John R. Smith
*IBM*

# VERL: An Ontology Framework for Representing and Annotating Video Events

**Alexandre R.J. François and Ram Nevatia**
***University of Southern California***

**Jerry Hobbs**
***Information Sciences Institute, USC***

***Robert C. Bolles***
***SRI International***

Video annotation lets users rapidly index data in large multimedia databases and make inferences about the data stream's content. A human operator can specify gross indices, such as the date or hour, and perhaps the location and the activity (for example, a football game) at the time of capture. Finer-grain annotations require separating the video into meaningful segments such as shots and scenes. Many automated techniques and annotation formats for capturing this type of information exist. However, the descriptions they produce don't define the data stream content adequately for semantic retrieval and reasoning. Such descriptions must be based on observable or inferrable events in the data streams.

We describe a framework for video event representation and annotation that's based on the definition of an ontology suitable for video content. An ontology consists of a specific vocabulary for describing a certain reality and a set of explicit assumptions regarding the vocabulary's intended meaning.[1] The Video Event Representation Language (VERL)[2-4] describes an event ontology, and the Video Event Markup Language (VEML)[4] lets us annotate instances of the events described in VERL (see the sidebar "Using VERL and VEML" on p. 78 for an example).

Annotating video in VEML consists of describing instances of events and objects, in a previously defined ontology using VERL. Figure 1 illustrates the conceptual elements involved and their relationships: annotations draw on one or several domain-specific ontologies, defined according to VERL concepts and constructs. VEML incorporates extensible event and object type hierarchies rooted in VERL's event and object concepts. The annotations also draw on VEML for content organization. Marking up data streams in a well-defined language rooted in an ontology enables nonambiguous data sharing among users. Furthermore, annotation data is accessible to automatic machine manipulation for indexing or inferencing.

The framework we describe resulted from discussions at a series of workshops sponsored by the US government's Advanced Research and Development Activity (ARDA). It therefore includes the intellectual contributions of many individuals. Currently, only a small community of researchers is using VERL and VEML; this article aims to publicize the framework to a broader community as its relevance depends on its widespread adoption. An earlier description can be found elsewhere.[5]

## Editor's Note

The notion of "events" is extremely important in characterizing the contents of video. An event is typically triggered by some kind of change of state captured in the video, such as when an object starts moving. The ability to reason with events is a critical step toward video understanding. This article describes the findings of a recent workshop series that has produced an ontology framework for representing video events—called Video Event Representation Language (VERL)—and a companion annotation framework, called Video Event Markup Language (VEML). One of the key concepts in this work is the modeling of events as composable, whereby complex events are constructed from simpler events by operations such as sequencing, iteration, and alternation. The article presents an extensible event and object ontology expressed in VERL and discusses a detailed example of applying VERL and VEML to the description of a "tailgating" event in surveillance video.

*—John R. Smith*

## VERL

VERL is a formal language for representing events for designing an ontology for an applica-



1070-986X/05/$20.00 © 2005 IEEE

tion domain and for annotating data with that ontology's categories.

In VERL, we describe complex activities by composing simpler activities in a hierarchical framework. The lowest-level events are *primitive* events. *Composite* events are defined by compositions of lower-level events. Sequencing is the most common composition operation. For example, we'd describe an event involving a person getting out of a car and going into a building using the following sequence: opening car door, getting out of car, closing car door (optional), walking to building, opening building door, and entering building.

VERL also includes more complex composition operations such as iteration and alternation. Composite events containing multiple simultaneous events are called *multithreaded*. In such events, the event description must also comprise temporal relationships between subevents. To do this, we use Allen's interval algebra,[6] which defines qualitative relations between intervals. We could also use a more complete time ontology such as OWL-Time (http://www.isi.edu/~pan/OWL-Time.html), a time ontology for the Semantic Web, written in the Web Ontology Language (OWL).

This representation scheme lets us construct a large variety of composite events from relatively few common primitive events. The definition of primitive events is context dependent. We can consider walking a primitive, or we can describe it in terms of composition of leg motions. We also describe objects hierarchically, constructing complex objects from simpler ones, thus being able to construct many objects from a limited set.

VERL draws on previous research in computer vision and knowledge representation. In the computer vision community, Ivanov and Bobick suggested using context-free grammars.[7] Nevatia, Zhao and Hongeng, advocated using hierarchical decomposition and the single/multiple thread terminology.[8] Vu, Brémond, and Thonnat have developed similar concepts.[9] In artificial intelligence literature, Narayanan developed a formalism for executing actions and applied it to several linguistics problems.[2] This work has influenced the development of the process component of OWL-S, an ontology of services for the Semantic Web.

### Language constructs

We now briefly describe the elements and constructs of VERL.

**Objects, states, and events.** *Objects* have properties or attributes, which we can logically think of as one-argument predications. Objects can also stand in relation to other objects; we think of these as predications with two or more arguments.

