A Scalable Vector Symbolic Architecture Approach for Decentralized Workflows

Chris Simpkin*, Ian Taylor†, Graham A. Bent‡, Geeth de Mel‡ and Raghu K. Ganti§
*Cardiff University, UK
Email: {SimpkinC,Taylorij1}@cardiff.ac.uk
†IBM Research, UK
Email: {Gbent,geeth.demel}@uk.ibm.com
‡IBM TJ Watson Research Center, USA
Email: rganti@us.ibm.com

***Abstract*—Vectors Symbolic Architectures (VSAs) are distributed representations that combine random patterns, representing atomic symbols across a hyper-dimensional vector space, into new symbolic vector representations that semantically represent the component vectors and their relationships. In this paper, we extend the VSA approach and apply it to decentralized workflows, capable of executing distributed compute nodes and their inter-dependencies. To achieve this goal, services must be discovered and orchestrated in a decentralized way with the minimum communication overhead whilst providing detailed information about the workflow - tasks, dependencies, location, metadata, and so on. To this end, we extended VSAs using a hierarchical vector chunking scheme that enables semantic matching at each level and provides scaling up to tens of thousands of services. We then show how VSAs can be used to encode complex workflows by building primitives that represent sequences (pipelines) and then extend this to support full Directed Acyclic Graphs (DAGs) and apply this to five well-known Pegasus scientific workflows to demonstrate the approach.**

Keywords–vector symbolic architectures; decentralized workflows; semantics; associative memory models.

I. Introduction

Workflows provide a robust means of describing applications consisting of control and data dependencies along with the logical reasoning necessary for distributed execution. For wired networks, there have been a wide variety of workflow systems developed [1]–[10]. A scientific workflow is a set of interrelated computational and data-handling tasks designed to achieve a specific goal. It is often used to automate processes, which are frequently executed, or to formalize and standardize processes. The majority of existing workflows rely on centralized management and therefore require a stable endpoint in order to deploy such a manager. In more dynamic settings, such as Mobile Ad Hoc Networks (MANETs) [11] where more collaborative applications (e.g., multi-user chats, or distributed analytics for coalitions [12], [13]) are needed, on demand workflows that are capable of spontaneously discovering multiple distributed services without central control are essential. The resulting distributed pathways are complex, and in some cases impossible to manage centrally because they are based on localized decisions, and operate in extremely transient environments.

In the current state-of-the-art, discovery of workflow steps is not dynamic; the exact services to be used for each workflow step must be specified in advance and, further, workflow orchestration is typically managed via a central point of control. Macker and Taylor [14] provides a mechanism that is decentralised, however, the specification of each workflow step, ip-address, connections and data, must be known in advance and is passed around between services via a JSON data block. Wittern et al. [15] present a graph based data model that aims to capture relationships between services and their usage detail for API ecosystems; the approach provides an interface that enables consumers to search for required services and service providers to obtain usage and compatibility statistics between the services and those of other providers as well as discovering new requirements. In contrast, we describe the use of VSAs to enable individual services to be self-describing and to learn contexts for which they are compatible. We describe a decentralised, multicast, environment where individual services can interrogate VSA encoded workflow messages, compute their compatibility to participate in a particular workflow step and be chosen if they are the best match. This is analogous to a group of humans listening to various work requests and deciding for themselves that they are capable and available to do a piece of work, i.e., all humans can understand the work request message and each human knows for themselves if they are available and capable of doing the the work. Naturally, more than one person may offer to do a particular job and we describe mechanisms for negotiating who gets to do the work.

However, applying VSAs to workflows requires several extensions and our contributions to this area can be summarized as follows:

- **Scaling Through Chunking:** To address scalability, we extend VSAs using a hierarchical vector chunking scheme that is capable of binding multiple levels of abstraction (workflow and sub-workflows/branches) into single vector. This approach scales to tens of thousands of vectors while maintaining semantic matching.
- **Encoding Workflows:** We employ our chunk encoding scheme to encode and decode very large sequences of services.
- **Representing Workflows Primitives:** We extend the encoding scheme to support Directed Acyclic Graph (DAG) workflows having one-to-many, many-to-many, and many-to-one connections.
- **Distributed Discovery and Orchestration:** We show how our VSA encoding scheme can be used for distributed discovery and orchestration of complex workflows. Workflow vectors are multicast to the network and participating services compute their own compatibility and offer themselves up for participation in the workflow.

We show that the result provides several desirable features and byproducts: it can encode workflows /sub-workflows that can be unbound on-the-fly and executed in a completely decentralized way; associated metadata can also be embedded

into the vector, e.g., security, configuration, etc.; the vector representation is extremely compact and self-contained and can be passed around using standard group transport protocols; and semantic comparisons or searches are scoped within a sub-group of services in a workflow, allowing scoped service matchmaking. We, then, apply the implementation to several Pegasus [16] workflows (Montage, CyberShake, Epigenomics, Inspiral Analysis, and SIPHT) and analyze the output.

The rest of the paper is structured as follows. In the next section, we provide some background into VSAs. In Section III, we outline the contributions we have made to scale VSAs and the extensions we applied for Workflows. In Section IV, we show how the VSA approach is applied to Linear Workflows and then to more complex DAG workflows. We discuss the resulting architecture and implementation in Section V, apply it to several Pegasus workflows in Section VI, and in Section VII, we conclude.

