PVMv2 Specification

Version 0.5.0

Thomas Bytheway Lucian Carata

January 30, 2019

1 Introduction

PVMv2 is a model designed for converting low level event traces into graphs representing data flow interactions.

It achieves this by allowing people with an in-depth understanding of event trace semantics to use a minimal DSL for defining a mapping between the two worlds. The mapping governs how different event types in the trace create new data flows, change existing ones or modify the resulting graph in a different (restricted) way. This mapping is then applied to actual trace events, resulting in the iterative construction of a data flow graph.

This document aims to describe the conceptual basis behind the PVMv2 model, explain how the resulting dataflow graph is structured and what assumptions can be made about it, and how one would go about creating a mapping to interpret new data under the guise of the PVMv2 model.

2 Abstract Types

The PVMv2 model defines a number of different abstract types which are used both to represent data in the resulting graph format, and as a basis for deriving concrete types to represent real trace entities. Figure 1 shows a UML representation of the different types and how they are related. Below we will go into more detail on each of the types in turn.

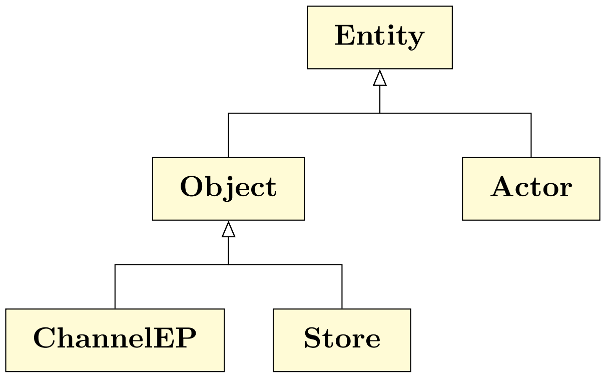

Figure 1: A diagram of the basic abstract types in PVMv2

Entity is the root abstract type. The common feature of all entities is that they possess a uniquely identifying name that is either totally unique for some subtypes, or is maintained for version chains for other subtypes. In general concrete types should not be derived from Entity unless there is no better option available.

Actor entities are active entities that perform actions, generally processing units such as UNIX processes, though they can also represent more abstract notions of processing. They are directly comparable to subjects in the orange book. They act upon objects, or other actors.

Objects are passive entities that are used by actors. Generally data carrying entities.

Store Object type that stores data internally. Such objects version on first write and last write. libPVM uses a Store subclass called EditSession to represent time intervals where a Store is concurrently modified by multiple users. If thread/process interleaving data within an EditSession is important, one will have to look at the events in the original trace and deduce things from there.

ChannelEP (end point) Object type that data flows through without internal storage. Does not version. May participate in connections.

3 Graph Schema

In this section we will describe how a well formed PVMv2 graph will appear. This does not necessarily inform directly how an encoding of such a graph may appear in a given medium, but should be taken as a strong suggestion of how it will be structured. In Appendix B we describe the encoding of such graphs in Neo4j and in Appendix C their encoding into CDM.

3.1 Nodes

All graph nodes possess a unique id that is not shared with any other nodes in the same graph.

3.1.1 Schema Nodes

Schema nodes describe concrete types that are defined by the mapping that created a database. Each node has a name of the type being defined, an abstract type that the concrete type is a class of, and a list of potential properties that it is valid for a node of that type to have.

3.1.2 Context Nodes

Context nodes describe the context within which other entities and relations have been created. Each has a `type` field that notes a schema node describing the properties that the context node may have, it may then have additional fields matching these named properties.

3.1.3 Entity Nodes

Entity nodes represent an Entity of an abstract type in the graph, they can be additionally subclassed into the abstract types as defined in section 2. All entity nodes have a `uuid` field defining the unique name for the entity, a `type` field naming the concrete type that the entity is a representative of, and a `ctx` field noting the context under which the entity was created.

