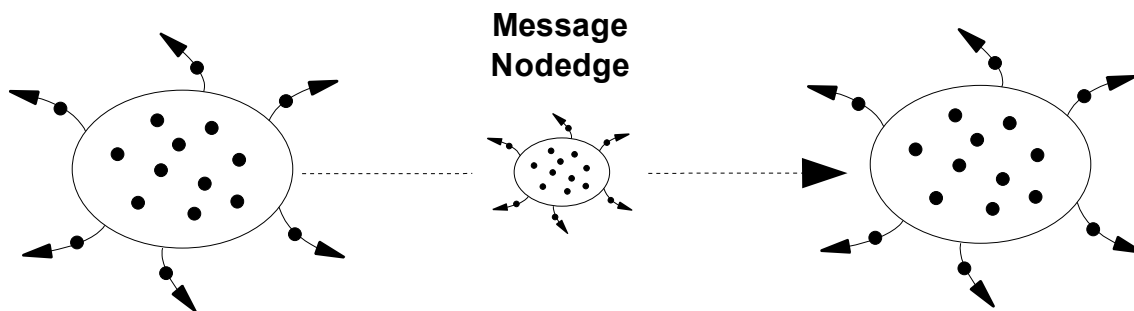
In particular, a GUID is definitely not meant to specify a location on the internet where the nodedge can be found. As we shall see later a single nodedge may be located on multiple machines, and indeed may move location during its lifetime.

Quick note: for ease of reading diagrams of nodedges will still use simple words or tokens to represent references, even though we have now defined more accurately the actual data model to be used for references in SP.

4.4 Messaging and Nodedge Behaviour

So far we have covered the way in which Nodedges are built out of References and how References are in turn built out of Tokens, but all of this describes a static data model, not a computational framework. Messaging and nodedge behaviour are the missing ingredients that animate the SP nodedge graph into a computational paradigm.

Computation is initiated when one nodedge sends a 'message' to another nodedge. These messages are themselves nodedges, although typically these message nodedges do not persist for very long.

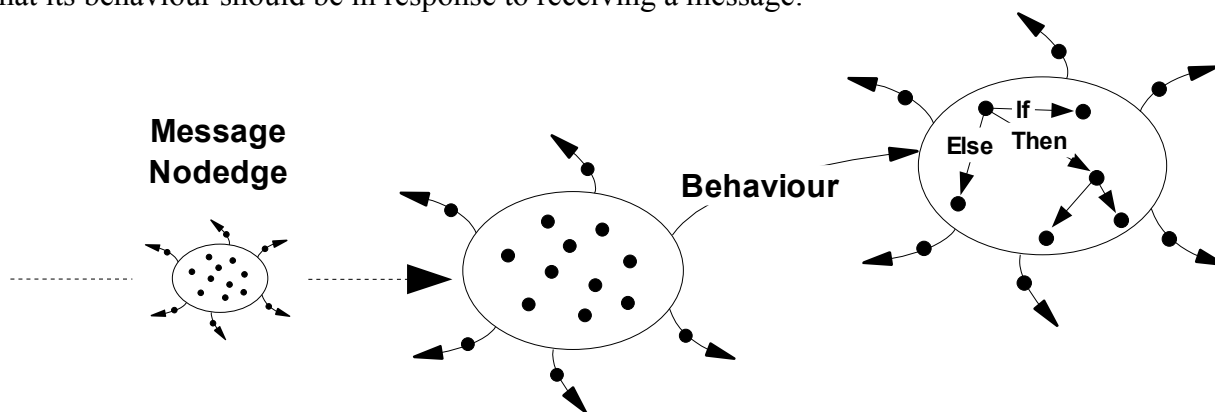


When a message arrives at a nodedge, the recipient of the message is said to 'behave' in response to the message. This behaviour can involve only the following types of action:

- Change the state of the behaving nodedge
- Delete the behaving nodedge
- Create a new nodedge or message nodedge.
- Send a message to another nodedge

The behaving nodedge only has direct access to examine itself, the received message nodedge and any new message nodedges that it creates. It can only directly modify itself or the new message nodedges it has created. All other interactions with other nodedges have to be mediated by sending and receiving message nodedges. Indeed, all other referenced nodedges might potentially only exist in agents across the other side of the world. An important consequence is that nodedges encapsulate their data from direct access by other nodedges in a similar way that an object does within the object oriented paradigm.

How the nodedge should behave is determined by its relationships to other nodedges. For example, we could imagine that the recipient nodedge has a typed reference of type 'Behaviour' that defines what its behaviour should be in response to receiving a message:



Sometimes the behaviour of the message recipient will include asynchronously sending back a response message to the original message sender. Indeed, it is expected that this will often be the

case, and this kind of request / response message pair is the key way in which the SP graph is expected to interact with computer programs running in other paradigms. From the perspective of other parts of the SP graph the 'gateway' passing messages to other systems would itself appear like a nodedge which happens to have the behaviour of the other system.

When a message arrives at a nodedge it's up to the agent holding this recipient nodedge to animate it to behave appropriately given what the agent knows about its defined behaviour. So, finally, it's time to look in more detail at what we mean by an 'agent'.

