Semantically Enhanced Virtual Learning Environments Using Sunflower

Daniel Elenius, Grit Denker(✉), and Minyoung Kim

SRI International, 333 Ravenswood Ave, Menlo Park, CA 94025, USA
grit.denker@sri.com
http://www.sri.com

Abstract. Teaching procedural skills is relevant for a broad range of applications, from IT administration to automotive repair to medical diagnostics. Virtual learning environments reduce the cost, time, and risk, and increase the availability of such training. We introduce ontologies and rules to characterize the objects in the learning environment, and the actions that the user can perform on them. These semantic models are used as the basis for automated reasoning about a student's actions and their effects, and guide automated assessment and feedback to the student. We describe our system and models in the context of weapon skills such as disassembling and assembling a rifle.

1 Introduction

Teaching procedural skills is relevant for a broad range of applications, from IT administration to automotive repair to medical diagnostics. While "learning by doing" approaches are highly effective because learners gain knowledge as they solve problems in the relevant environment, cost, time or risk often make it infeasible to provide learning systems in those environments.

Virtual environments (VEs) are a feasible solution that overcome these limitations while still providing "learning-by-doing"-type of user experiences. They also provide the added benefit of flexible delivery platforms that allow users to learn where and when they want.

To provide learning systems based on VEs, various capabilities and automated tools need to be implemented as part of the VE and provide functionality such as context-aware feedback, personalization to adapt learning content to a student's capabilities, or assessment. Such automated tools promise to make learning systems more effective for the individual student and they would both reduce the cost of using VEs for training and open the door to self-directed learning systems, in which users can acquire procedural skills at their own pace.

Traditional approaches to learning require direct observation by an instructor to provide functionality such as assessment, context-aware feedback or adaptation of learning content. Our approach uses semantic technologies to enable the automation of such functionalities. We have developed a framework called Semantically Enabled Automated Assessment in Virtual Environments (SAVE)

© Springer International Publishing AG 2016
E. Garoufallou et al. (Eds.): MTSR 2016, CCIS 672, pp. 81–93, 2016.
DOI: 10.1007/978-3-319-49157-8_7

which can observe learners operating within an instrumented VE, assess their performance, and provide helpful feedback to improve their skills.

At the core of SAVE is the capability to meaningfully understand what a student is doing in the VE, and the effects of those actions on the environment. Consider a VE for teaching a military student how to disassemble, clean and assemble a weapon. Knowledge that the student clicked on a given screen coordinate has very limited use for assessment. Instead, an understanding of the higher-level semantics of performing that mouse click, e.g., "the student released the charging handle while keeping the bolt catch pressed," is essential. This understanding extends beyond knowledge of what was done, and requires insight into important relationships (e.g., spatial, causal, functional) among the objects in the VE. With this level of characterization, the merits of a particular action can be understood: whether it is at all possible given the current state of the weapon, whether the action has the intended effect (e.g., removing an ammunition cartridge from the chamber), whether the student's actions satisfy the security protocol, and whether the action demonstrates specific domain knowledge.

Furthermore, the system should support exploration, where the student is free to choose among a wide range of actions. Emergent, rather than pre-programmed behavior, is key. We need to be able to base the assessment of students' performance on not only the ability to following exact procedures, but also on whether they achieve a given outcome – possibly in an unanticipated way. Furthermore, describing the behavior of objects in the environment (for example, an M4 rifle) is a task that requires domain expertise – not something that should be left to programmers.

1. Point weapon in safe direction.
2. Attempt to place the selector lever on SAFE. Note: If weapon is not cocked, lever can't be pointed toward safe.
3. Remove the magazine from the weapon, if present.
4. Lock the bolt open.
   (a) Pull the charging handle rearward.
   (b) Press the bottom of the bolt catch.
   (c) Move the bolt forward until it engages the bolt catch.
   (d) Return the charging handle to the forward position.
   (e) Ensure the receiver and chamber are free of ammo.
5. Place the selector lever on safe.
6. Press the upper portion of the bolt catch to allow the bolt to go forward.
7. Place the selector lever from SAFE to SEMI.
8. Squeeze trigger.
9. Pull the charging handle fully rearward and release it, allowing the bolt to return to the full forward position.
10. Place the selector lever on SAFE.

Fig. 1. A procedural task: clearing a rifle.

These considerations motivate an approach using declarative specifications that can be created, modified, re-used, and understood by domain experts – a *semantic* approach. In SAVE, we use ontologies and rules to provide semantic characterizations of objects and actions in the domain.