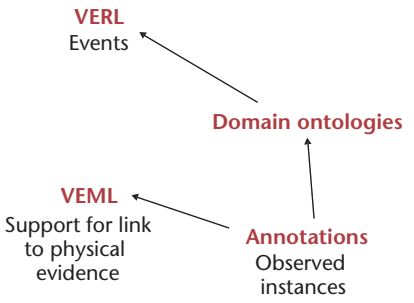


*Figure 1. Components of ontology-based event-level annotation.*

We think of properties, attributes, and relations as *states*. The term "state" often means the aggregate of all the properties and relations of all relevant entities at a given moment in time. In ordinary language, however, we talk about the state of a single entity that persists over time—for example, the state of John's being sick. We use this notion of state in this article.

A change of state in an object triggers an *event*. Thus, when a rock rolls down a hill, its location changes. We can write `change(p(x), q(x))` or `change(at(x, y), at(x, z)`. Events generally occur in a specified interval. They typically have locations as well, inherited from the participants' locations. We could also import a location ontology.

**Types.** An annotation is a pair consisting of a thing in a VERL ontology and a designation of a location in the video data—that is, `<thing, loc>`.

Three basic *types* exist in the language. Everything is a *thing*. Thing describes a physical object, a state, or an event. Two types of things exist: *ents* and *evs*. The type ent encompasses entities, and can generally be thought of as physical objects, although some applications apply the term more broadly. The type ev encompasses states and events. Normally, a person is of type ent, and his or her running is of type ev. In some applications, we can expand this hierarchy to more specific types. For example, we could introduce *person* and *vehicle* as subtypes of type ent.

An annotation is a pair consisting of a thing in a VERL ontology and a designation of a location in the video data—that is, `<thing, loc>`. We can specify locations in any standard formalism including local or world coordinates.

## Using VERL and VEML

To illustrate the Video Event Representation Language (VERL) and the Video Event Markup Language (VEML), we consider *tailgating*, an action that involves gaining access to a secure facility by entering behind an authorized individual. Obviously, many subscenarios correspond to tailgating. As an example, we consider a version that consists of a sequence of subevents involving two people, *x* and *y*, and a door. The sequence is as follows: person *x* approaches the door, unlocks the door, and goes through (while person *y* is nearby), and then *y* enters the door (without having unlocked it).

An observation of this event would consist of the observation of an instance of each subevent type, consistently involving instances of the required objects, at times that are compatible with the constraints imposed by the sequence. From these observations, a human or machine could infer the occurrence of a complex event of type tailgating. A corresponding VEML annotation should specify the instance's characteristics and link it to its supporting evidence.

Figure A shows a sample set of VERL definitions leading to the description of a tailgating event.

The VEML file portion in Figure B encodes the global information about the data and scene, and then describes the objects and events detected in the data (scene-centric part only for legibility). In this example, four objects of interest exist:

```
SUBTYPE(person, ent)
SUBTYPE(facility, ent)
SUBTYPE(portal, ent)
SUBTYPE(door, portal)
SUBTYPE(window, portal)
```

**(a)**

```
PRIMITIVE(container(ent x))
PRIMITIVE(mobile(ent x))
PRIMITIVE(open(portal x))
PRIMITIVE(closed(portal x))
PRIMITIVE(locked(portal x))
PRIMITIVE(unlocked(portal x))
PRIMITIVE(portal-of(portal p, container
c))

//x is on the inside of the container y
PRIMITIVE(inside-of(ent x, ent y))
 // (= close) x is within
    // some distance of y, where the
    //distance is context dependent.
PRIMITIVE(near(ent x, ent y)
```

**(b)**

```
RULE(IMPLY(person(x), mobile(x))
//people are mobile
RULE(IMPLY(facility(x), container(x))
//all facilities are containers
RULE(IMPLY(portal(p), AND(container(c),
//portal-of(p, c))))
//portal open => container
RULE(IMPLY(AND(portal-of(p, c), open(p)),
//container(c)))
```

**(c)**

```
PROCESS(far(ent x, ent y),
NOT(near(x, y)))
PROCESS(outside-of (ent x, ent y),
NOT(inside-of (x, y)))

PROCESS(approach(ent x, ent y), cause(x,
change(far(x,y), near(x, y))))
PROCESS(leave(ent x, ent y), cause(x,
change(near(x, y), far(x, y))))
PROCESS(exit(ent x, ent y), change(inside-
of(x,y)),outside-of(x,y)))
PROCESS(enter(ent x, ent y),
change(outside-of(x,y)),inside-of(x,y)))
PROCESS(unlock(portal p),
change(locked(p)),unlocked(p)))
PROCESS(open(portal p),
change(closed(p)), open(p)))
```

**(d)**

```
//definition of a Tailgating event – x is
//near y when y gets access to a facility,
//& then x enters behind y
//without authorization
SINGLE-THREAD(tailgate(ent x, ent y,
facility f),
    AND ( portal-of(door, f),
      Sequence(
        approach(y, door),
        unlock(y, door),
        open(y, door),
        AND(enter(y, f), near(x, y)),
        NOT(unlock(x, door)),
        enter(x, f))))
```

**(e)**

*Figure A. Sample set of VERL definitions: (a) subtypes of entities (a partial taxonomy of ents for this example); (b) primitive properties of ents (if you were to specify mobile and container as subtypes of ent, you couldn't have mobile containers); (c) rules associated with ents and evs; (d) processes describing relationships, events, and so on; and (e) description of a tailgating event.*

- Person1, who unlocks the door and enters the facility legitimately,
- Person2, who enters by following Person1 through the door,
- Facility1, the locked container that the two people enter, and
- Door1, the door through which the two people enter the facility.