4.5 Agents

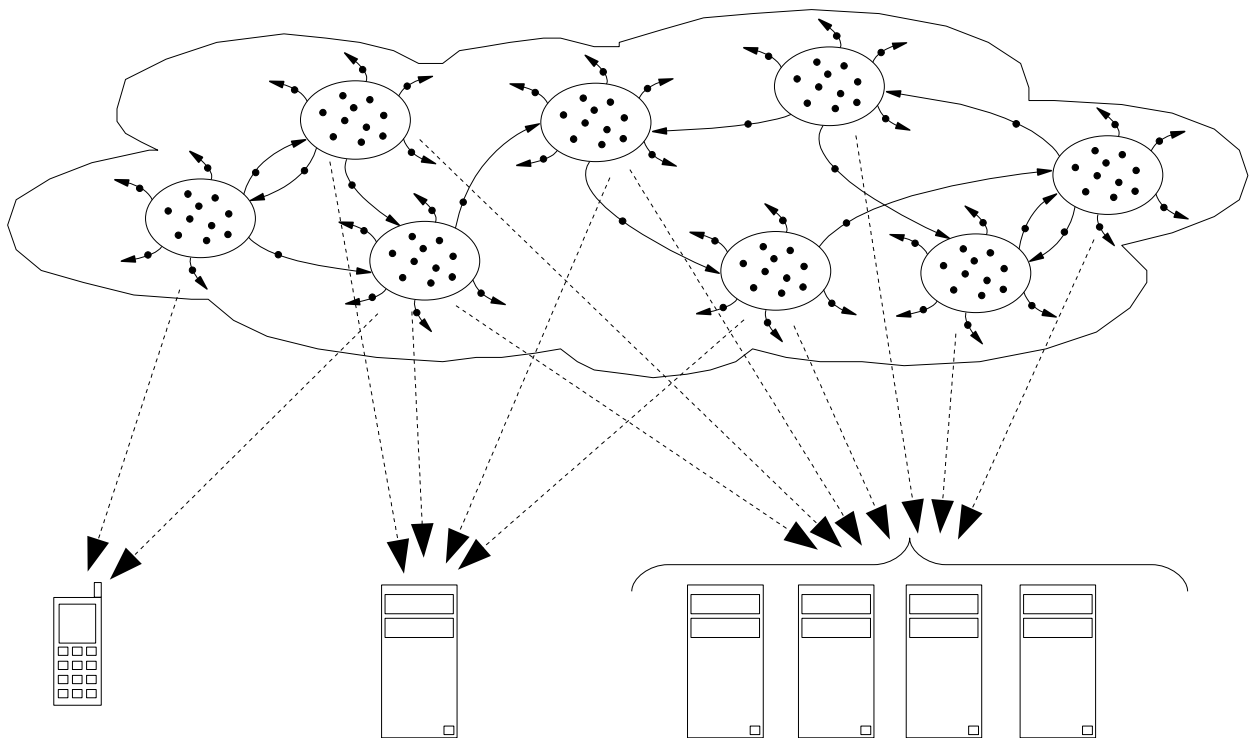
Agents are the computer programs that actually hold and manipulate a set of nodedges that form part of the global SP nodedge graph. Agents are free to add or remove nodedges from their local set of nodedges, and to change the state of these nodedges. Agents are themselves represented within the SP graph as a nodedge that can be referenced like any other nodedge. Agents also manage the message passing between nodedges and animate the appropriate behaviour of any nodedge that receives a message.

So, it is the agents that animate the SP graph. Doing this appropriately is part of what it means for an SP agent to be 'understanding' the SP graph effectively. We will look at this further soon, but first we need to look at the way that the agents replicate the state of nodedges in a peer to peer network.

4.5.1 Peer to peer replication

Part of the job of the agent is to ensure that appropriate peer to peer replication of nodedges occurs. The detail of how this is managed will be covered in future papers. What matters here is the idea that a given nodedge might be replicated on multiple agents, or just exist on one agent.

As there is not a simple one to one mapping between each nodedge and a single agent on which it lives, so the SP graph is not neatly partitioned into regions of the graph that live on this or that machine. Rather the SP graph is distributed variously across the agents.



The replication of a given nodedge across multiple agents can be viewed as an appropriate reflection of the fact that the 'concept' represented by the replicated nodedge is indeed 'known' to all of the agents that hold a copy.

As multiple agents on multiple machines may potentially attempt to modify the same nodedge at the same time all such activity is managed within transactions. Again, the exact logic of how these transactions are managed is beyond the scope of this initial introduction paper. The important point at this stage is that the agents need to manage appropriate distributed transactions between each of the agents 'participating' in the 'running' of a given nodedge. This ensures that the state of each nodedge remains appropriately consistent across all of its participating agents.

When one nodedge sends a message to another, sometimes both of the nodedges will exist within one agent and a single agent can animate the entire interaction. On other occasions the recipient nodedge may exist on a different agent, in which case the message must be sent to the other agent for it to forward on to the relevant nodedge. In this scenario the interaction is animated by two agents.

Crucially, it is the agent programs that must determine whether or not an appropriate copy of the recipient nodedge is held locally or not, and if not, where to find an remote copy in another agent. From the perspective of the nodedge sending the message there is only one big SP graph without any need to care about local / remote distinctions.

4.5.2 Why replicate in this way?

It is expected that a typical nodedge may end up replicated onto three or four agents, each of which is likely to be running on a separate physical machine. Such a nodedge, instead of having one location on the internet would therefore be located in four locations. Indeed, over the lifetime of a nodedge its location(s) on the internet may change many times.

Managing all of the extra information to be able to keep track of the current locations of nodedges, and the transactional states of distributed nodedges appears to be a large cost. The traditional temptation is to avoid this kind of complexity and rather build a set of buckets that each hold specific data that never moves, and then you don't need to worry about the extra complexities of replication.

The thinking behind SP is to suggest that now is the appropriate time for these extra costs to be surmounted once and for all within the foundation of a new approach to computing. By getting the agents to manage this peer to peer replication, SP allows programmers to focus on what we want to do with computers, rather than how to do the plumbing between the buckets.

If we can also use the information held within the nodedges to help tune the way in which the replication occurs (e.g. large data sets don't get replicated to mobile phones and important data is replicated more times than dispensable data) then the potential benefits of such a replicating peer to peer foundation are significant and include:

- A built in 'backup' of all important data – no need to treat this as a separate task
- A built in ability to route around temporary failure or network disconnection to specific physical machines.
- A natural sense that data is not rooted in one physical bucket, but can easily be moved around between agents.
- A natural sense that data can be generated or edited in one agent, but then intensive data analysis activities can happen on other, more powerful agents without losing the 'link' between the copies of the data.