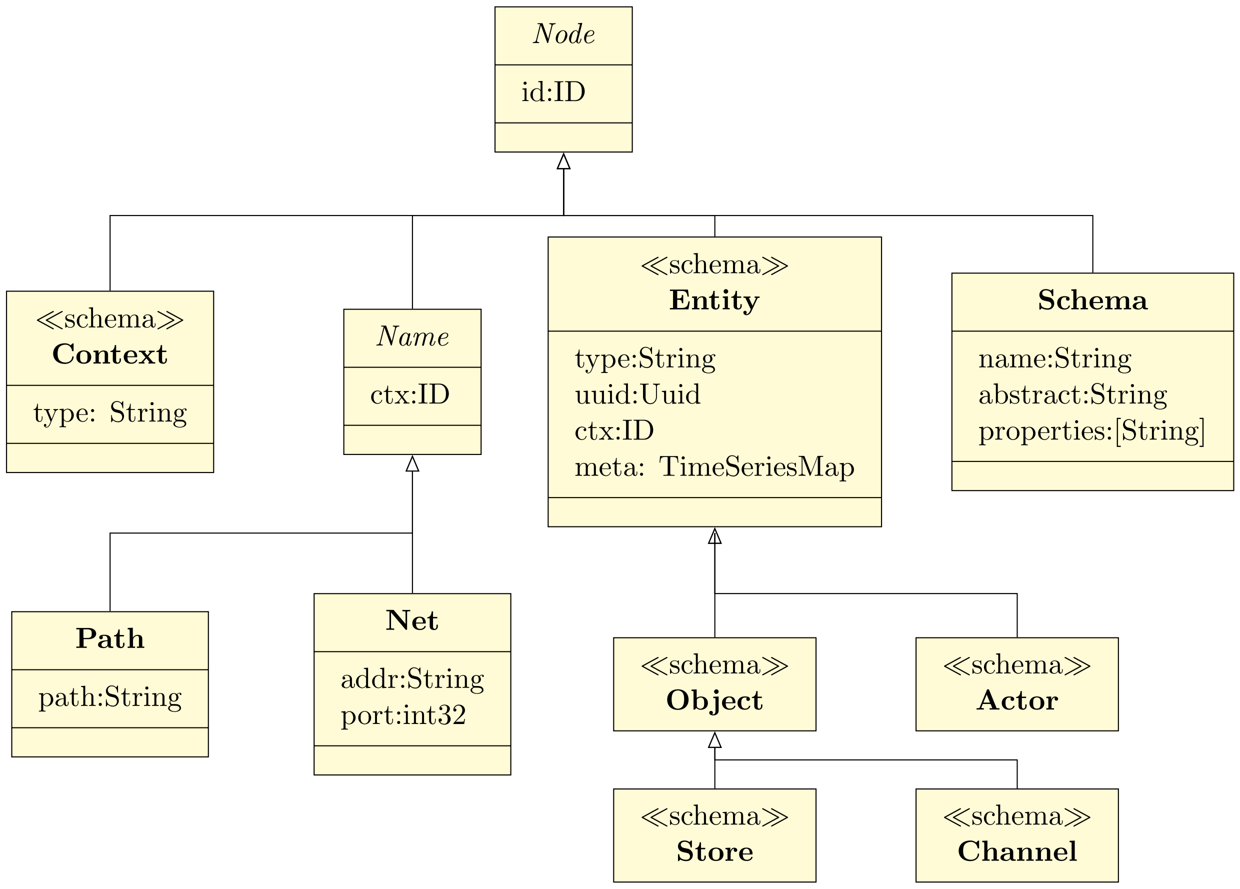

Figure 2: A diagram of the potential nodes in a PVMv2 database.

3.1.4 Non-canonical Names

Non-canonical names are represented in the graph as nodes. These are meant for names that do not uniquely identify nodes, but that may only be valid for limited periods of time. Path-names are a prime example of the type of name this is most usefully applied to. We currently support two types of non-canonical names, however this may be expanded in future versions of the spec.

Path Names

Path name nodes represent file paths in the pvm graph, the paths contained must be valid pathnames but beyond that there are no restrictions, both relative names and names including '.' and '..' are allowed. A path name node has a single `path` field which holds the path it refers to.

Network Addresses

Net address nodes store network addresses used to describe entities, they each have an `addr` field which contains the IP address or hostname portion of the network address and a `port` field which contains the port for the address.

3.2 Relations

All relations also possess a uniquely identifying id that will not repeat within the same graph.

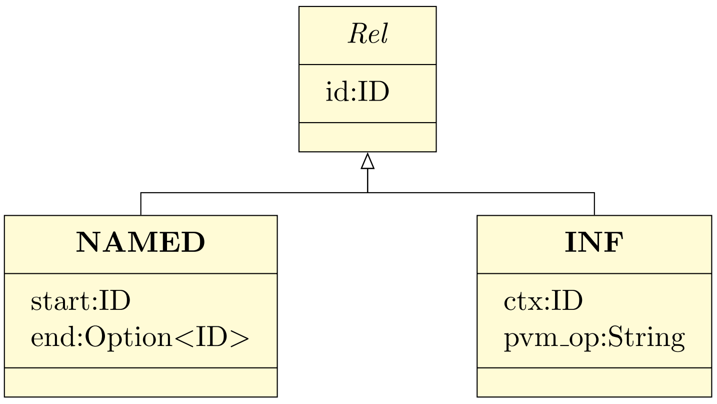

Figure 3: A diagram of the potential relations in a PVMv2 database.

