

# PVMv2 Specification

Thomas Bytheway      Lucian Carata

September 17, 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.

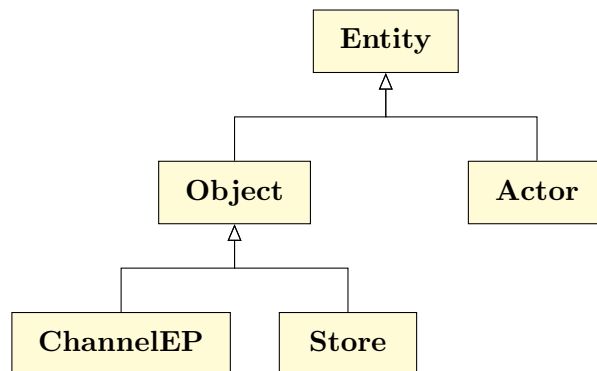


Figure 1: A diagram of the basic abstract types in PVMv2

**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

### 3.1 Nodes

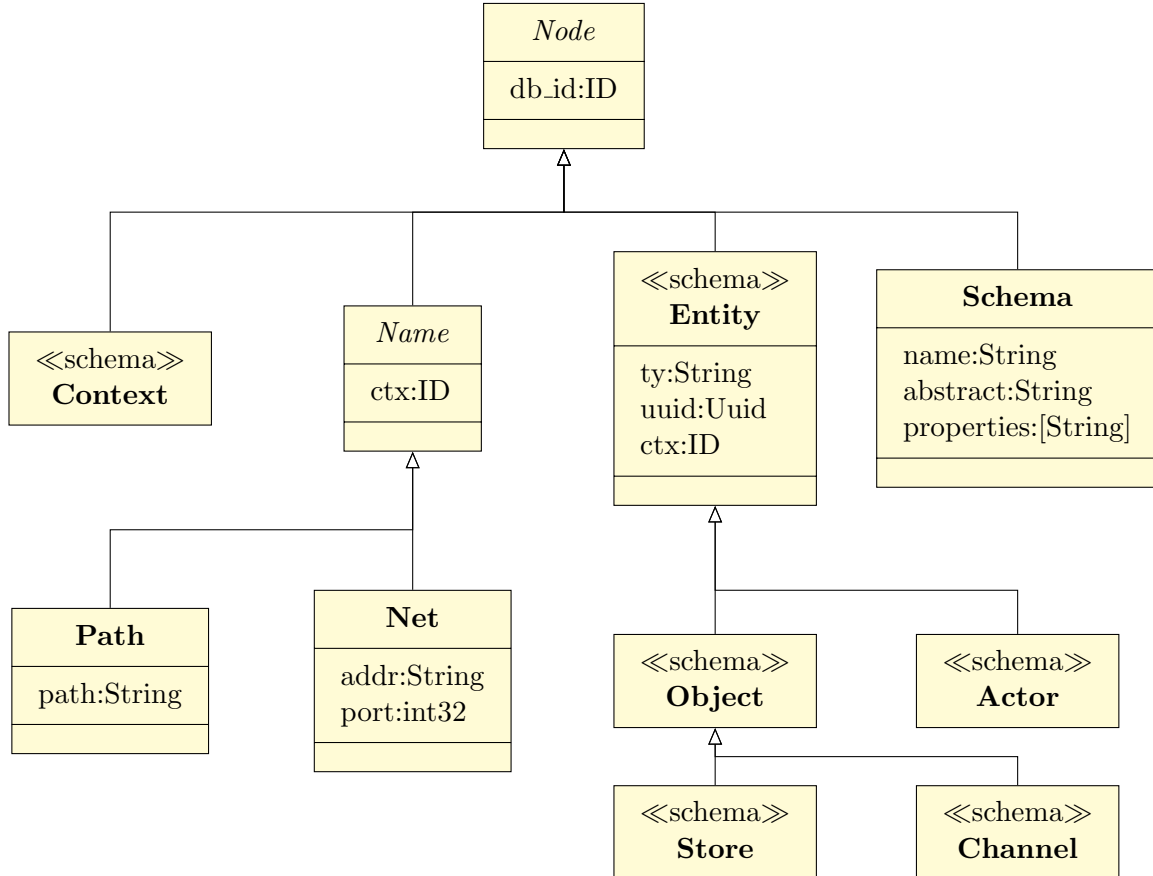


Figure 2: A diagram of the potential nodes in a PVMv2 database.

#### 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 earlier. 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

#### Path Names

#### Network Addresses

## 3.2 Relations

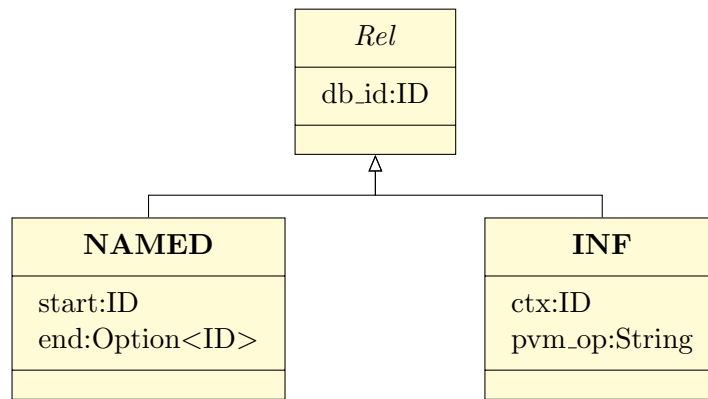


Figure 3: A diagram of the potential relations in a PVMv2 database.

**INF** Indicates that the source has potentially imparted data or control to the destination.

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

## 4 Mappings

### 4.1 Concrete Type Declaration

### 4.2 Verbs

**Declare** Forces the creation of entity if it does not already exist.

**Sink** 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** Called when as part of a mapped function, an actor sources data from an entity. Creates an INF relation from entity to actor.

**Connect** 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**

**Unlink**

**Property**

## 5 A Worked Example