Though SAVE is applicable to any procedural skills, for the purpose of this paper, we discuss the semantic models and the reasoning for a military domain use case: disassembling, cleaning and reassembling a weapon. This task was of interest to our client and exhibited sufficient real-world complexity to challenge our system. Figure 1 shows the procedure, from [1], for clearing a weapon, which is part of a larger set of skills in this context. Note that, while this is a relatively straightforward procedure, in general, procedural skills can have different variants, optional parts, and so forth.

2 SAVE Overview

SAVE employs various components that generate or make use of semantic models. (1) The *Semantic 3D Annotation Editor* (S3D Editor) allows a 3D content author to associate objects in a 3D model with ontological concepts. The ontological concepts are part of a semantic model that is described in detail in Sect. 4. (2) The *Content Assembly Tool* allows a user to build the training-specific 3D scene. A 3D scene consists of various 3D assets, some with annotations (e.g., the objects with which the student will interact in their learning exercise) and others without annotations (e.g., background objects). (3) The *Exercise UI* serves two purposes. It is used by instructors to record a sequence of actions that will serve as a basis for solutions against which the student will be assessed. Because the VE objects were annotated with semantic classes using the S3D Editor, and the scene was assembled using semantically annotated 3D models, the VE can request actions for VE objects (and their components) from the underlying semantic reasoner and visualize them in the Exercise UI. Once the semantically enhanced virtual training environment has been set up, students use the Exercise UI to attempt to perform the intended tasks. The student sees the 3D objects (e.g., the M4 rifle and its components), and is able to apply generic actions (push, pull, etc.) to them. When the student does so, the action and its parameters (i.e., which components were selected) are communicated to the Flora reasoner, which has the M4 ontology loaded. The reasoner determines the effects of the given action, if any, and updates its KB (knowledge base) accordingly. It then communicates back to the Exercise UI the changes in the state of the environment. The UI uses this information to redraw the 3D components in their updated state. (4) The exercise solution editor shows the action traces to the instructors and allows them to add annotations to capture permissible generalizations to the solution. The generalized solution is the basis for SRI's assessment capability, which is designed to accommodate more open-ended procedural skills for which there can be a range of solutions with significant variations among them. (5) As the student interacts with the VE for a learning task, her actions are recorded as a semantically annotated action trace. The *Automated Assessment* component within SAVE analyzes semantic traces of learner actions against the generalized solution trace and provides contextually relevant feedback.

Details about the automated assessment and user studies for this use case are reported in [3], and the solution editor is described in [6]. This paper focuses on the semantic models used by the system, and the reasoning that happens in the VE at run-time, i.e., while the student is using the system for training.

3 Sunflower Overview

Existing languages like OWL, SWRL, and RIF, and associated editing and reasoning systems, do not support many of the features required for modeling virtual training environments. For example, SWRL does not support n-ary predicates, aggregation or higher order expressions, structured output (such as CSV or XML), or tracing or debugging of reasoning with rules. The Sunflower[1] suite is intended to fill this gap. Sunflower is a set of libraries and tools based on the Flora-2 language[2], which in turn is implemented as a layer on top of XSB[3].

Flora-2 is a highly expressive knowledge representation language and associated reasoning engine developed and maintained primarily by Michael Kifer at Coherent Knowledge Systems. While Flora-2 has its origins in the *logic programming* research community, OWL has its root in *description logics*. Flora supports, among other things, n-ary formulas, negation-as-failure, aggregation, higher-order predicates, functions, frame syntax for classes and instances, infix mathematical expressions, prioritized or default rules, and knowledge base update operators. Flora-2 integrates ontologies and rules in a powerful way.

On top of Flora-2, *Sunflower Foundation* is a library, implemented mostly in Java and partially in C/C++ and Flora itself, which provides many features that are essential to building applications based on Flora rules and ontologies. These features include a Flora parser that generates a detailed syntactic representation of Flora content in Java, syntactic manipulation of that representation, a higher-level ontology model, importers and exporters for other languages (RDF, OWL, SWRL, CSV, SQL, etc.), an interface to the Flora reasoner, a live RDF triple store connector, an explanation module that produces structured explanations of reasoning results to the user, and a natural language module that produces English paraphrases of reasoning results and explanations. The other main components of the Sunflower suite are *Sunflower Studio* – an Eclipse-based IDE for working with Flora-2 content, and *Sunflower Server* – a Web server that exposes much of the Sunflower Foundation functionalities over HTTP using REST APIs. More details on the Sunflower suite can be found in [2]. This paper describes how we use it in the SAVE system to represent and reason about actions in semantic VEs.