Several events might be detected and encoded in the file, but some of the key events for this example are

- Person1 approaches Door1,
- Person2 follows Person1,
- Person1 unlocks Door1, and
- Person2 enters Door1 by tailgating Person1.

```
<scene>

<ontology>
<source>…/ontologies/physicalSecurity.verl
</source>
</ontology>

<streams>
     <video id="sneak02">
      <offset unit="frames">0</offset>
      <duration unit="frames">450
      </duration>
      <samplingRate>30</samplingRate>
      <source>/home/dvtt2/IU/video/
      data/sneak02/sneak02.sriv
      </source>
    </video>
</streams>

<context>
<!— To Be Determined —>
</context>

<objects>
    <object type="PERSON" id="OBJECT1">
      <property name="name" value=
      "Person1"/>
      <tracks></tracks>
    </object>
    <object type="PERSON" id="OBJECT2">
      <property name="name" value=
      "Person2"/>
      <tracks></tracks>
    </object>
    <object type="FACILITY" id=
      "OBJECT3">
      <property name="name" value=
      "Facility1"/>
      <tracks></tracks>
    </object>
    <object type="ENTRANCE" id=
      "OBJECT4">
      <property name="name" value=
      "Door1"/>
      <tracks></tracks>
    </object>
</objects>
```

```
<events>

    <event type="APPROACH" id="EVENT1">
      <begin unit="frames">136</begin>
      <end unit="frames">247</end>
      <property name="name" value=
      "Approach1"/>
      <argument argNum="1" value="Person1"/>
      <argument argNum="2" value="Door1"/>
    </event>
    <event type="FOLLOW" id="EVENT2">
      <begin unit="frames">177</begin>
      <end unit="frames">247</end>
      <property name="name" value=
      "Follow1"/>
      <argument argNum="1" value=
      "Person2"/>
      <argument argNum="2" value=
      "Person1"/>
    </event>
    <event type="UNLOCK" id="EVENT3">
      <begin unit="frames">260</begin>
      <end unit="frames">332</end>
      <property name="name" value=
      "Unlock1"/>
      <argument argNum="1" value="Person1"/>
      <argument argNum="2" value="Door1"/>
    </event>
     …
    <event type="TAILGATE" id="EVENT12">
      <begin unit="frames">177</begin>
      <end unit="frames">508</end>
      <property name="name" value=
      "Tailgate1"/>
      <argument argNum="1" value=
      "Person1"/>
      <argument argNum="2" value=
      "Person2"/>
      <argument argNum="3" value=
      "Facility1"/>
    </event>
</events>

</scene>
```

*Figure B. Example file-encoding events in VEML.*

**VERL expressions.** In VERL, variables and constants can be any one of the three types.

We define a VERL expression (vexpr) as

- a constant or variable: `vexpr → constant | variable`. For example, John, X1, Fire-1, and E1 can all be vexprs. The vexpr type is the constant or variable type. Thus, if John is an entity constant, then E1 will be an event variable if, for example, it refers to John's running, and so on.

- a function symbol applied to the appropriate number of vexprs as an argument: `vexpr → fct "(" [ vexpr { "," vexpr }* ] ")"` (square brackets indicate an element is optional—for example, the function might have no arguments; curly brackets group elements; and the Kleene star [*] means zero or more instances). The arguments must be of the right type. For example, if *head-of* is a function taking one entity vexpr as its argument, then head-of (John) is a vexpr. The function determines the resulting thing type. Thus, head-of would be an entity function and head-of (John) would be an entity.

- a predicate symbol applied to the appropriate number of arguments: `vexpr → pred "(" [ vexpr { "," vexpr } * ] ")"`. The arguments must be of the right type, and the result is always of type ev. For example, if "change" is a predicate symbol relating two things of type ev, then change (E1, E2) is a vexpr of type ev.

- a logical operator applied to the appropriate number of vexprs of type ev:

```
vexpr →

"AND" "(" vexpr {"," vexpr } * ")" |
"OR" "(" vexpr {"," vexpr } *
")" |

"IMPLY" "(" vexpr "," vexpr ")" |
"NOT" "(" vexpr ")" |

"EQUIV" "(" vexpr "," vexpr ")"
```

- AND and OR take one or more arguments. IMPLY and EQUIV take two arguments. NOT takes one argument. The result is always of type ev. Things of type ev can be event types or event tokens. NOT takes an event type as its argument. Thus, if we say that `NOT (run(John))` occurs at a location in the video data, we're saying that no event of type `(run(John))` occurs there.