II. Vector Symbolic Architecture Overview

VSAs [17] are distributed representations that can be considered to sit somewhere between pure connectionist approaches and classic symbolic approaches, in that they employ *'atomic'* symbols to build complex representations of objects, like the classical symbolic approach to cognitive modeling and artificial intelligence research. However, the *atomic* symbols employed are random patterns of values spread over a hyper-dimensional vector space of dimension **N**. VSA symbols represent data and object features using random symbolic component vectors and combine these into vectors that semantically represent the component vectors and their relationships. VSA vectors are, therefore, said to be semantically self-describing [18]. *Atomic* vectors can be real valued like Plate's Holographic Reduced Representation (HRR) [19], typically having dimension $512 \leq N < 2048$, or binary vectors, such as Pentti Kanerva's Binary Spatter Codes (BSC) [20], typically having $N \geq 10,000$. For the work described here, we chose to build off Kanerva's BSCs, but most of the equations and operations listed and discussed should also be compatible with HRRs [21], too.

In BSCs, atomic symbols are assigned a random vector to represent an entity and due to the very high dimensionality employed, such *atomic* vector symbols are uncorrelated to each other with a high probability [20]. Higher level/complex concepts are built by combining *atomic* vectors using a *bundling* or *superposition* operation. A key advantage of this approach is that *superposition* involves no computationally expensive iterative learning of weights like traditional connectionist approaches [22]; rather, learning is achieved by direct combination of sub-features. An additional advantage is that VSA methods are completely deterministic and hence analysable and explainable, unlike traditional connectionist methods.

Superposition is the combination of sub-feature vectors into a same sized compound vector such that each vector element participates in the representation of many entities, and each entity is represented collectively by many elements [21]. Normalised Hamming Distance (HD) can be used to probe such a vector for its sub-features without unpacking or decoding the sub-features. If two high level concept vectors contain a number of similar sub-features, such vectors are said to be *semantically* similar, e.g., if we have three services:

$$Service1 = AudioIN_v + DeNoise_v + Convolution_v + Classify_v$$
$$Service2 = AudioIN_v + DeNoise_v + DFT_v + Classify_v$$
$$Service3 = AudioIN_v + LowPass_v + PowerSpec_v + Classify_v$$

where '+' is the superposition or bundling operator; then, comparing *Service1* with *Service2* will give a match since they have 3 common sub-features. Also, *Service1* and *Service2* will be more similar to each other than they are to *Service3*.

An issue arises, however, when using superposition to compare compound vectors in this way because such compound vectors behave as an unordered *bag* of features. Thus, if:

$$Service4 = AudioIN_v + DeNoise_v + Classify_v + ShutDownLine_v$$

then *Service4* would be equally similar to *Service1* as *Service2*, despite having a different output step. In order to resolve such issues VSAs employ a *binding* operator that allows feature values such as *DeNoise* and *Classify* to be associated with a particular *field name*, or *role*. This is analogous to how variable names are used in programming languages to associate values with a particular property, e.g., speed=3.

Services must agree upon a common method to assign atomic vectors for *roles* whereas feature vectors are usually compound vectors built up from lower level compound vectors and/or *atomic* vectors. When a *role* is bound to a value this results in a *role-filler* pair. Feature values such as *DeNoise* can be detected or extracted from the *role-filler* using an inverse binding operator. Bitwise XOR is used as both binding and unbinding with BSCs because it is its own inverse. In addition it is commutative and distributive over superposition ([20], page 147). It is also lossless, which means that both *roles* and *fillers* can be retrieved from a *role-filler* pair without any loss, e.g., using $'.'$ as the bitwise XOR operator; if $\boldsymbol{Z = X.A}$ then $\boldsymbol{X.Z = X.(X.A) = X.X.A = A}$, since $\boldsymbol{X.X = 0}$, the zero vector. Similarly $\boldsymbol{A.Z = X}$. Due to the distributive property, the same method can be used to test for sub-feature vectors embedded in a compound vector as follows:

$$\boldsymbol{Z = X.A + Y.B} \tag{1}$$
$$\boldsymbol{X.Z = X.(X.A + Y.B) = X.X.A + X.Y.B} \tag{2}$$
$$\boldsymbol{X.Z = A + X.Y.B} \tag{3}$$

Examination of (3) reveals that vector **'A'** has been exposed, thus, if we perform $\boldsymbol{HD(X.Z, A)}$ we will get a match. The second term $\boldsymbol{'X.Y.B'}$ is considered noise because $\boldsymbol{'X.Y.B'}$ is not in our known 'vocabulary' of features/symbols. When a role and value (*filler*) are bound together this is equivalent to preforming a mapping or *permutation* of a vector value's elements within the hyper-dimensional space, so that the new vector produced is uncorrelated to both the role and filler vectors. For example, if $\boldsymbol{V = R.A}$ and $\boldsymbol{W = R.B}$ then $\boldsymbol{R}$, $\boldsymbol{A}$ and $\boldsymbol{B}$ will have no similarity to $\boldsymbol{V}$ or $\boldsymbol{W}$. However, comparing $\boldsymbol{V}$ with $\boldsymbol{W}$ will produce the same match value to comparing $\boldsymbol{A}$ with $\boldsymbol{B}$. In other words, if $\boldsymbol{A}$ is closely similar to $\boldsymbol{B}$ then $\boldsymbol{V}$ will be closely similar to $\boldsymbol{W}$ because *binding* preserves distance within the hyper-space ([20, page 147]).

Thus, *binding* with *atomic* role vectors can be used as a method of *hiding* and *separating* values within a compound vector while maintaining the comparability between vectors.