[1] http://sunflower.csl.sri.com.
[2] http://flora.sourceforge.net/.
[3] http://xsb.sourceforge.net/.

The Flora-2 Language. The authoritative documentation for Flora-2 is its user manual[4]. Here, we give a brief overview of only the features that we use elsewhere in this paper, without precisely defining syntax and semantics.

Terms. Flora identifiers can (optionally) use namespaces and namespace prefixes, as in RDF and OWL. We omit these for readability and space reasons here. There are the usual primitive data values like integers, strings, etc. Data values can be typed, e.g., `"Hello World"^^\string`. The boolean values are written `\true` and `\false`. Lists are written as `[1,2,3]`, optionally with a "tail" part, `[a|b]`. *Functional terms* are written `f(t1,...,tn)`, where the `ti` arguments are themselves terms.

Frames. `A : B` means `A` is an instance of `B`. `A :: B` means `A` is a sub-class of `B`. `A [ p -> V ]` means that `A` has value `V` for property `p` (i.e., this is a *subject, property, object* triple, in RDF terms). We call `[..]` an *instance frame*. `A [| p {m..n} => R |]` means that `A` has range `R` for property `p`, with min-cardinality `m` and max-cardinality `n` (the cardinality part is optional; `m` and `n` are non-negative integers, or `*` for "any"). We call `[|..|]` a *class frame*.

Formulas. Conjunctions of expressions are separated by comma (`,`). Several expressions can be grouped together into one statement, and frame expressions can be nested. For example, `a : A :: B [p -> V, q -> W [r -> Z]] [|p => R|]` is equivalent to `a : A, a :: B, a[p->V], a[q->W], W[r->Z], a[|p => R|]`.

Conjunction can also be written `\and`. Similarly, disjunction uses semi-colon (`;`) or `\or`. There are additional logical operators such as `\if..\then..\else`. There are several types of *negation*, including Prolog-style negation-as-failure, `\+` and Flora's well-founded negation `\naf`. Parentheses can be used to disambiguate operator precedence.

Statements. Flora statements are delimited by a period (.). *Rules* have the form `head :- body`, where `head` and `body` are flora expressions which may contain *variables*. Variables start with a question mark, e.g., `?x`, and may be typed using the `^^` notation. Rules may be preceded by a *rule id descriptor*, `@!{R}`, where `R` is a unique name for the rule.

An object-oriented-style *dot notation* can be used as a shortcut for property chains. For example, `a.b.c` refers to the value of `?x` in `a[b->?y[c->?x]]`.

Comments use the Java/C++ style: `//` for single-line comments, and `/* ... */` for multi-line comments.

Flora also has Prolog-style *predicates*, `p(t1,t2,t3)`. Predicates that have side effects are marked as *transactional* by prepending the name with a percent sign, e.g., `%p`.

Examples of operators that cause side effects are the *knowledge base update* operators, including `insert{p}` and `delete{p}`, for inserting and deleting the fact `p` to/from the knowledge base, respectively. The `writeln` predicate can be used to print to the console.

[4] http://flora.sourceforge.net/docs/floraManual.pdf.

4 Semantic Models

The main components of the semantic models for SAVE are: an ontology of components (physical objects) that the student can interact with, rules for creating components (and their sub-components), an ontology of actions that the student can perform, and rules for performing actions on components. We now describe each of these in turn, followed by examples of querying these models. These models and queries were tested by an in-house subject matter expert.

4.1 Component Ontology

In the SAVE scenario, we focused on procedural tasks around the M4 rifle. Thus, we needed to model the components of this rifle, and their parts structure. Figure 2 shows an exploded component view of the lower half of the rifle. We modeled the components to the level of detail necessary for the tasks we were interested in (clearing the rifle, disassembly, cleaning, and assembly). For other tasks, such as detailed gunsmithing work, a higher level of detail would be required.

We created a simple ontology to capture the meronomy (parts hierarchy) of physical objects, with properties like `hasDirectPart` and `hasRegion`. We also introduced rules to introduce `hasPart` as the transitive closure of `hasDirectPart`, so that we can reason about nested components.