- A solid foundation on which to build new programming environments that treat all data as potentially remote.
- A natural way to think about and implement data sharing between organisations (and individuals) where each organisation would hold its copy of the shared nodedge, but these copies could be meaningfully linked.

Overall, a key benefit of this approach is that it would allow us to move away from thinking and working with data in isolate buckets.

4.5.3 Animating nodedges appropriately

We've talked about how agents are meant to animate the SP graph 'appropriately', so how does this happen?

One way to think about an agent is to view it as a kind of virtual machine. While the Java virtual machine understands and runs programs and data in bytecode, so an SP agent understands and runs programs that are defined by nodedges. In particular (and as discussed in section 4.4) when a message arrives at a nodedge, the agent is meant to animate the triggered behaviour of the nodedge by examining the way that the receiving nodedge is related to other parts of the SP graph. As discussed in section 3, an agent comes to understand parts of the SP graph either axiomatically, deductively or behaviourally. We shall look at each of these in turn.

4.5.3.1 Understanding the SP graph axiomatically

In a sense we could say that the Java virtual machine (JVM) understands bytecode axiomatically as this understanding is hard wired into the JVM program. In the same way agents can have the meaning of parts of the SP graph hard wired.

To explore how this works we're going to look at a toy example of a simple nodedge whose behaviour is to delete itself upon the arrival of any message. To do this we're going to need to build up the data model for this nodedge more accurately than we've done so far.

First we're going to imagine a type of URI based GUID that we can guarantee is unique. Let's pretend we own the domain name "axiomatic.guid" and hence we can generate our own GUIDs of the form "axiomatic.guid/123" and know that no one else has authority to create GUID strings like this.

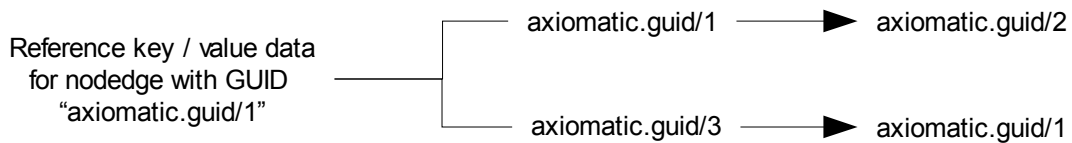
Now we're going to start generating some specific GUIDs and labelling them to represent specific meaning. First we need to build up the GUIDs for the agent to be able to understand explicit References to nodedges. The 'Easy GUID Label' in the right hand column is only for making diagrams in this paper easier to read.

GUID	Intended meaning	Easy GUID Label
axiomatic.guid/1	What kind of nodedge Reference is this?	KIND_OF_REFERENCE
axiomatic.guid/2	An explicit nodedge reference using a GUID	GUID_REFERENCE
axiomatic.guid/3	What is the GUID for the referenced nodedge?	NODEDGE_GUID

Now, as stated earlier we want all GUID keys that build up a Reference to be thought of as a form of simple reference to a nodedge. Hence the GUIDs in the table above are thought of as uniquely

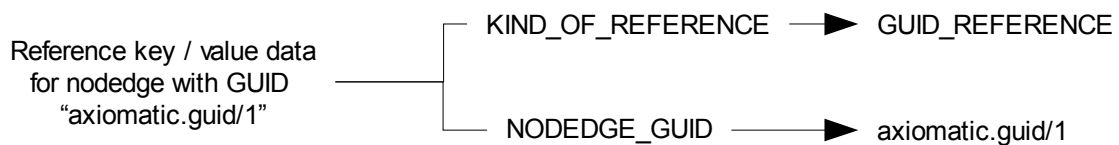
identifying nodedges. So, we can now represent the key / value data that would form the Reference for the nodedge that should be represented by the GUID “axiomatic.guid/1” given above.

If we draw this data using the raw GUIDs that would actually form the key / value data structure then this would look like:



If we ignore the 'Intended meaning' column in the table above then this data structure appears to the eye as fairly meaningless. Indeed, it's actually quite important to the SP approach that the underlying nodedge data is *not* intended to be directly human readable, but only machine readable.

For our ease of reading here we will now look at this data structure again with some of the GUIDs replaced by their 'Easy GUID Label' from the table above (but please note that this is not the way that SP handles human readable labels).



Like this the diagram makes sense to a human reader. One way for the Reference data to make sense to an agent is for the agent to be directly programmed to understand the intended meaning of the three specific GUIDs in the table above. This way the agent would 'axiomatically' understand the Reference structure to be an explicit reference to the nodedge with GUID “axiomatic.guid/1”.

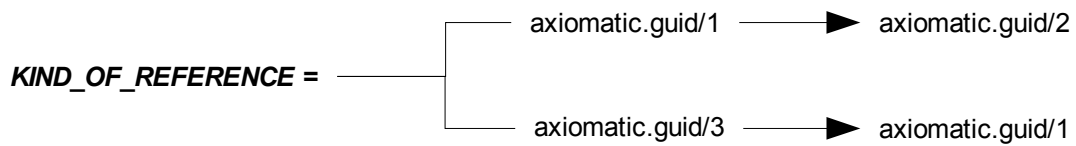
What about a full nodedge data structure? For this we need a few more GUIDs:

GUID	Intended meaning	Easy GUID Label
axiomatic.guid/4	What is the behaviour of this nodedge?	BEHAVIOUR
axiomatic.guid/5	What is the explicit GUID reference to this nodedge?	MY_GUID_REFERENCE
axiomatic.guid/6	Nodedges with this behaviour should be deleted when any message arrives as the nodedge.	DELETE_ON_MESSAGE_ARRIVAL
example.guid/1	Our example nodedge	EXAMPLE_1