We can use a constant or variable as a label on a vexpr: `vexpr → {constant | variable} ":" vexpr`.

The resulting vexpr refers to the same thing as its constituent vexpr and is of the same type. We can use the label elsewhere to refer to that thing. Without labels, ambiguities could result. Suppose we refer to John's running twice in a file: `run(John) ... run(John)`. These might or might not be the same instance of running. If we say `E1: run(John) ... E1`, they refer to the same instance of running.

**Defining composite events in VERL.** *Process* is the basic operator for defining composite events. It takes a predication and a vexpr as its two arguments. The predication is a predicate applied to the appropriate number of arguments where the arguments have an optional type specification:

```
defn → "PROCESS" "(" pred "("
   [argspec {"," argspec}* ")"
   ["," vexpr] ")"

argspec → type variable | variable
```

The second process argument is optional, and if it's missing, we assume the process is primitive—that is, it's directly implemented in software in the given application. For example, if we have the predicate *located-at* relating a thing to an entity, and a predicate change relating two things of type ev, we can define the predicate move as

```
PROCESS(move(thing x, ent y, ent z)

   change(located-at(x, y),located-
     at(x, z)))
```

That is, for a thing *x* to move from entity *y* to entity *z*, *x*'s location must change from *y* to *z*.

Labels defined inside a process statement are local to that process statement. Thus, if we write

```
PROCESS(move(thing x, ent y, ent z),
```

```
change(e1: located-at(x, y),
  e2: located-at(x, z))),
```

we can't write e1 or e2 outside of the process statement and expect it to refer to the same entity it refers to inside the process statement.

We can use three other operators in place of process: *primitive*, *single-thread*, and *multithread*. We use primitive when the process definition has no second argument. Primitive means that the predicate isn't defined in VERL but is implemented in code directly. Single-thread means that all constituent events in the definition happen sequentially without overlap. We use multithread when no such constraint exists. For example, if located-at and change are primitive predicates, a move event is a single-thread event. We'd rewrite the previous example as:

```
PRIMITIVE(located-at(thing x, ent y))

PRIMITIVE(change(ev e1, ev e2))

SINGLE-THREAD(move(thing x, ent y,
  ent z), change(located-at(x, y),
    located-at(x, z))
```

**Inference rules.** In addition to annotating specific events and defining composite properties, relations, and events, we might also want to specify inference rules that let us draw conclusions from what we recognize in the data. For this we use the operator *rule*, which takes two vexprs of type ev as its arguments: `Rule(vexpr, vexpr)`.

A rule is an implication; the first vexpr implies the second. For example, suppose we define `carry(x, y, a, b, t) (x carries y from a to b during time interval t)`, as *x* holds *y* during *t* and *x* moves from *a* to *b* during *t*:

```
PROCESS(carry(x, y, a, b, t),

  AND(hold(x, y, t), move(x, a, b, t)))
```

Then, if we want to say that when *x* carries *y* from *a* to *b* during *t*, *y* also moves from *a* to *b* during *t*: `RULE(IMPLY(carry(x, y, a, b, t), move(y, a, b, t)))`.

We interpret variables in the implication's antecedent as universally quantified and variables occurring in the consequent but not in the antecedent as existentially quantified. Thus, we interpret `RULE(p(x, z), q(x, y))` as `(A x, z) [p(x, z)` $\rightarrow$ `(E y) q(x, y)]`.

**Control structures.** Constructing complex events by composing simpler events is central to our representation scheme. We distinguish between single-thread events, in which only one event is happening at a given time, and multithread events, in which more than one event is happening.

As mentioned earlier, sequence is the most fundamental relation among component events. First one event occurs and then another event occurs. We can encode this if we reify events—that is, treat events as individuals to which constants and variables in our logic or language can refer. The expression `Sequence(e1, e2)` describes the composite event consisting of event e1 followed by event e2 where the events occur in sequence and don't overlap. To construct longer sequences, we let *sequence* take an arbitrary number of arguments: `Sequence (e1, e2, e3, ...)`. The resulting vexpr describes the composite event consisting of all the argument events occurring sequentially.

VERL also defines control structures for iteration or loops, as described elsewhere.[2]

**Temporal relations.** Representing the temporal relations among component events is crucial in recognizing composite events. For most applications, describing the relations among the temporal intervals occupied by the component events, according to Allen's interval algebra,[6] is sufficient. This is because agents respond primarily to other agents' actions or moving objects' behavior. This is true even if one of the threads is precisely timed, such as a conveyor belt in a factory where the worker is responding primarily to the part's appearance rather than the passage of a certain amount of time.