Next, we introduced the specific classes for the M4's components. There are about 80 of these classes in our ontology. Each class has sub-properties of `hasDirectPart` and `hasDirectRegion` to support indication of the correct types

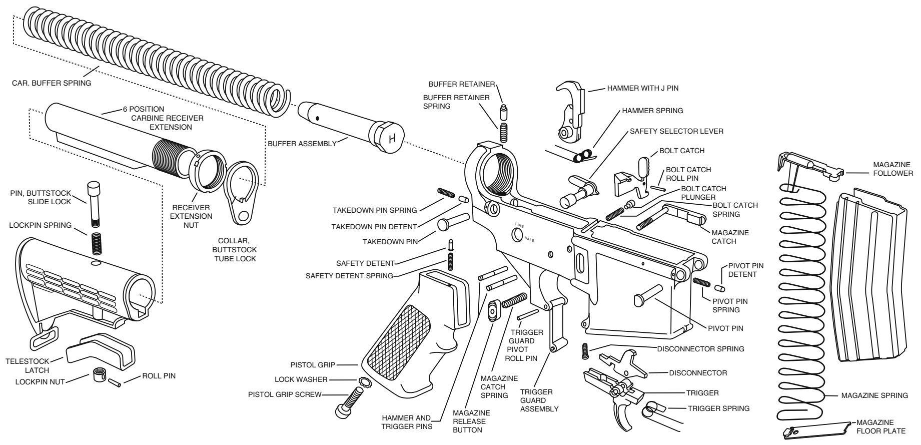

Fig. 2. M4 rifle parts diagram (lower half)

and cardinalities of its sub-components[5]. As an example, the definition of the "lower half" component is:

```
LowerHalf :: PhysicalObject [|
  selector {1..1} => Selector,
  magazine {0..1} => Magazine,
  magazineReleaseButton {1..1} => MagazineReleaseButton,
  hammer {1..1} => Hammer,
  trigger {1..1} => Trigger,
  pivotPin {1..1} => PivotPin,
  takedownPin {1..1} => TakedownPin,
  boltCatch {1..1} => BoltCatch,
  buttStock {1..1} => ButtStock,
  lowerReceiverExtension {1..1} => LowerReceiverExtension,
  bufferRetainer {1..1} => BufferRetainer
|].
```

Note that some of the components may have slightly different names in Fig. 2 due to differences in terminology. The figure shows many more components than the properties of our `LowerHalf` class have. This is primarily because, in our ontology, those components are found under nested sub-components.

The properties `selector`, `magazine`, and so on are all sub-properties of `hasDirectPart`. These all relate to further sub-components, like the `Selector`:

```
Selector :: Switch [| switchPosition {1..1} => SelectorMode |].
```

This component has no further sub-components. Instead, it illustrates another feature of our component classes: the ability to capture the current *state* of the component. The property `switchPosition` indicates the current position of the selector switch. The range class `SelectorMode` is essentially an enumeration of three possible values: `Safe`, `Semi`, and `Burst`. As we shall see, these state properties have essential importance when it comes to modeling the actions that one can perform on the components.

4.2 Component Creation Rules

In our SAVE framework, the student interacts with *instances* of the rifle and its components. Thus, we need to be able to create an instance hierarchy that corresponds to the class-level component hierarchy. Furthermore, we may need several copies of certain components, each with unique identifiers. Doing this manually (or in programming code) is tedious and error-prone. Instead, we define rules which allow us to create component instances, along with all their sub-components. These rules are made possible by Flora's support for *knowledge base update* primitives, which allow us to modify the KB at runtime. We call these rules *constructor rules*, since they are analogous to constructors in object-oriented programming languages. The constructor rule for the `LowerHalf` class is

[5] In OWL, one might instead use qualified cardinality restrictions. Other ways of modeling also exist in Flora.

```
@!{CreateLowerHalfRule}
%create(LowerHalf,?lower) :-
  %create(Selector,?selector), %create(Hammer,?hammer),
  %create(Trigger,?trigger), %create(PivotPin,?pivotPin),
  %create(TakedownPin,?takedownPin), %create(BoltCatch,?boltCatch),
  %create(Magazine,?magazine),
  %create(MagazineReleaseButton,?magreleasebutton),
  %create(ButtStock,?buttstock), %create(LowerReceiverExtension,?lre),
  %create(BufferRetainer,?bufferRetainer), %create_name(LowerHalf,?lower),
  insert{ ?lower : LowerHalf [
      selector -> ?selector, hammer -> ?hammer,
      trigger -> ?trigger, pivotPin -> ?pivotPin,
      takedownPin -> ?takedownPin, boltCatch -> ?boltCatch,
      magazine -> ?magazine, magazineReleaseButton -> ?magreleasebutton,
      buttStock -> ?buttstock, lowerReceiverExtension -> ?lre,
      bufferRetainer -> ?bufferRetainer ] }.
```