This important property can be used to encode position and temporal information about sub-feature vectors within a compound vector. It also explains why we can state that $'X.Y.B'$ from (3) above, will not match to any known symbol, however, note that we can get back to B from $'X.Y.B'$; simply perform the appropriate xor's, $B = ((X.Y.B).X).Y$. We can now rephrase our *Service* description to differentiate its sub-features, i.e., we can reformulate $Service1$ to:

$$Service1 = Input_{rv}.AudioIN_v + Cleanup_{rv}.DeNoise_v + Process_{rv}.Convolution_v + Output_{rv}.Classify_v$$

This clearly resolves the incorrect matching between $Service1$ and $Service2$ with $Service4$. To test if $Service1$ uses $DeNoise_v$ as its cleanup step we perform:

$$HD(xor(Cleanup_{rv}, Service1), DeNoise_v) \quad (4)$$

When using 10kbit vectors, if the result of (4) is less than 0.47 then the probability of $DeNoise_v$ being detected in error is less than 1 in 10^9 ([20, page 143]). If we have an audio signal we want to classify, we might multicast a request vector $Z = Input_{rv}.AudioIN_v + Output_{rv}.Classify_v$ which would cause listening services such as services 1, 2 and 3 to respond or become activated. We could further query the responding services to determine what type of cleanup and processing they do as per (4).

III. Extensions to VSAs for Workflows

The number of detectable sub-features that can be superimposed into a single vector is of limited capacity, 89 vectors for BSCs of dimension 10k [23], and this issue must be addressed in order to encode large workflows. Chunking is a bundling method that combines groups of vectors into a single compound vector which is then used as base for further bundling operations, recursively producing a hierarchical tree structure where each node in the tree is a compound vector, as shown in Figure 1. Various methods of recursive *chunking* have been described [19]–[21], [23]. However, such methods suffer from limitations when employed for multilevel recursion - some lose their semantic matching ability if any single term differs, others can not maintain separation of sub-features for higher level compound vectors when lower level chunks contain the same vectors. We addressed these issues and describe a novel recursive encoding scheme that provides semantic matching at each level by combining two different methods of permuting vectors.

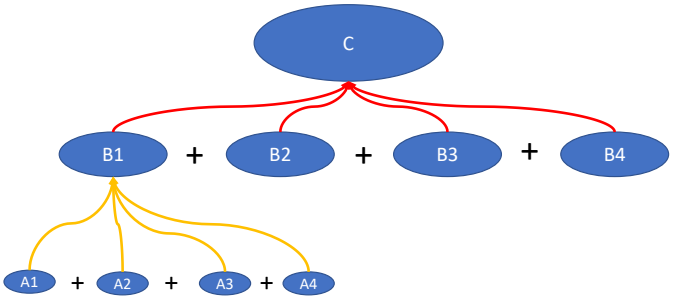

Figure 1. Workflow Chunk Tree, chunking proceeds from the bottom up.

In our scheme, the terminal nodes are worker services, the higher level nodes are concepts used to apply grouping to parts of workflow. The higher level nodes (known as *'clean-up memory'* [19], [20], [23]) are still services but they simply provide a proxy to the services to be *unbound* and executed, and thus are typically co-located with the first service of the sub-sequence, i.e., there is no network overhead. In a centralised system, *Clean-up memory* is typically implemented as an autoassociative memory. For our distributed workflow system, *clean-up memory* is implemented by the services themselves matching and resolving to their own vector representation.

Recchia et al., [24] point out that, for large random vectors, any mapping that permutes the elements can be used as a binding operator including cyclic-shift. The encoding scheme shown in (5) employs both XOR and cyclic-shift binding to enable recursive bindings capable of encoding many thousands of sub-features even when there are repetitions and similarities between sub-features:

$$Z_x = \sum_{i=1}^{x} Z_i^i . \prod_{j=0}^{i-1} p_j^0 + StopVec. \prod_{j=0}^{i} p_j^0 \quad (5)$$

Omitting $StopVec$ for readability, this expands to,

$$Z_x = p_0^0.Z_1^1 + p_0^0.p_1^0.Z_2^2 + p_0^0.p_1^0.p_2^0.Z_3^3 + ... \quad (6)$$

where

- '$.$' $= XOR$ and '$+$' $= BitwiseMajority\_Vote$.
- The exponentiation operator is redefined to mean cyclic-shift, $+ve$ exponents mean $Cshift\_right$, $-ve$ exponents mean $Cshift\_left$.
- Z_x is the next highest semantic *chunk* item containing a *superposition* of x sub-feature vectors. Z_x chunks can be combined using (5) into higher level chunks, e.g., Z_x might be $B1$, the superposition of $A1, A2, A3, ...$
- $Z_1, Z_2, Z_3, ...$ are sub-features being combined for the individual nodes in Figure 1. Each 'Z' is itself a compound vector representing a sub-workflow or a compound vector description for an individual service step.
- $p_0, p_1, p_2, ...$ are a set of known *atomic* role vectors used to define the current position/step in the workflow. The reason multiple 'p' vectors are XOR'ed together to define a single position within the workflow is to provide an iterative method for ordered activation of workflow steps during workflow execution, see (9).
- x is the, definable, *'chunk_size'*.
- $StopVec$ is a role vector that indicates to Z_x that all sub-feature/workflow steps have been executed.

A. Workflow execution

During workflow execution of a chunk tree similar to Figure 1 and encoded using (5), control first passes down the chunk tree, i.e., from $C \rightarrow B1 \rightarrow A1$, before traversing horizontally, $A1 \rightarrow A2 \rightarrow A3 \rightarrow A4 \rightarrow B1\_StopVec$. At this point $B1$ *'sees'* its own $StopVec$ and employs (9) to traverse horizontally at the next higher semantic level; see also flow arrows in Figure 2.