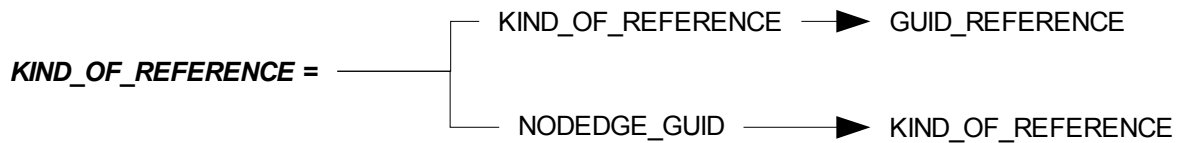
Notice that the last GUID here is for a different domain name, “example.guid”, which we will use in our examples here for GUIDs whose intended meaning is not understood by our agents axiomatically.

To further aide our ability to draw human readable diagrams we are also going to establish (just for this paper) an 'Easy Reference Label' that can be used in place of the full Reference structures for referring to the GUIDs that we have defined. For this we will simply use a bold, italicised version of the 'Easy GUID Label' that we've already given for each GUID.

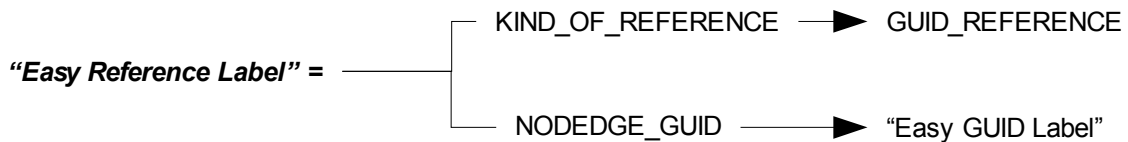
So, the explicit reference for the nodedge with GUID “axiomatic.guid/1” that we have used above would take the 'Easy Reference Label' of **KIND_OF_REFERENCE**. Hence we can draw this as:



Or replacing the GUIDs with the “Easy GUID Labels”:



Or more generally, for any “Easy GUID Label” the associate “Easy Reference Label” is defined as:

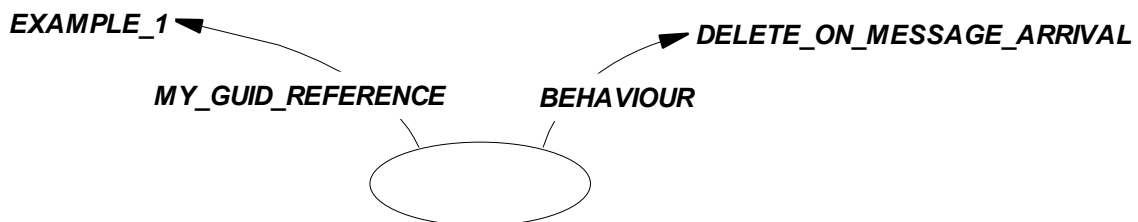


Again, just to be really clear, this convention is just to help make this paper easier to read, it has nothing to do with how human readable labels are handled within SP.

Now, with all of our GUIDs and associated labels defined we can build our example nodedge with the simple 'delete on any message arrival' behaviour. First in the simple text format we were using in section 4.1:

```
{ 'MY_GUID_REFERENCE' : 'EXAMPLE_1', 'BEHAVIOUR' : 'DELETE_ON_MESSAGE_ARRIVAL' } & {}
```

Or diagrammatically:



Now, I hope, it has become clear what it would mean for an agent to 'understand' this example nodedge 'axiomatically'. The agent's program would have to have all of the GUIDs described here hard wired into its programming in the appropriate way so that when a message does arrive at our example nodedge, the agent deletes the nodedge and thereby has successfully 'understood' how to correctly animate this nodedge.

4.5.3.2 Understanding the SP graph deductively

So now that we've got our toy example off the ground with our agent understanding the 'delete on message arrival' behaviour axiomatically, what would it mean for the agent to understand a behaviour deductively.

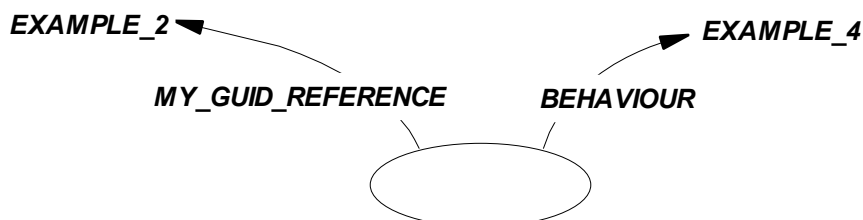
To further build upon our toy example we're first going to need some more GUIDs with specific intended meanings that our imagined agents are going to understand axiomatically. These are:

GUID	Intended meaning	Easy GUID Label
axiomatic.guid/7	Nodedges with this behaviour should do nothing when a message arrives	DO_NOTHING_BEHAVIOUR
axiomatic.guid/8	What is the subject of this nodedge?	SUBJECT
axiomatic.guid/9	What is the predicate of this nodedge?	PREDICATE
axiomatic.guid/10	What is the object of this nodedge?	OBJECT
axiomatic.guid/11	Predicate: "Has identical meaning"	IDENTICAL_MEANING