Time comes in two varieties: instants and intervals. Thus,

```
SUBTYPE(temporal-entity, thing)

SUBTYPE(instant, temporal-entity)

SUBTYPE(interval, temporal-entity)
```

Of two distinct instants, one is before the other. The predicate after is the inverse of before: `before(t1, t2), after(t1, t2), t1 = t2`.

An instant *t* and an interval T can be in several possible relations: `begins(t, T), inside(t, T), ends(t, T)`. It's possible that none of these is true.

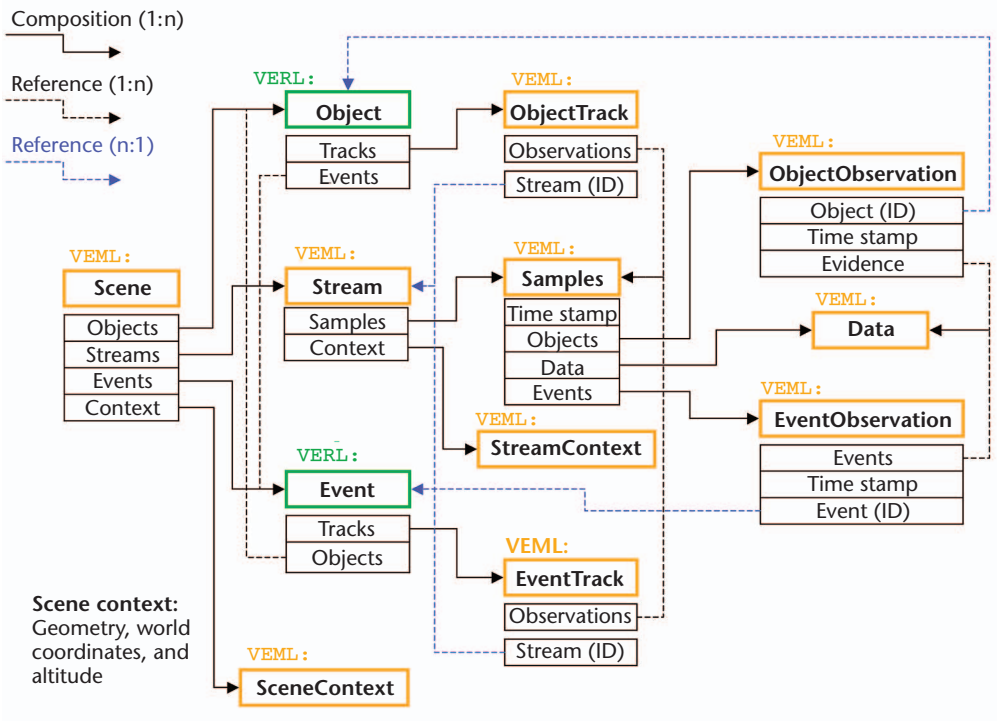


*Figure 2. VEML scene description structure.*

Six possible basic relations exist between two intervals: `before(T1, T2)`, `meets(T1, T2)`, `overlaps(T1, T2)`, `begins(T1, T2)`, `contains(T1, T2)`, and `ends(T1, T2)`. These relations form the basis of Allen's interval algebra. We define them in terms of begins, inside, and ends relations between instants and intervals.

Two possible relations exist between events and times:

- Events can occur instantaneously: `at-time (e, t)`, where *t* is an instant.
- Events can occur across intervals with a duration: `during(e, T)`, where T is an interval.

The OWL-Time (OWL is the Web Ontology Language; see http://www.w3.org/2004/OWL) ontology elaborates these concepts and includes measures of duration, clock and calendar terms, and temporal aggregates.[3]

### VERL semantics

Although VERL expressions are designed to appear natural to human users, each expression corresponds to a formal first-order logical statement. Thus, standard inference engines can reason with VERL-derived annotations. We omit the details here due to their complexity and the lack of space.

### Hierarchies of domain-specific ontologies

VERL allows encoding of knowledge with various degrees of generality, resulting in hierarchies of domain-specific ontologies in which concepts with a narrower range of relevance are defined in terms of more general concepts. Examples of general ontologies include time, time relationships, and spatial relationships. For example, we could divide an ontology of objects and video events pertaining to physical security into more specific subdomains, such as bank monitoring, outdoor surveillance, and railroad crossing monitoring.[3] Each subdomain is likely to draw on common concepts that it refines or specializes in its specific context.