Referring to (6), an initiator or requester prepares the workflow, Z_x, for instantiation onto the network by performing an *unbind* operation, using (9), thus exposing the first workflow step, Z_1, as shown:

$$Z_1' = (T + p_0^0.Z_x)^{-1} \quad (7)$$

$$Z_1' = p_0^{-1}.T^{-1} + Z_1^0 + p_1^{-1}.Z_2^1 + p_1^{-1}.p_2^{-1}.Z_3^2 + ... \quad (8)$$

The T vector is a known *atomic* role vector used by Z_x's children to calculate their position within the sub-workflow. It is only bundled onto the workflow vector when an initiator or higher level node is requesting execution of its own sub-workflow, i.e., when traveling down the workflow.

In (8), note that all other Z vectors remain hidden because they are still permuted. Thus, listening services can only match to Z_1. As control passes, horizontally, from $Z_1 \rightarrow Z_2 \rightarrow Z_3$ ... each active service uses the current permutation of the T vector to calculate its zero based position ***'n'*** within the currently active parent chunk vector. It can then activate the next workflow step in the chunk by repeating the *unbind* operation, generalized as:

$$Z'_{n+1} = (p_n^{-n}.Z'_n)^{-1} \tag{9}$$

Hence, $Z'_2 = (p_1^{-1}.Z'_1)^{-1} = p_1^{-1}.p_0^{-2}.T^{-2} + p_1^{-1}.Z_1^{-1} + Z_2^0 + p_2^{-2}.Z_3^1 + \ldots$ Thus, execution proceeds in a completely decentralized manner whereby each node is activated when its preceding node, or parent, *unbinds* the currently active chunk vector and multicasts it to the network.

B. Local Arbitration

A major advantage of the VSA approach is the ability to find semantic matches because each service can extend beyond simple matches to include measures of real-time compute utility as well as policy. For certain scenarios, such as military coalition environments, there is a need to ensure multiple copies of services are distributed throughout the communications network. In order to decide which service is invoked, we employ a process of *local arbitration*, which is achieved as follows. Using terminology from (7) and (9), if the currently active service is Z'_n, then before transmitting the next service request, it enters *match collecting mode* in order to arbitrate matches from all nodes that reply within a tunable window of time. After the interval expires, the highest ranking responder is selected and a *continue* message is broadcast by Z'_n identifying the winner. Since all communications are multicast, all services see all messages, and consequently the winning service continues and losing services discontinue. To reduce communication overhead further matching services delay their response by an interval inversely proportional to their match value. Thus, better matches respond quicker. If a service *sees* a higher match value before it has responded then it terminates without sending a reply.

C. Pre-provisoning and Learning to get ready

From (9) we see that each workflow step is exposed by iterative application of *'p'* vector permutations. Non-matching services can use this method to *peek* a vector enabling anticipatory behavior such as the pre-provisioning of a large data-set or changing a device's physical position, e.g., drones. Obviously, services can *peek* multiple steps into the future and could *learn* how early to start pre-provisioning. This ability to anticipate could be used to perform more complex, online, utility optimisation learning. For example, a drone, by monitoring multiple workflows may be able to understand that it will be needed in 10 minutes to perform a low priority task and in 15 minutes for a high priority task. Under these circumstances it may choose not to accept the low priority task.

IV. VSA Representation of Complex Workflows

As a test case and to compare to alternative approaches (e.g., [14]), we modeled each word of Shakespeare's play Hamlet as a service and applied hierarchical chunking to abstract into stanzas, scenes, and acts (see Figure 2). This approach tests the capability of the chunking scheme to encode serial and chunked workflows where the services at the lowest level are the 4620 unique words of the play. The semantic level above are the individual stanzas spoken by each character; the level above this are individual scenes of the play (e.g., A1S1, A1S2); next are the five acts, A1 to A5, and finally a single 10kbit vector semantically represents the whole play. The individual word services are distributed in a communications network and by multicasting the top level vector the whole play is performed in a distributed manner with 29770 word services being invoked in the correct order.

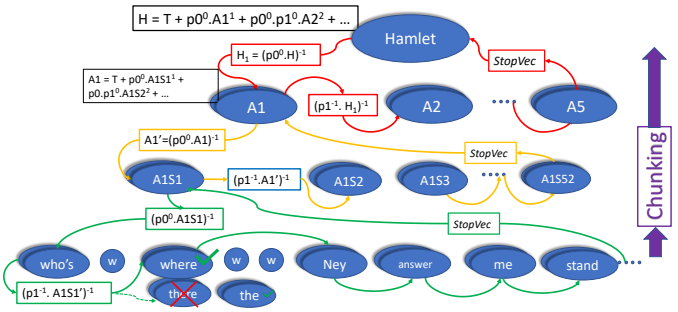

Figure 2. Hamlet as a serial workflow.

We employ a vector alphabet, a unique vector per character, and (5) to build a semantic vector description of each word-service in the test case. The idea is that each letter making up a word represents some feature of a service description, i.e., analogous to the different input/output/name/descriptions parts of a real world service. Thus, variable lengths of words and similarity of spellings represent a mix of different services of different complexity and functional compatibility. We can use this feature to find semantically similar service compositions when the best match composition is not available, i.e., we can find alternative words or stanzas, as shown in Figure 2, where the word *where* is selected as an alternate to *there*.