INF

Indicates that the source has potentially influenced, directly (through data) or indirectly (through control flow) the destination. An INF relation has a `ctx` property that notes the context that it was created under. The relation also has a `pvm_op` field that describes which PVM verb was being processed when the edge was created, this can have one of the following values: Source, Sink, Connect, Version or Unknown. Edges with the Version op value are created when a verb causes a new node to come into existence which is notionally a new version of an existing node. It should be expected that two nodes on the ends of a version edge have the same UUID and represent a continuation of the same object. Unknown should not be used explicitly and is mostly provided as a safe default 'don't know' option for parsing. Inf relations are in general only permitted between nodes that are Entity nodes of their subclasses.

NAMED

Indicates that the source has been mentioned by the destination non-canonical name.

4 Mappings

A PVMv2 mapping describes how to process a certain type of raw event trace under the PVMv2 model. It comprises of a set of concrete type definitions, which describe the different types of entities that exist within the raw trace, and how those entities match to the PVMv2 abstract types. It also contains a set of mapping expressions utilizing the PVMv2 verbs to describe the semantic connotations of a given raw event, resulting in notionally a function that given a raw event applies a set of updates to the graph describing the current state in PVMv2 semantics.

4.1 Concrete Type Declaration

4.2 Verbs

`declare(ctype, uuid) -> e`
Forces the creation of an Entity `e` named by `uuid` if it does not already exist.

`sink(src:Actor, dst:Entity)`
Called when as part of a mapped function, an Actor `src` sinks data to an Entity `dst` as an atomic operation. Causes Stores to version, creating a new Store entity connected to the previous one by an INF relation. Creates an INF relation from Actor `src` to Entity `dst`.

`source(src:Entity, dst:Actor)`
Called when as part of a mapped function, an actor sources data from an entity. Creates an INF relation from entity to actor.

`connect(ep1:ChannelEP, ep2:ChannelEP, dir)`
Called as part of a mapped function to declare that two channel nodes are connected to each other and that data can flow between them. The dir argument indicates the direction of flow, either mono or bi. Mono indicates that data may only flow from the first channel to the second, bi indicates that data may flow in both directions.

`mention(e:Entity, n:Name)`
Indicates that as part of a mapped function an Entity `e` has been reffered to using the name `n`. This will generate a NAMED relation from `e` to `n` if one does not already exist.

`unlink(e:Entity, n:Name)`

`property(e:Entity, pname:str, pval:str)`

A A Worked Example

B Neo4j Serialisation

All database objects have a `db_id` property that encodes the database identifier for that object. These should be interpreted bitwise as unsigned 64 bit integers (rather than signed 64 bit integers as neo4j would normally present them).

All database nodes have the "Node" label, and are indexed by their `db_id` field on this label.

Schema nodes are labelled with "Node, Schema". They then have `name`, `abstract` and `properties` fields as described earlier.

Context nodes are labelled with "Node, Context". They have a `type` field which matches the `name` field of a Schema node noting what schema the Context node is following. The node then has additional fields matching the names in the matching schema node's `properties` field.

Name nodes are all labelled "Node, Name".

Path name nodes are labelled "Node, Name, Path". They then have a `path` field.

Net name nodes are labelled "Node, Name, Net". They then have an `addr` field containing the net address and a `port` field containing the port number.

Entity nodes are labelled "Node, Entity" plus additional labels for sub types such as "Actor" or "Store". They all have a `type` field noting thier concrete entity type, which should match a Schema node, they have a `uuid` field noting thier UUID, a `ctx` referencing the `db_id` value of a Context node, a `meta` field with a historic time series of updates made to property values and as a convenience feature they also have the most recent value for each property value stored as a field on the node.

INF edges have the label "INF". They have a `ctx` field that matches the `db_id` of a Context Node, a `pvm_op` field that notes what PVM operation caused their creation.

NAMED edges have the label "NAMED". They have a `start` field that matches the `db_id` of a Context node and potentially a `end` field that matches the `db_id` of a Context node.

C CDM Serialisation