## VEML

VEML is a language for recording the observation of instances of concepts defined in an event and object ontology specified in VERL. VEML consists of a set of structures, compatible with the VERL definitions, that allows links to physical (media) evidence. Figure 2 shows VEML's top-level structures and their relationships. The scene, defined as a convex region of space and time, represents the largest scope of a VEML annotation (and thus its root). A scene consists of objects (instances), and events (instances) involving the objects (in the remainder of the article, we'll specify "instance" only when it's

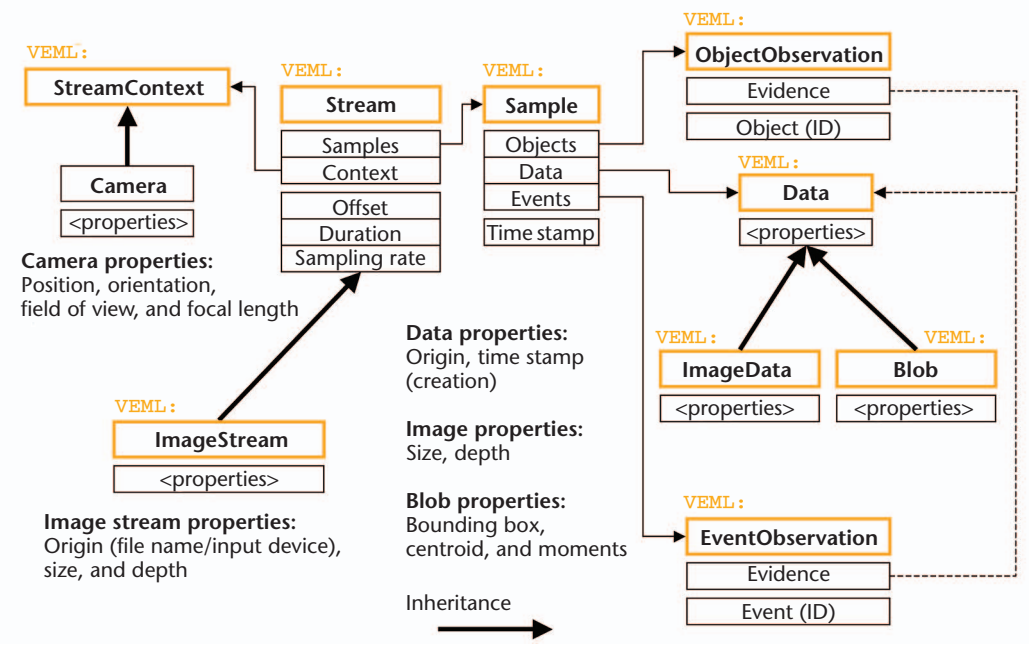


*Figure 3. Stream specialization.*

ambiguous). The scene can be described in various recordings or data streams.

Scene context data, such as geometry, world coordinates, and geographical information, are the high-level scene components. Each component type is the root of an extensible-type hierarchy allowing specialization. Furthermore, at this level, all entities are defined in the scene over a period of time and carry start and duration (or end) time data. Instances of these types therefore encode information in a scene-centric approach, independent of stream-level specifics.

As the following sections describe, streams, objects, and events structures and their relationships represent the most powerful and distinguishing aspect of VEML.

## Streams

In VEML, a scene description can involve several data streams of different types and durations, overlapping (or not) arbitrarily. Figure 3 details the defining attributes of a generic data stream.

We add traditional metadata through specialized context data structures or by deriving specialized stream structures from the base stream class (such as image or audio streams). Any data stream, however, consists of a sequence of samples. Each time-stamped sample carries associated data. The data associated with a sample include both raw data and processed data. We define data structures for the data through specialization (image data, blobs, and so on). In addition, each sample carries a list of object observations and a list of event observations. These structures establish a relationship between scene-centric and stream-centric description (data evidence).

## Objects and events

VEML's object and event concepts are rooted in VERL, so the corresponding structures are quite similar.

Figure 4 (next page) illustrates how VEML's extensible event-type hierarchy encodes the relevant object ontology (regrouped in domain-specific ontologies, or d-onts). For both analytical and retrieval purposes, object instances must be linked to relevant observations in data streams.

As mentioned earlier, an object instance will occur in individual samples of data streams. The object observation structure, stored in the sample's object (observations) list, establishes the link between the object instance (ID) and the evidence in the data attached to the sample. The list of consecutive observations of a given object in a given stream constitutes a trajectory (a term directly taken from the object-tracking application domain).

The corresponding structure is stored in an object's tracks list, and establishes the link to a stream and the object observations in the stream (via the samples). If complete information is stored, the data structures allow, for example, retrieval of all observations of a given object in a given stream (trajectory), in all streams, or,