We note that this workflow is a linear sequence of services, next we show how such an approach can be extended to DAG workflows by employing three phases:

1) A recruitment phase where services are discovered, selected and uniquely named.
2) A connection phase where the selected services connect themselves together using the newly generated names.
3) An atomic *start* command indicates to the connected services that the workflow is fully composed and can be started.

Thus, in mathematical terms, using (6):

$$\begin{aligned} WP &= p0^0.(Recruit_{Nodes})^1 + \\ &\quad p0^0.p1^0.(Connect_{Nodes})^2 + p0^0.p1^0.p2^0.Start^3 \\ Recruit_{Nodes} &= p0^0.Z_1^1 + p0^0.p1^0.Z_1^2 + ...p0^0.p1^0.p2^0.p3^0.Z_1^4 \\ &\quad + p0^0.p1^0.p2^0..p4^0.Z_2^5 ... + p0^0.p1^0.p2^0..p9^0.Z_2^{10} \\ &\quad + p0^0.p1^0.p2^0..p10^0.Z_3^{11} + p0^0.p1^0.p2^0..p11^0.Z_4^{12} \\ &\quad + p0^0.p1^0.p2^0..p12^0.Z_5^{13} + p0^0.p1^0.p2^0..p15^0.Z_5^{16} \\ &\quad + p0^0.p1^0.p2^0..p16^0.Z_6^{17} + p0^0.p1^0.p2^0..p17^0.Z_7^{18} \\ &\quad + p0^0.p1^0.p2^0..p18^0.Z_8^{19} + p0^0.p1^0.p2^0..p19^0.Z_9^{20} \end{aligned}$$

$$\boldsymbol{Connect}_{Nodes} = (\boldsymbol{p0^0}.\mathbb{P}_1^1 + \boldsymbol{p0^0}.\boldsymbol{p}_1^0.\mathbb{C}_1^2) + (\boldsymbol{p0^0}.\boldsymbol{p1^0}.\boldsymbol{p2^0}.\mathbb{P}_2^3 + \boldsymbol{p0^0}.\boldsymbol{p1^0}.\boldsymbol{p2^0}.\boldsymbol{p3^0}.\mathbb{C}_2^4) + ...$$

where each $\boldsymbol{Z_n}$ in $\boldsymbol{Recruit}_{Nodes}$ is the semantic/compound vector representation of each service, built from the <job> entries found in the DAX. A generic description of each service was built from the service name and its description and used to build the workflow request vector. For individual instances of a service, e.g., *mDiffFit*, we additionally encode the instance's parameters and resource names to create similar but distinct service instances to, again, show that service discovery can be achieved when descriptions are not identical.

The resulting workflow, $\boldsymbol{WP}$, is a superposition representing the linear sequence of steps needed to discover, connect and initiate the workflow. Hence, execution of the workflow proceeds in a similar manner to that described in Section III-A, but with some additional workflow specific processing carried out by each selected node. The top-level vector, *WP* is prepared as per (7)

$$\boldsymbol{WP}_1 = (\boldsymbol{T} + \boldsymbol{p}_0^0.\boldsymbol{WP})^{-1} = \boldsymbol{Recruit}_{nodes} + \boldsymbol{noise}$$

When multicast, this exposes and activates the $\boldsymbol{Recruit}_{nodes}$ service which, operating as a cleanup service, carries out the the same operation to initiate the recruitment phase.

$$\boldsymbol{Recruit}'_{nodes} = (\boldsymbol{T} + \boldsymbol{p}_0^0.\boldsymbol{Recruit}_{nodes})^{-1}$$
$$\boldsymbol{R}'_1 = \boldsymbol{p}_0^{-1}.\boldsymbol{T}^{-1} + \boldsymbol{Z}_1^0 + \boldsymbol{p}_1^{-1}.\boldsymbol{Z}_1^1 + \boldsymbol{p}_1^{-1}.\boldsymbol{p}_2^{-1}.\boldsymbol{Z}_1^2 + ...$$

where $\boldsymbol{Z_1}$ is a request for the first node in the DAG, an mProjectPP in the Montage DAG. This will be matched by all listening mProjectPPs. Acting as the local arbitrator, see Section III-B, the $\boldsymbol{Recruit}_{nodes}$ service multicasts its preferred match from the replies received. The newly discovered and activated service uses the current permutation of the $\boldsymbol{T}$ vector to calculate its position ($\boldsymbol{NODE}_{id}^n$) in the $\boldsymbol{Recruit}_{nodes}$ phase from which it can calculate its unique parent and child vector names to be used during the $\boldsymbol{Connect}_{Nodes}$ phase. Thus, the first mProjectPP, having position *p0* and being a $\boldsymbol{Z_1}$, calculates its parent and child names as,

$\boldsymbol{\mathcal{P}_0} = \boldsymbol{Z}_1^0 \,.\, (\boldsymbol{NODE}_{id}^0 \,.\, \boldsymbol{ROLE\_parent})$

$\boldsymbol{\mathcal{C}_0} = \boldsymbol{Z}_1^0 \,.\, (\boldsymbol{NODE}_{id}^0 \,.\, \boldsymbol{ROLE\_child})$