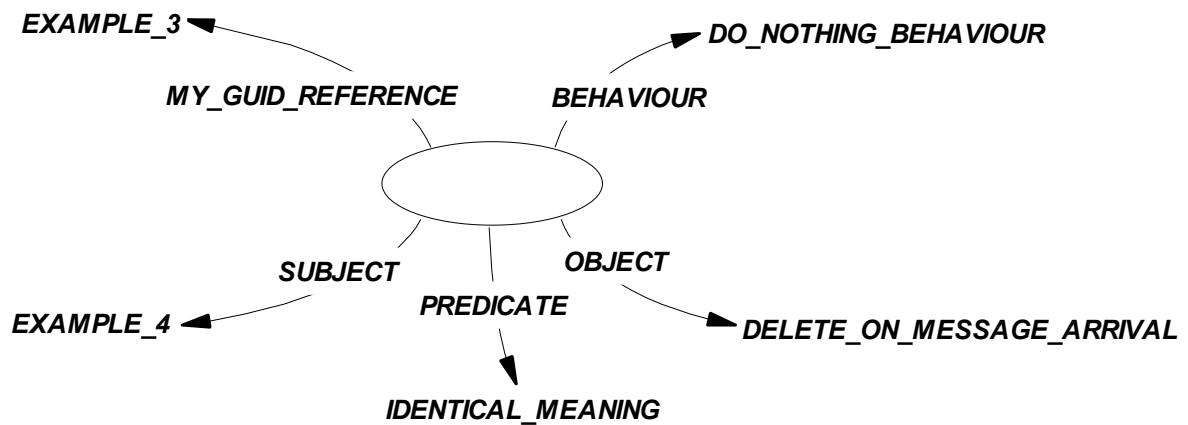
And we're also going to define three new example GUIDs that are not axiomatically understood by the agent:

GUID	Intended meaning	Easy GUID Label
example.guid/2	An example nodedge	EXAMPLE_2
example.guid/3	An example nodedge	EXAMPLE_3
example.guid/4	An example nodedge that we want to also mean the 'delete on message arrival' behaviour but the agents do not have this programmed axiomatically.	EXAMPLE_4

Now, using the same 'Easy Reference Label' convention as before, and this time just looking at the diagrammatic version we can imagine that the agent holds the following nodedge:



Now with this **EXAMPLE_2** nodedge alone, the agent cannot understand what the intended behaviour is. However, if the agent also holds and 'understands' the following RDF like nodedge (nodedge **EXAMPLE_3**) :



Then the agent can deduce the intended action to take when a nodedge has **EXAMPLE_4** as its behaviour reference. So, now if a message were sent to the nodedge **EXAMPLE_2** the agent should correctly delete **EXAMPLE_2**.

Hopefully this toy example has illustrated what we mean by saying that an agent can come to understand the meaning of parts of the SP graph deductively.

Note that within SP it is seen as 'good practice' for all nodedges to have a behaviour of some kind explicitly referenced. That's why in the case of **EXAMPLE_3** above we've used a 'do nothing' behaviour.

As the 'edge' information held by this nodedge contains the <subject, predicate, object> triplet as used by an RDF triple, so we could have defined some kind of behaviour to reflect this fact (which we could have labelled here as **RDF_BEHAVIOUR**). In roughly this kind of way we can start to see how SP could interoperate with all RDF ontologies and inference engines. The RDF triple represented by nodedge **EXAMPLE_3** could be drawn as:



4.5.3.3 Understanding the SP graph behaviourally

Finally, what does it mean for an agent to understand parts of the SP graph behaviourally?

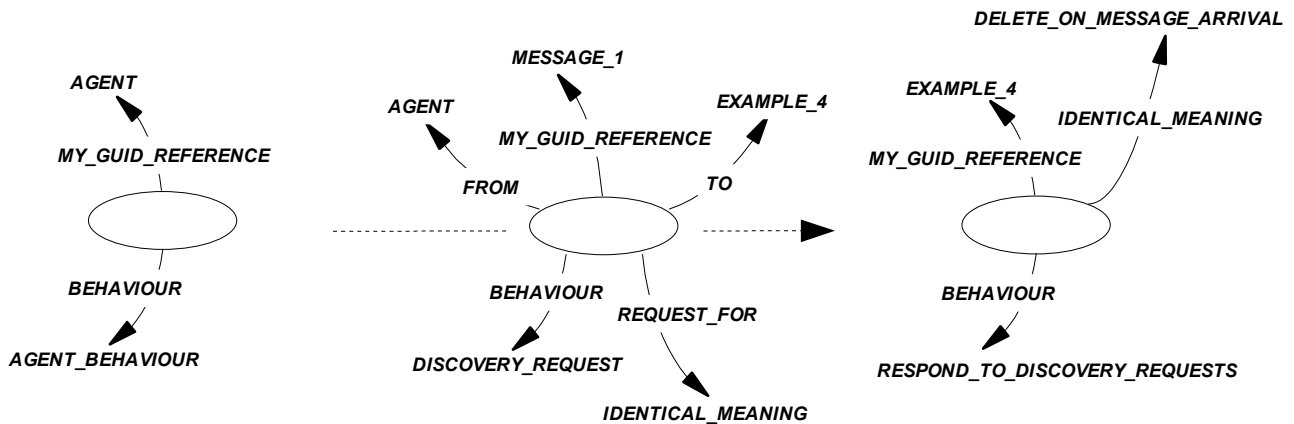
Imagine our agent only had the nodedge **EXAMPLE_2** from above and did not hold **EXAMPLE_3**. Therefore the agent would not be able to make the deduction that **EXAMPLE_4** has the identical meaning as **DELETE_ON_MESSAGE_ARRIVAL** and therefore the agent does not know how to animate the nodedge **EXAMPLE_2**. The agent simply does not understand references to **EXAMPLE_4**.

However, the nodedge structure of **EXAMPLE_2** has a reference to **EXAMPLE_4** and so the agent can use this reference to send messages to **EXAMPLE_4**. In this way that agent can 'probe' **EXAMPLE_4** to try to discover how **EXAMPLE_4** is related to other nodedges, and thereby what it is meant to mean.