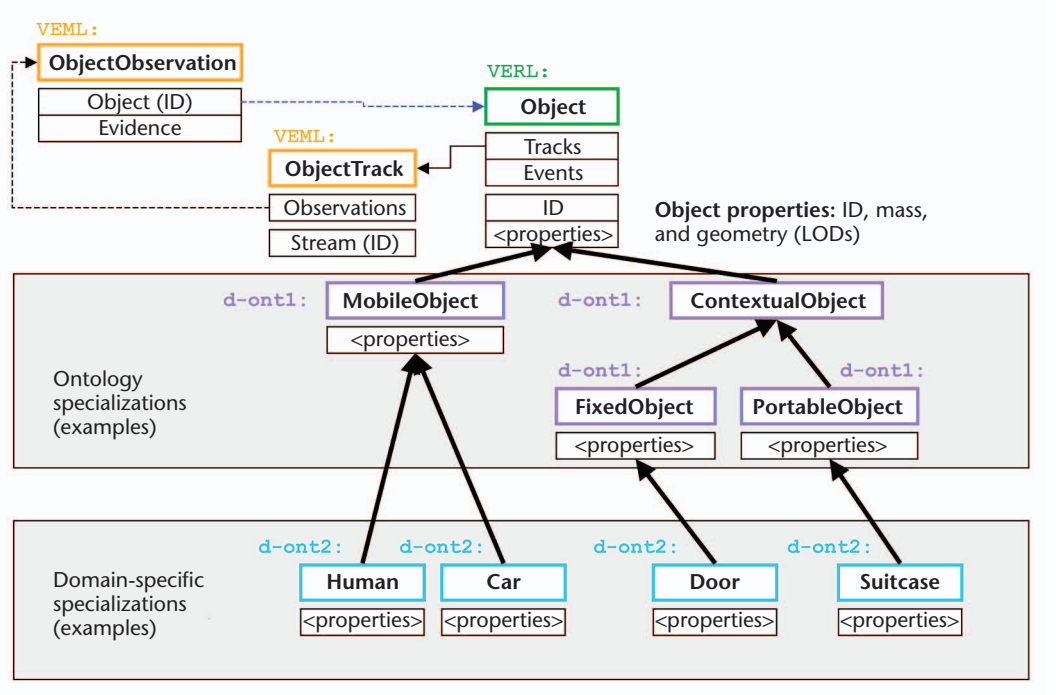


*Figure 4. Object ontology encoding. D-ont stands for domain ontology.*

reversely, retrieval of the corresponding object instance from an observation in a sample. This last case corresponds to the scenario in which an operator, while watching an annotated video, pauses and clicks on an object (area in a frame) to query information about the object defined in the scene. Finally, object instances point to all of the event instances in which they participate, and, inversely, events reference all objects involved.

Figure 5 shows the data structures for event ontology encoding. We achieve correspondence between scene-centric, stream-centric, and sample-level event instance encoding in the same way as for object instances. The root generic event type is specialized to capture the generic event types defined in VERL: primitive, single-threaded (sequence), and multithreaded events. In particular, encoding complex event instances captures the relationships with corresponding subevent instances. This connection occurs at the scene-centric level.

## Implementation

VEML underwent two proof-of-concept implementation phases. The first implementation was VEML 1.0.[3] Its primary purpose was to validate the underlying concepts; it took the form of a direct encoding of the data structures as a constrained subset of the Extensible Markup Language (XML, http://www.w3.org/XML) using XML document type definitions and XML schema. Annotation files produced for a reference video corpus conformed to this syntax.

VEML 2.0 is the latest expression of VEML,[4] and is in the form of a constrained subset of OWL called OWL Description Language, or OWL-DL (http://www.w3.org/2004/OWL), itself a constrained subset of XML. OWL-DL provides a type system supporting inheritance, containers, and references needed to encode and support the data structures defining VEML. Because VEML imports VERL elements, at least a partial expression of VERL in OWL is also required, although OWL-DL isn't powerful enough to express all VERL constructs.

Using OWL to express VEML lets us leverage existing and future tools based on both OWL and underlying standards. For example, Jena is a Java framework for building Semantic Web (http://www.w3.org/2001/sw/) applications. It provides "a programmatic environment for RDF [resource description framework], RDFS [RDF schema], and OWL, including a rule-based inference engine" (http://Jena.sourceforge.net). Protégé (http://protege.stanford.edu)[10] is an integrated software tool that was particularly useful for developing the current VEML prototype in OWL. Furthermore, Protégé is an appropriate environment for encoding and visualizing domain-specific event and object ontologies. However, optimal user