It then enters *Listening for Connections Mode* while, as the new *local arbitrator*, it also multicasts the next recruitment request by performing an unbind on its received vector $\boldsymbol{R}'_1$, thus $\boldsymbol{R}'_2 = (\boldsymbol{p}_1^{-1}.\boldsymbol{Z}'_1)^{-1} = \boldsymbol{p}_1^{-1}.\boldsymbol{p}_0^{-2}.\boldsymbol{T}^{-2} + \boldsymbol{p}_1^{-1}.\boldsymbol{Z}_1^{-1} + \boldsymbol{Z}_1^0 + \boldsymbol{p}_2^{-2}.\boldsymbol{Z}_1^1 + ...$ which will cause another mProjectPP to be selected and this decentralized process repeats until the last service to be recruited, the $\boldsymbol{Z_9}$, mjPeg, service unbinds and transmits the next vector, the $\boldsymbol{Recruit}_{nodes}$ *StopVec*. The $\boldsymbol{Recruit}_{nodes}$ cleanup service detects its stop vector, causing it to perform an unbind and multicast of $\boldsymbol{WP'}$ therby activating the $\boldsymbol{Connect}_{nodes}$ phase:

$$\boldsymbol{WP}_2 = (\boldsymbol{T} + \boldsymbol{p}_1^{-1}.\boldsymbol{WP}_1)^{-1} = \boldsymbol{Connect}_{nodes} + \boldsymbol{noise}$$

At this point, all recruited services are listening for connection request on their unique parent and child vectors. The activated $\boldsymbol{Connect}_{nodes}$ service, acting as a cleanup service, uses (7) to initiate and activate the first *parent* node of the $\boldsymbol{Connect}_{nodes}$ phase.

$$\boldsymbol{Connect}'_{nodes} = (\boldsymbol{T} + \boldsymbol{p}_0^0.\boldsymbol{Connect}_{nodes})^{-1}$$
$$\mathbb{P}'_1 = \boldsymbol{p}_0^{-1}.\boldsymbol{T}^{-1} + \mathbb{P}_1^0 + \boldsymbol{p}_1^{-1}.\mathbb{C}_1^1 + \boldsymbol{p}_1^{-1}.\boldsymbol{p}_2^{-1}.\mathbb{P}_2^2 + ...$$

When a service matches to its *parent* vector it simply performs the next unbind/multicast since in doing so it will activate its associated *child* service, automatically informing the child service of the location of its resources/output/ip-address.

When a service receives a multicast that matches to its *child* vector it can lookup the sender/parent's ip-address and send a unicast '*hello*' message to the parent, thus establishing the required connection before activating the next *parent* by performing a further unbind/multicast of the $\boldsymbol{Connect}_{nodes}$ vector. This process repeats until the final child request is processed causing the $\boldsymbol{Connect}_{nodes}$ service to detect its *StopVec* which, in turn, causes it to unbind and multicast the *StartVec* indicating to all nodes that the workflow has been fully constructed and processing can be started.

V. Implementation

Our VSA platform is implemented in `Python2` and has a modular architecture with several components that are capable of being reused as plugins to other systems.

The Workflow Importer component imports a Pegasus workflow description (DAX) file. This is an, XML format, multi-nested dictionary description of a workflow which details each service node and its input output resources. The Workflow Importer reads the DAX file into a python dictionary. It then parses the dictionary and extracts the *job* entries to create a list of vectors that represent each service node in the DAX, the *NodeVectors* list. Similarly, it traverses the *child* section of the DAX producing the *EdgeVectors* list, a paired list of vectors representing the parent (output) and child (input) connections of the workflow. The Workflow Importer passes *NodeVectors* and *EdgeVectors* to the VSA Creator.

The VSA Creator is used to bind the lists of vectors into a single vector, a reduced representation, of the workflow using chunking, see Section III. Chunking is performed bottom up so that higher level vectors are produced as needed. These are recursively rebound until the vector list is reduced to a single vector value. The *NodeVectors* list and the *EdgeVectors* list are combined separately producing two high level vectors, the $\boldsymbol{Recruit}_{Nodes}$ vector and the $\boldsymbol{Connect}_{Nodes}$ vector. The VSA Creator then binds these two vectors together with the *Start* vector into a single vector representing the entire workflow, the *WorkFlow* vector. This *WorkFlow* vector and all its associated sub-vectors are encapsulated in a *chunk tree* object as per Figure 1, which is then passed to the VSA executor.

The VSA Executor *flattens* the workflow by distributing copies of all non-terminal chunk vectors into the terminal (bottom level/worker) nodes. Non-terminal nodes are distributed to the first child of a parent node to decode the first vector in a higher level vector. For robustness, the VSA Executor can be made to distribute more than one copy of the cleanup objects into other terminal node objects.

The Logging Component collects metrics as the workflow runs to feed into external processors. Logging currently collects a trace of the nodes and edges that are being processed by the workflow.

The Visualisation Component takes the log output and generates a DAG layout graph using Graphviz [25].

VI. Comparative Evaluation

For the evaluation, we imported five different DAX workflows generated using the Pegasus workflow generator [16]:

1) Montage (NASA/IPAC) stitches multiple input images together to create custom mosaics of the sky.
2) CyberShake (Southern Calfornia Earthquake Center) characterizes earthquake hazards in a region.
3) Epigenomics (USC Epigenome Center and Pegasus) automates various operations in genome sequence processing.
4) Inspiral Analysis (LIGO) generates and analyzes gravitational waveforms from data collected during the coalescing binary systems.
5) SIPHT (Harvard) automates the search for untranslated RNAs (sRNAs) for bacterial replicons in the NCBI database.