All the constructor rules use a common `%create` predicate, which takes two arguments: a component class, and a (resulting) instance object. The rule body has essentially three parts. First, we create all the child components. This step depends on the constructor rules for the sub-components. Secondly, we create a new name for our new component (using the `%create_name` predicate, which we define elsewhere). Finally, we insert into the KB facts which connect the sub-components to the new top-level component, and assert the type and initial state of the component. Now, we can issue a query, `%create(LowerHalf,?x)`. This query will cause Flora to create a number of new instances, each connected in the appropriate way. The variable `?x` will be tied to the top-level instance representing the lower half component itself. Normally, we create the whole rifle in one go, using the top-level `M4` component as the first argument to `%create`.

4.3 Action Ontology

We built a high-level action ontology by adapting the taxonomy in [9] for our needs. The generic actions in our ontology are: `Attach`, `Close`, `Detach`, `Extract`, `Insert`, `Inspect`, `Lift`, `Open`, `Point`, `Press`, `Pull`, `Push`, and `Release`. Each of these is defined as a class, which is a subclass of the `Action` class. A specific action that occurs in space and time is considered to be an instance of the corresponding action class. Each action has a fixed set of parameters. These are defined on the action class. For example, the `Insert` class (here slightly simplified) is defined as:

```
Insert :: Action [
  description ->
    "Insert an object into another object"^^\string,
][|
  thingInserted {1..1} => PhysicalEntity,
  insertedInto {1..1} => PhysicalEntity
|].
```

The action takes two parameters, both of which are physical entities: the thing inserted, and the thing inserted into.

Modeling actions as instances presents us with a problem: We need to create a new instance, and related property assertions, for each individual action that the user takes. This is somewhat cumbersome, especially for testing purposes. Fortunately, Flora has some nice features that provide a solution to this problem. We can define a functional term pattern

```
insert(?_TI,?_II) : Insert [
  thingInserted -> ?_TI, insertedInto -> ?_II
].
```

This allows us to treat functional terms of the form `insert(?x,?y)` as terms, with property value `?x` for `thingInserted` and `?y` for `insertedInto`. We can use such terms directly in queries and rules, without having to explicitly declare a new instance first.

Next, we found that these generic actions were not quite sufficient to model all the intended tasks. At the same time, we did not want to pollute our generic task ontology with very specific tasks. Hence, we introduced a new ontology of "mechanics" actions: `PullAndHold`, `PushAndHold`, `TightenScrew`, `LoosenScrew`, and `SelectSwitchPosition`.

4.4 Action Rules

The final component of our semantic models is the set of *action rules*. These rules describe the preconditions and effects of the different actions, as applied to components of the M4 rifle. This is by far the largest part of our semantic models. As a simple example, we the rule for inserting a magazine is:

```
@!{InsertMagazineRule}
%do(?action^^Insert,?del,?add) :-
  // Action Parameters
  ?action [
    thingInserted -> ?mag^^Magazine,
    insertedInto -> ?lower^^LowerHalf
  ],
  // Preconditions
  ?lower [ magazine -> ?mag [ attached -> \false]],
  // Effects
  ?del = [ ${?mag [attached -> \false]} ],
  ?add = [ ${?mag [attached -> \true]} ],
  %kb_update(?del,?add).
```

Each action rule uses the predicate `%do`, which takes three arguments: the action instance, and two result arguments which we call the *delete-list* and the *add-list*. We will return to these lists shortly. The action variable is typed to the correct type of action (`Insert` in this case). The first part of the rule (*Action Parameters*) retrieves the parameters from the action instance, and checks the types of those arguments. In this case, the value of the `thingInserted` property must have type `Magazine`, and the `insertedInto` must be a `LowerHalf` (this is the part of the rifle that the magazine is inserted into). The second part of the rule is the *Preconditions* part. Here, we can check the state properties on the relevant components, to make sure the action is possible. In this case, we check that the magazine is not already inserted in the rifle. If the preconditions fail, the entire rule fails, and there is no change in the KB. Finally, in the *Effects* part of the rule, we perform the KB updates that represent the change in the world that the action performs. Typically, the KB update modifies the state properties of the components that are involved in the action. The KB updates are performed by a convenience predicate that we introduced (definition not shown here), called `%kb_update`. This predicate takes two arguments: a *delete-list* and an *add-list*. These lists contain the Flora formulas to delete from, and add to, the KB. In the current rule, we simply change the value of the `attached` property on the magazine. These two lists are also returned as result arguments of the entire `%do` predicate, in case the caller needs to know the rule's effects.