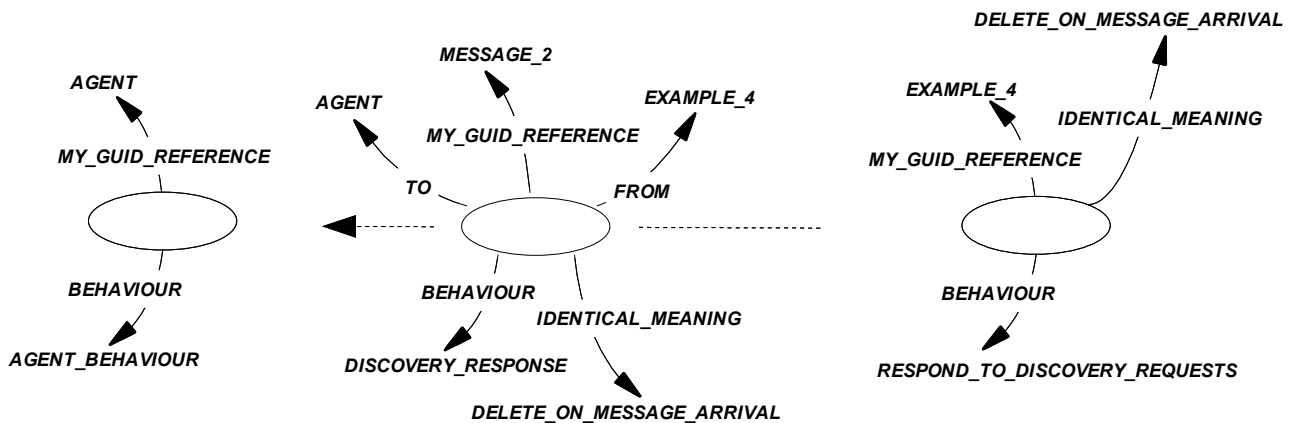
To continue with our toy examples, we'll quickly look at some simplified messages to illustrate this kind of 'discovery' of the meaning of **EXAMPLE_4**. As before we need to define some more nodedges for our example, but this time we'll skip the table and jump straight to 'Easy Reference

Labels' that will hopefully be sufficiently self explanatory.

So, we're going to use the nodedge **AGENT** to represent the agent that is trying to understand the meaning of the **EXAMPLE_4** nodedge that is referenced by **EXAMPLE_2** as its behaviour. In the diagram below we're trying to depict the nodedge **AGENT** sending a message, **MESSAGE_1**, to the nodedge **EXAMPLE_4**. It is a 'Discovery Request' message asking for a reference to an identical meaning for the recipient **EXAMPLE_4** nodedge.



We can then imagine that nodedge **EXAMPLE_4** responds with the message **MESSAGE_2** giving the requested information:



When the agent receives this response message (and assuming that the construction of the response message is already 'understood' by the agent either axiomatically or deductively) then the agent will have received the necessary information to now know that **EXAMPLE_4** has identical meaning to the nodedge **DELETE_ON_MESSAGE_ARRIVAL** and therefore it can now correctly animate the nodedge **EXAMPLE_2**.

Hopefully this toy example has given a sense of what it means for the agent to 'probe' a reference with messages and see how it behaves in response to these request messages. If the response messages are intelligible to the agent then it can learn about the reference.

4.5.3.4 Agents as virtual machines that can learn

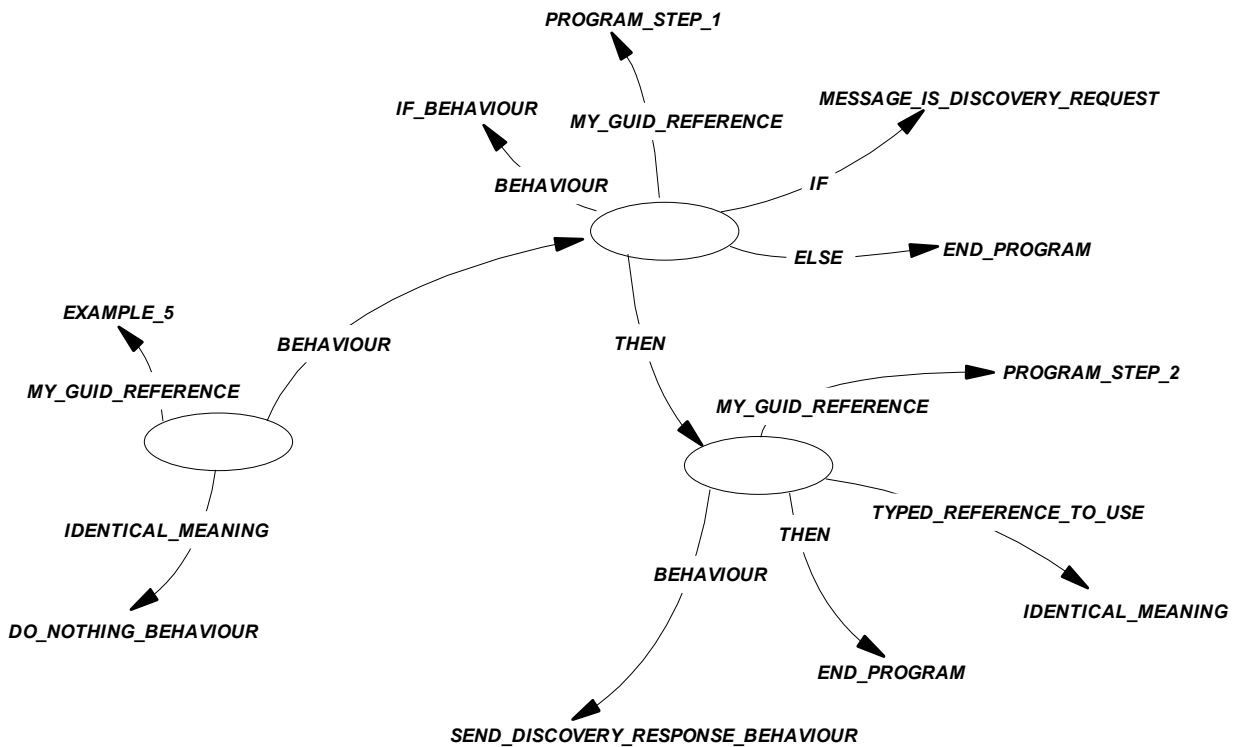
We've looked at the three key ways that agents can 'understand' the SP graph. Consequently, one way to view the agents are as virtual machines that can learn to interact with the SP graph beyond their initial axiomatic programming. This, for example, is in stark contrast to a JVM which will only ever be able to interpret Java bytecode.