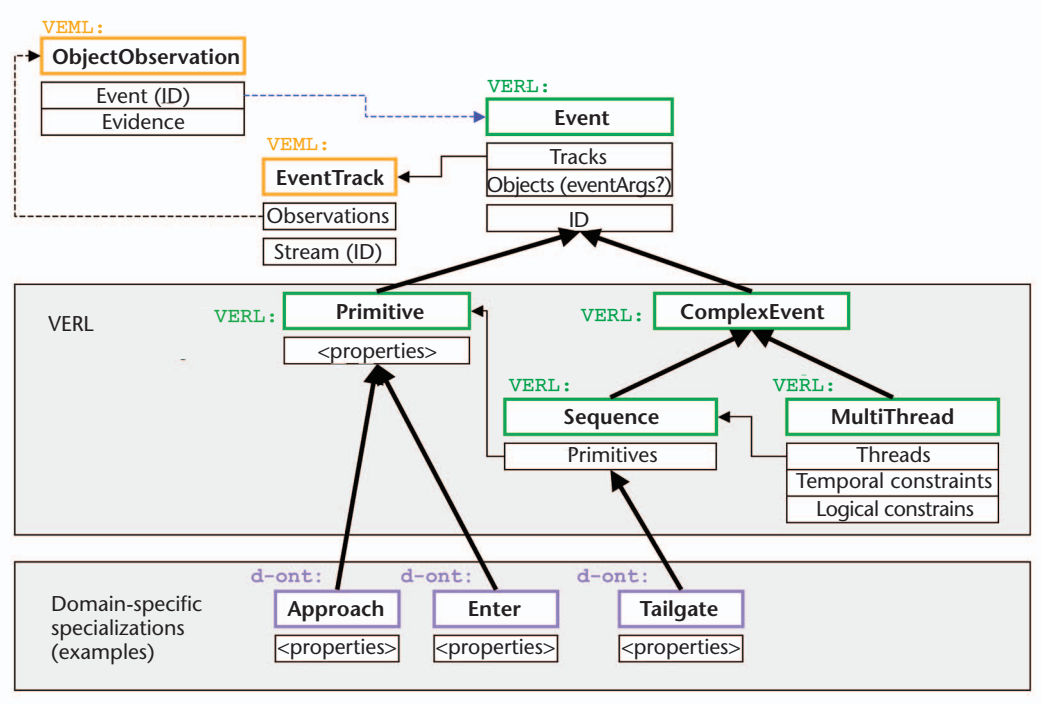


*Figure 5. Event ontology encoding.*

experience requires dedicated video scene annotation and visualization tools.

VEML is extensible, and the definition of the underlying data structures is left open for more specific elements. Specific applications might therefore require specific data formats and descriptors—such as a method-specific data format for supporting evidence in the observation of a specific event type in video frames. This is also where we could import descriptors from existing standards, such as MPEG-7.[11] VEML's distinguishing feature is the underlying set of high-level data structure encoding and relating the event ontology and scene-centric and stream-centric representations. We could also implement these elements in the context of an existing formalism other than OWL, such as MPEG-7, and possibly consider integrating them in the standard.

## Conclusion

We described an extensible and hierarchical framework for video event ontology and annotations. For this framework to be useful, detailed domain ontologies and tools for annotation need to be developed. Integration of this framework with MPEG-7 standards would be another important step. We hope that a community of users will emerge resulting in accomplishment of these objectives. **MM**

## Acknowledgments

The US government's Advanced Research and Development Activity funded this research under a contract to the Department of Energy's Pacific National Northwest Laboratory. We are grateful to the workshop participants, who are listed in a more complete report.[4]

## References

1. S. Bechhofer et al., *OWL Web Ontology Language Reference*, Feb. 2004, http://www.w3.org/TR/owl-ref.
2. S. Narayanan, *KARMA: Knowledge-Based Action Representations for Metaphor and Aspect*, PhD dissertation, Univ. of California, Berkeley, 1997.
3. B. Bolles and R. Nevatia, *ARDA Event Taxonomy Challenge Project Final Report*, Feb. 2004; https://rrc.mitre.org//nwrrc/event-taxonomy-final-report.pdf.
4. B. Bolles and R. Nevatia, *A Hierarchical Video Event Ontology in OWL*, ARDA Challenge Workshop Report, 2004; https://rrc.mitre.org//nwrrc/OWL-events-final-report.pdf.
5. R. Nevatia, J. Hobbs, and B. Bolles, "An Ontology for Video Event Representation," *Proc. IEEE Workshop on Event Detection and Recognition*, IEEE Press, June 2004.
6. J.F. Allen and G. Ferguson, "Actions and Events in Interval Temporal Logic," *Spatial and Temporal Reasoning*, O. Stock, ed., Kluwer Academic Publishers, 1997, pp. 205-245.

7. Y.A. Ivanov and A.F. Bobick, "Recognition of Visual Activities and Interactions by Stochastic Parsing," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 22, no. 8, Aug. 2000, pp. 852-872.
8. R. Nevatia, T. Zhao, and S. Hongeng, "Hierarchical Language-Based Representation of Events in Video Streams," *Proc. Workshop Event Mining* (in conjunction with IEEE Int'l Conf. Computer Vision and Pattern Recognition [CVPR]), IEEE CS Press, 2003, p. 39.
9. T. Vu, F. Brémond, and M. Thonnat, "Automatic Video Interpretation: A Novel Algorithm for Temporal Scenario Recognition," *Proc. 18th Int'l Joint Conf. Artificial Intelligence*, Springer, 2003, pp. 523-533.
10. N.F. Noy et al., "Creating Semantic Web Contents with Protégé-2000," *IEEE Intelligent Systems*, vol. 16, no. 2, 2001, pp. 60-71.
11. J.M. Martinez, ed., *MPEG-7 Overview*, ISO/IEC report no. JTC1/SC29/WG11N5525, Int'l Organization for Standardization, Mar. 2003, http://www.chiariglione.org/mpeg/standards/mpeg-7/mpeg-7.htm.