We ran the experiment on a MacBookPro11-4; Intel Core i7, 2.8 GHz; 4 cores; 16 GB memory using the CORE/EMANE network simulator using the Python toolkit discussed in the previous section in order to verify workflows could be loaded, encoded and then executed using the VSA format. We then recreate the workflows the VSAs have encoded to verify against the original. This proceeds as follows. Once imported, the DAX workflows are processed using the VSA creator to build the semantic vector workflow encoding, and apply the recruitment and connectivity phases to create service instances of the workflow jobs and interconnections. During the execution of the workflow using our simulator, we extract the metrics described in the previous section, which essentially contain a log of the run in the order of execution. This results in a set of nodes and edges being generated which we graph using Graphviz.

Figure 3 shows the resulting comparisons of the five workflows. The coloured images represent the Pegasus generated workflows and blue workflows show the VSA generated reconstruction of the workflows. Aside from the cosmetic difference, this demonstrates that all workflows were composed and correctly connected accurately in all cases.

VII. Conclusions

In this paper, we applied and demonstrated the viability of using VSA approach to encode workflows containing multiple coordinated sub-workflows in a way that allows the workflow logic to be unbound on-the-fly and executed in a completely decentralized manner. The Hamlet test-case demonstrated that we can use VSA service discovery to select alternate services on-the-fly. We anticipate that such an approach will lend itself well to edge networks where the transient nature of the mobile nodes will require such dynamic and decentralized control. In addition this test-case demonstrates that our encoding scheme is scalable, i.e., 30k individual services steps where successfully encoded and decoded in the correct order. The Pegasus test-case demonstrates the potential for encoding more complex multi-pathway workflows by encoding and decoding a number of Pegasus DAGs.

The local arbitration mechanism employed to choose the best matched services not only demonstrates a method that enables workflows to be orchestrated without a central point of control but can also be used to perform utility optimsation.

Using VSAs to enable services to become self-describing has a distinct advantage because of superposition. Conventional approaches could use multicast to transmit a bag of features across the network but each individual component feature within the bag would have to be examined and compared separately for a service to assess its compatibility to the request. In addition VSAs are robust to noise, they support mathematical inference and analogical mapping operations [19], [20] which could be used to learn similarities between vector symbols across coalitions and infer new workflows from previously seen workflows. For these reasons, our future work will therefore focus on such self-describing service compositions in order to realize the vision set out in [13].

Acknowledgements

This research was sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence under Agreement Number W911NF-16-3-0001. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Research Laboratory, the U.S. Government, the U.K. Ministry of Defence or the U.K. Government. The U.S. and U.K. Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

References

[1] M. Wieczorek, R. Prodan, and T. Fahringer, "Scheduling of scientific workflows in the askalon grid environment." SIGMOD Record, vol. 34, no. 3, 2005, pp. 56–62.

[2] T. Fahringer et al., Workflows for e-Science. Springer, New York, 2007, ch. ASKALON: A Development and Grid Computing Environment for Scientific Workflows, pp. 143–166.

[3] Altintas et al., "Kepler: An Extensible System for Design and Execution of Scientific Workflows," in 16th International Conference on Scientific and Statistical Database Management (SSDBM). IEEE Computer Society, New York, 2004, pp. 423–424.

[4] P. Kacsuk, "P-grade portal family for grid infrastructures," Concurr. Comput. : Pract. Exper., vol. 23, March 2011, pp. 235–245.

[5] Deelman et al., "Pegasus: a Framework for Mapping Complex Scientific Workflows onto Distributed Systems," Scientific Programming Journal, vol. 13, no. 3, 2005, pp. 219–237.

[6] Oinn at al., "Taverna: A Tool for the Composition and Enactment of Bioinformatics Workflows," Bioinformatics, vol. 20, no. 17, November 2004, pp. 3045–3054.

[7] A. Harrison, I. Taylor, I. Wang, and M. Shields, "WS-RF Workflow in Triana," International Journal of High Performance Computing Applications, vol. 22, no. 3, Aug. 2008, pp. 268–283.

[8] Barga et al., "The trident scientific workflow workbench," in Proceedings of the 2008 Fourth IEEE International Conference on eScience. Washington, DC, USA: IEEE Computer Society, 2008, pp. 317–318.

[9] T. Glatard, J. Montagnat, D. Lingrand, and X. Pennec, "Flexible and efficient workflow deployment of data-intensive applications on grids with MOTEUR," Int. J. High Perform. Comput. Appl., vol. 22, August 2008, pp. 347–360.

[10] B. Balis, "Increasing scientific workflow programming productivity with hyperflow," in Proceedings of the 9th Workshop on Workflows in Support of Large-Scale Science, ser. WORKS '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 59–69.

[11] S. Basagni, M. Conti, S. Giordano, and I. Stojmenović, Mobile Ad Hoc Networking: Edited by Stefano Basagni et al. IEEE, 2004.

[12] T. Pham, G. Cirincione, A. Swami, G. Pearson, and C. Williams, "Distributed analytics and information science," in In IEEE International Conference on Information Fusion (Fusion),, 2015.