This is only 'one way' to view the agents as they are also helping to persist the nodedges in a transactional peer to peer manner that could be contrasted to a traditional database, and they are also propagating messages between the nodedges across the network using location independent GUIDs in contrast to approaches that use location specific URLs.

4.5.4 Expressing programs in the SP graph

In the toy examples given in the previous section the 'delete on message arrival' behaviour was represented by just one nodedge with a specific GUID. A more interesting approach to defining behaviour within SP is to use part of the SP graph to describe the program for a given behaviour.

Again as a toy example we could imagine the following nodedges:



The two nodedges on the right form a very simple two step program. Remember that this is a program to be run by the agent when a message arrives at the nodedge **EXAMPLE_5** on the left of the diagram.

So, upon message arrival program step 1 checks whether or not the message was a discovery request (from the earlier toy example) and if so then it proceeds to step 2, otherwise the program ends. Step 2 then constructs and sends back the response message by examining the *typed references* of the nodedge running the program (**EXAMPLE_5** in this case) to find out whether there is an 'identical meaning' reference that can be sent back to the message requester.

Notice how this program could be used by many nodedges to define their behaviour.

4.5.4.1 The SP graph is the data model not the user interface!

Clearly the diagram above shows a very simple toy example, and it also clearly shows that drawing these kind of 2D diagrams would be a very difficult way to program. Indeed, it would be impossible to seriously consider programming computers by drawing such graphs.

So, it is vital to remind the reader that one of the concerns that motivated this research (discussed in section 2.1) is that we still think of programming as something that can be done with pen and paper away from a computer. Within the SP paradigm programming without the aid of a computer is practically impossible.

However, once you have a computer helping you program, then much of the complexity of the underlying SP graph can be hidden from the user. The SP graph is to help the agents, not the humans. The agents can then help the humans.

Indeed, one key difference between the needs of a human and a computer agent is that humans are great at understand the wider context in which some data belongs. Humans find it confusing if too much context information is repeated again and again. Conversely, computer agents need to have at least some explicit context information in order to 'understand' data. Often this context information is hard wired into the program, thus allowing the program and the humans to both deal with 'naked' data.

In contrast, SP encourages at least some context information to always link any nodedge to a wider context from which the agent could at least attempt to learn more about a nodedge that it doesn't fully understand. Some people say that 'information' is data within a context, and in this kind of way SP tries to let the agents work with information and not just naked data.

However, the result is a data model that is virtually unintelligible for humans without the aid of an agent. With ubiquitous computing upon us, there is hardly a shortage of computers to fill this role.

4.5.5 Agents that 'natively understand' a standard set of nodedges

If we call the set of nodedges listed in our toy examples above with GUIDs from the "axiomatic.guid" domain name as the "Example Standard", then we can say that a particular agent 'natively understands' the Example Standard if it is programmed to axiomatically know what to do when it encounters references to any of the nodedges in this set.

We use the term 'natively' only in relation to the agent's ability to understanding an entire standard set of nodedges 'axiomatically'.

As we shall see in section 5, it is through building up such standard sets of nodedges that SP is ultimately able to ground the SP graph. Agents can then be built that natively understand some of these defined standards.

Crucially, SP insists that such standard sets of nodedges do indeed behave as referenceable nodedges to which agents can send discovery messages. This should mean that we could have multiple standards understood natively by different agents within the SP graph. The intention is that through deduction and behavioural interactions agents that natively understand different standards could still interact successfully.

For example, agent program *C* could, let's say, be programmed in the procedural language C and be programmed to understand axiomatically a set of references called *C-Standard*.

We could then also program an agent program *J* written in the object oriented programming language Java that is programmed to understand axiomatically a different set of references *J-Standard*. In theory we could then build appropriate mappings between *C-Standard* and *J-Standard* that agents running the programs *C* and *J* could respectively 'learn' so as to be able to communicate

effectively with each other.

However, for the moment the research into SP has gone down the route of trying to build up a generic standard called 'Standard SP'. It is hoped that all SP agents initially programmed will be able to natively understand Standard SP. Some further details about Standard SP are in section 5.

4.5.6 Optimising agent performance

Interacting directly with the SP graph may not always be the most efficient way to run programs or analyse data. Therefore we can imagine programming our agents to compile programs from the SP graph into a lower level format, such as bytecode or machine code. This compilation, however, should be hidden from the end user as merely a dynamic form of performance optimisation, rather than a manual task that the user has to perform.

Similarly we can imagine at times the 'data' parts of the SP graph to also be compiled down into formats that are more efficient to process. For example we may make 'compiled' search indexes, databases or inference engines that take the source of their data to be a certain part of the SP graph.

What is important is that the SP graph remains the original source of all information with the SP world. The automatically compiled programs and data are just optimised copies being managed by the agents.

4.5.7 Agents as connectors

The agents also form the connection between the SP graph and the users and other computer programs. Typically there would be just one agent running on any given computer that is participating in the SP graph. This agent would also act as a 'gateway' between the SP approach to computing and any other programs running on that computer.

These connections will often be implemented as 'virtual' nodedges that behave like a nodedge from the perspective of the rest of the SP graph, but are actually gateways to other systems. One can also imagine defining messages that can be sent to the agent to trigger specific interactions with other systems or the computer on which the agent lives.

