PVMv2 Specification

Version 0.2.0

Thomas Bytheway Lucian Carata

October 4, 2018

1 Introduction

PVMv2 is a model designed to aid in the extraction of high level data flow interactions from low level event traces. It achieves this by using descriptive mapping programs written in a custom DSL by people who understand the semantic content of the event trace in question. This mapping can then be applied to actual trace events and results in the iterative production of a dataflow 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 to both 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.

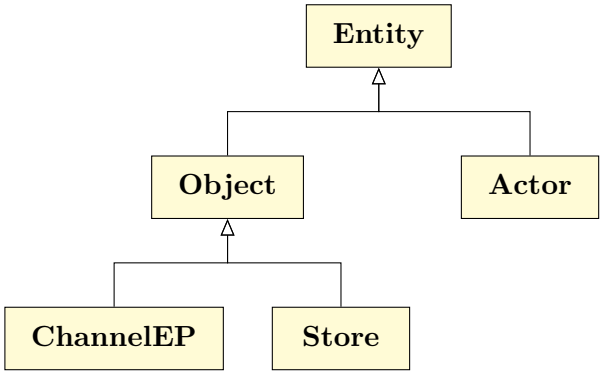

Figure 1: A diagram of the basic abstract types in PVMv2

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.

Object
Passive entities that are acted upon by actors. Generally data carrying entities.

Store
Object type that stores data internally. Versions from first write to last write, using EditSessions.

ChannelEP
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

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 `ty` 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.

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.

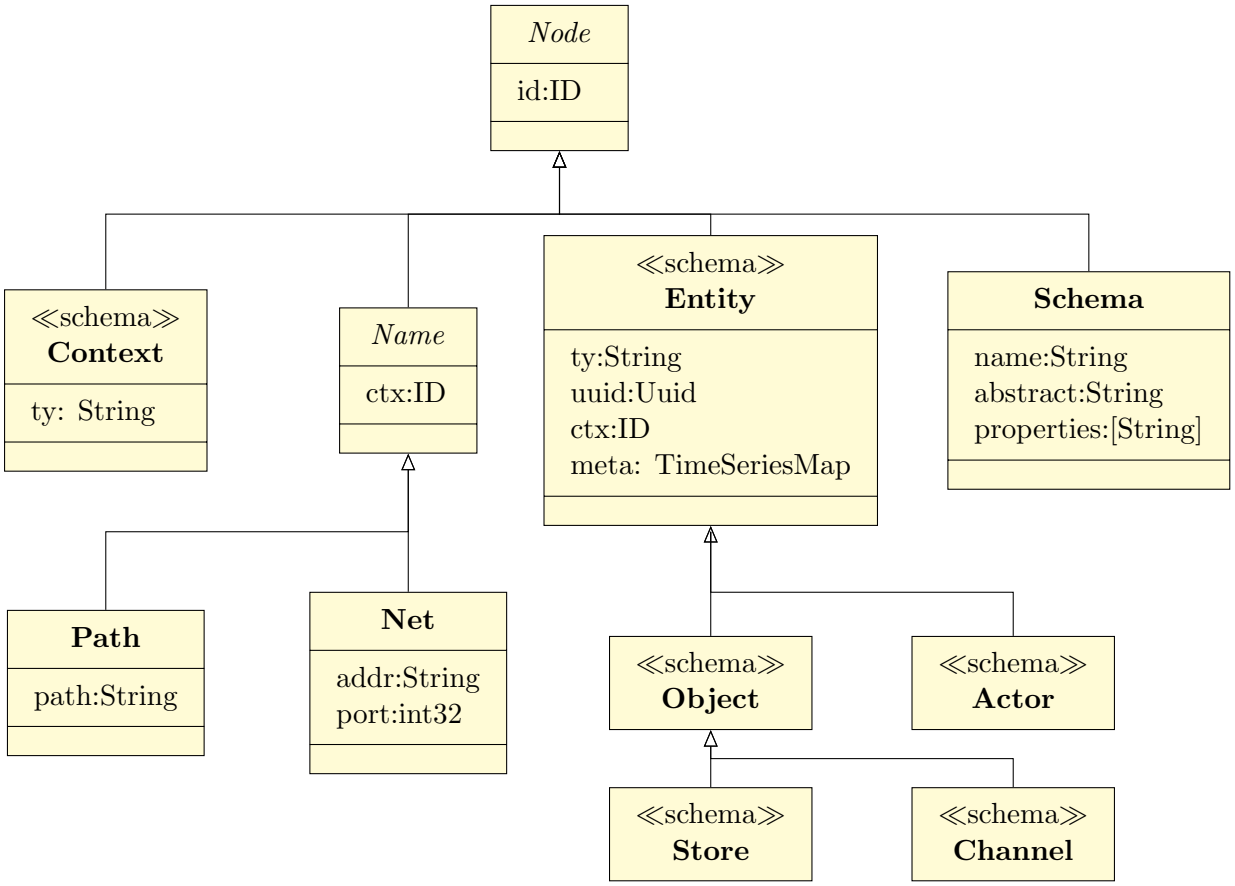

Figure 2: A diagram of the potential nodes in a PVMv2 database.

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.

INF

Indicates that the source has potentially imparted data or control to the destination. An INF relation has a `ctx` property that notes the context that it was created under. The

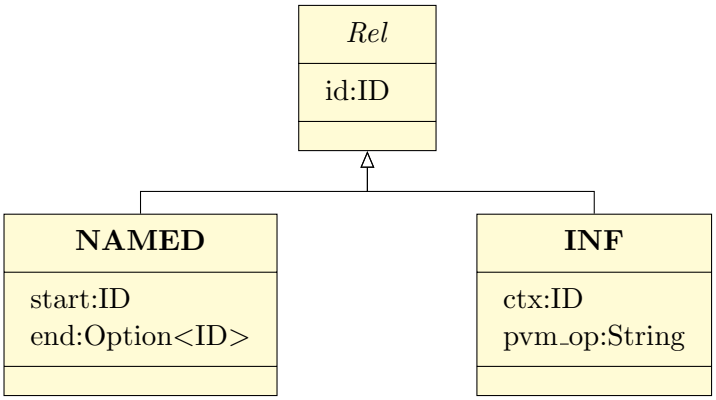

Figure 3: A diagram of the potential relations in a PVMv2 database.

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 entity if it does not already exist.

`sink(src, dst) -> ()`
Called when as part of a mapped function, an actor sinks data to an entity 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 to entity.

`source(src, dst) -> ()`

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, ep2, 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, n) -> ()`

`unlink(e, n) -> ()`

`property(e, pname, pval) -> ()`

A A Worked Example

B Neo4j Encoding

C CDM Encoding