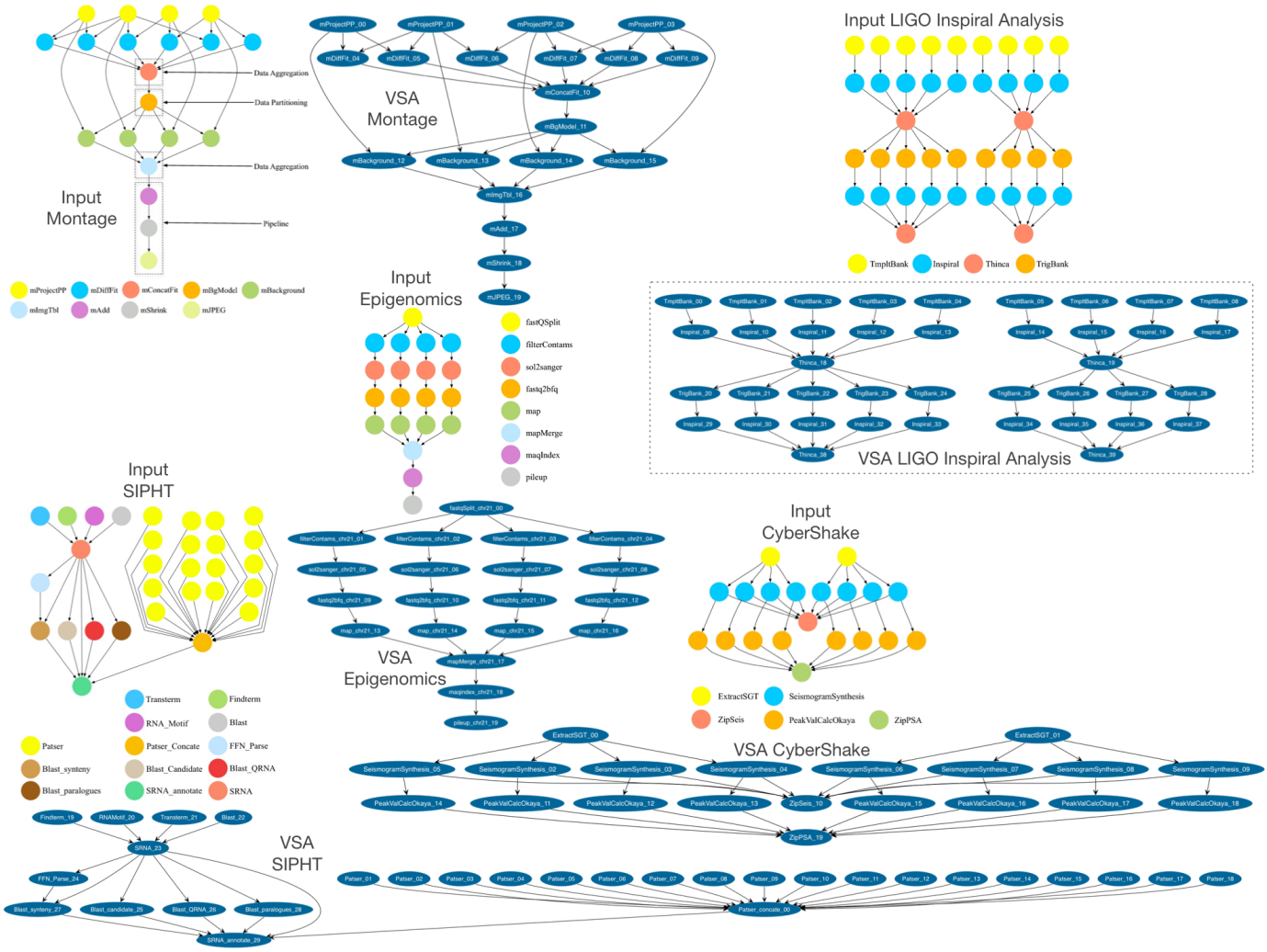

Figure 3. A comparison of five different DAX workflows as input and the VSA reconstructed workflows from post processing the semantic vector.

[13] D. Verma, G. Bent, and I. Taylor, "Towards a distributed federated brain architecture using cognitive iot devices," in 9th International Conference on Advanced Cognitive Technologies and Applications (COGNITIVE 17), 2017.

[14] J. P. Macker and I. Taylor, "Orchestration and analysis of decentralized workflows within heterogeneous networking infrastructures," Future Generation Computer Systems, 2017.

[15] E. Wittern, J. Laredo, M. Vukovic, V. Muthusamy, and A. Slominski, "A graph-based data model for api ecosystem insights," in Web Services (ICWS), 2014 IEEE International Conference on. IEEE, 2014, pp. 41–48.

[16] "Workflow Generator Pegasus," https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator, [accessed on 30/05/201/18].

[17] R. W. Gayler, "Vector symbolic architectures answer jackendoff's challenges for cognitive neuroscience," arXiv preprint cs/0412059, 2004.

[18] Eliasmith at al., "A large-scale model of the functioning brain," science, vol. 338, no. 6111, 2012, pp. 1202–1205.

[19] T. A. Plate, Distributed representations and nested compositional structure. University of Toronto, Department of Computer Science, 1994.

[20] P. Kanerva, "Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors." Cognitive Computation, vol. 1, no. 2, 2009, pp. 139–159.

[21] T. A. Plate, Holographic Reduced Representation: Distributed Representation for Cognitive Structures. Stanford, CA, USA: CSLI Publications, 2003.

[22] J. L. McClelland, "Connectionist models james l. mcclelland & axel cleeremans in: T. byrne, a. cleeremans, & p. wilken (eds.), oxford companion to consciousness. new york: Oxford university press, 2009."

[23] D. Kleyko, "Pattern recognition with vector symbolic architectures," Ph.D. dissertation, Luleå tekniska universitet, 2016.

[24] G. Recchia, M. Sahlgren, P. Kanerva, and M. N. Jones, "Encoding sequential information in semantic space models: comparing holographic reduced representation and random permutation," Computational intelligence and neuroscience, vol. 2015, 2015, p. 58.

[25] "Graphviz - Graph Visualization Software," http://www.graphviz.org, [accessed on 30/05/18].