4.6 Summary list of the core SP framework components

Now that the main building blocks of the core SP framework have been described, they can be listed as:

- Tokens (character strings or binary)
- GUIDs (Tokens that are known to be globally unique IDs of some kind)
- References (built out of GUIDs, Tokens and other References)
- Nodedges (built with a *typed reference / reference* map and a set of References)
- Message nodedges (sent between nodedges)
- Nodedge behaviour (triggered in response to receiving a message nodedge)
- Agents (the 'virtual machines' of SP that:
 - replicate persistent nodedge states using distributed peer to peer transactions
 - propagate message nodedges to an appropriate copy of a nodedge
 - animate nodedge behaviour as triggered by received message nodedges).

One final, but important point to note about this framework for SP is that it has been designed to try to minimise the amount of hard wired 'meaning content' in the underlying framework itself. Even 'positional' and 'type' meaning as been minimised where possible with three clear exceptions:

- The keys within the References are asserted to be globally unique GUID tokens for nodedges that are used to define the meaning of their associated values.
- The Reference structures are asserted to be references to other nodedges
- The *reference type* keys of the typed references in the nodedges are expected to indicate the meaning of the *reference* value.

Conversely:

- There is no expectation or requirement in general that the Tokens (other than the GUIDs) have any particular meaning.
- The meaning of the nodedges is completely undefined by the general framework
- The way in which behaviours are actually defined is completely undefined by the general framework.

Indeed, to go from this basic SP framework to a usable computation paradigm there is a lot of 'meaning content' that still needs to be built up. Crucially, within SP all of this meaning content should be built up as part of the SP graph and thus have potentially equal status with any other part of the meaningful semantic network that the agents can 'understand'.

So, the basic SP framework described here is intended to form the backbone data model to get the global SP graph off the ground with as little hard wired meaning in its structure as possible. We can then hang as much meaning as we want onto this meaningless framework.

To end this paper we will take just a brief look at the way such meaning can be built up.

5 Beyond the basic framework

As described earlier in the paper, we can talk about a particular agent understanding an entire standard set of nodedges axiomatically. In this circumstance we say that the agent understands the standard natively.

As the basic SP framework contains virtually no meaning content, we need to build up the SP semantic graph by creating such standards and then building agents that natively understand these standards.

To date the research into SP has looked at developing two such standards:

- Standard Semantic Programming
- Semprola

We shall very briefly take a look at what each of these standards covers.

5.1 Standard Semantic Programming (Standard SP)

It is (again) beyond the scope of this paper to go into the details of Standard SP, but rather we shall briefly take a look at the kinds of issues that it tries to address.

Standard SP comprises a set of nodedges with specified GUIDs, together with the description of the intended behaviour of an agent that can 'understand' these Standard SP nodedges axiomatically.

There is currently work in progress to develop an agent program (written in Java) that understands Standard SP natively. The key areas covered by Standard SP are:

- Standard way of generating GUIDs required to build up basic types of References.
- Standard ways to represent common computer data types (integers, strings, etc).
- Standard way to represent assertions from which the agent can make deductions.
- Standard request / response message nodedge structures
- Standard notification message nodedge structures
- Standard approach to discovery of the possible interactions available with another nodedge.
- Standard ways to specify the program for the agent to execute in order to animate the behaviours of nodedges in response to messages
- Standard ways to manage the information required for the peer to peer replication of nodedge states within distributed transactions.
- Standard ways to reference digital identity information for authentication and authorisation.
- Standard information to hold with each nodedge (such as its GUID and the last digital identity to modify the nodedge)

5.1.1 Connecting to other paradigms

With Standard SP in place it is then possible to build further layers into the SP graph. In particular it is possible to use the request / response message pairs to link portions of the SP graph to other computer systems and thereby to the user. Of particular interest to make SP relevant to the modern setting of existing systems, the request / response message pairs can be used to make cross paradigm connections to (for example):

- Method calls within object oriented programming
- REST interactions across HTTP
- Service oriented XML messaging architectures
- SQL query calls to databases.
- File system access
- Event driven user interfaces

Similarly bridges can be built between the SP graph and other forms of semantic networks such as RDF graphs and topic maps.

5.2 Semprola

Finally, it is worth briefly mentioning Semprola, the first *Semantic Programming language*.

As described in section 4.5.4 we can imagine building up a part of the SP graph to described a program for animating nodedge behaviour. Semprola is the first attempt to define the necessary set of nodedges to be able to describe procedural programs within the SP graph itself.

If an agent is programmed to natively understand Semprola, then it will be able to interpret and therefore execute programs constructed using Semprola within the SP graph. The Semprola standard builds upon Standard SP and therefore uses the same request / response and notification message structures as defined by Standard SP.

Thus, with Standard SP and Semprola in place, both 'programs' and 'data' can be represented and manipulated within the SP graph. Semprola will be explored further in a future paper.

6 Final word

In some ways Semantic Programming can be seen as both a revolutionary and an evolutionary proposal. It is revolutionary because it proposes a fundamental change to the way that we work with computers, replacing the dominance of text files and buckets of data with instead a single distributed nodedge graph. At the same time it is evolutionary in the sense that the SP framework draws much inspiration from existing, familiar approaches to computing.

In particular many readers will have noticed the influences from object oriented programming, event driven programming, service oriented architectures, semantic web and peer to peer networks to name but a few.

Furthermore, SP, and in particular Standard SP, have been designed to be able to usefully interoperate with existing systems as is essential if SP is to have any practical use. These practical issues will be addressed in later papers.

6.1 Further Information

Any comments or questions should be sent to Oli Sharpe directly via email at oli@gometa.co.uk
For further information about Semantic Programming, please visit the website:

www.semantic-programming.org