For each action rule, we also create a helper predicate that simplifies testing the rule. For example, for the action above:

```
@!{InsertMagazineHelperRule}
insert_magazine(?M4) :- %do(insert(?M4.lower.magazine,?M4.lower),?,?).
```

The action rules are very detailed and some of them get rather complex. Sometimes, the effects of an action are conditional, even after the preconditions have been satisfied. For example, to pull the trigger, the hammer must be cocked, and the selector must not be in the `SAFE` position. The effects of pulling the trigger depend on whether there is: (a) a round in the chamber, (b) a magazine in the magazine well, and (c) additional rounds in the magazine. Because these rules, like the component creation rules, utilize Flora's KB update operations, they are not expressible in less powerful languages such as OWL and SWRL.

4.5 Queries

As mentioned earlier, the action helper predicates can be useful in order to test our action rules. We can also create new predicates that represent sequences of actions, such as the "clearing a rifle" task in Fig. 1:

```
@!{ClearWeaponRule}
%clear_weapon(?M4) :-
  %point_weapon_at_target(?M4,ShootingBerm),
  \+%select_safe(?M4),
  %push_magazine_release_button(?M4),
  %pull_and_hold_charging_handle(?M4),
  %push_and_hold_bolt_catch_bottom(?M4),
  %release_charging_handle(?M4),
  %release_bolt_catch_bottom(?M4),
  %push_charging_handle(?M4),
  %inspect_chamber(?M4),
  %select_safe(?M4),
  %push_bolt_catch_top(?M4),
  %select_semi(?M4),
  %pull_and_hold_trigger(?M4),
  %pull_and_hold_charging_handle(?M4),
  %release_charging_handle(?M4),
  %select_safe(?M4).
```

Now, the query `%create(M4,?m4), %clear_weapon(?m4)` will succeed, and results in changes to the KB corresponding to the actions taken (i.e., the rifle is cleared and in a safe state).

We can also test an individual action rule and examine the add- and delete-lists that are returned. For example, we can execute a query to create a rifle, then load and fire it:

```
%create(M4,?m4), %insert_magazine(?m4),
%pull_and_hold_charging_handle(?m4),
%release_charging_handle(?m4),
%do(pull_and_hold(?M4.lower.trigger),?del,?add)
```

Note that `pull_and_hold` is a functional term defined using the technique described in Sect. 4.3, to avoid having to instantiate the action. The query results in the following value for `?del` (recall that both the delete- and add-lists are lists of reified formulas):

```
[${Magazine_1 [rounds -> [Round_2, ..., Round_30]]},
 ${Round_2 [location -> Magazine_1]},
 ${Round_1 : Round}, ${Round_1 [location -> Chamber_1]},
 ${Round_1 [casing -> Casing_1]},
 ${Trigger_1 [pulled -> \false]}]
```

and `?add`:

```
[${Magazine_1 [rounds -> [Round_3, ..., Round_30]]},
 ${Round_2 [location -> Chamber_1]},
 ${Casing_1 [location -> Outside]},
 ${Trigger_1 [pulled -> \true]}]
```

(We have abbreviated the long list of rounds in the magazine here). In other words: the round in the chamber; `Round_1` is gone, its casing is in the `Outside` location (i.e., it is ejected from the rifle); the top round in the magazine, `Round_2` is removed from the magazine and now located in the chamber; and the trigger is in the pulled state.

5 Related Work

In [5], the authors develop a "semantic-enabled assessment module" for a 3D environment, and [4] introduces a semantic approach to games, in order to enable more reusability and emergent gameplay. These projects each relate to different parts of the SAVE framework, but it is not clear what kind of semantic representations they use.

The approach of describing actions with preconditions and effects has a long history, dating back to the early days of AI planning systems [7]. These planning representations are typically focused on reasoning about achieving a certain goal state by chaining together a sequence of actions. Our present work, in contrast, *executes* actions selected by a user. More importantly, planning representations are typically specialized for a given domain, and are based on a less expressive logic. The action descriptions in our work have access to a full-featured ontology language.

In [8], we created ontological descriptions of virtual environments. However, the project focused on support for reasoning about simulation fidelity as it relates to large-scale training exercises and simulations. In the current work, we are instead focused on modeling actions and objects on a detailed, individual level.

6 Conclusions and Future Work

We have developed the semantic models necessary for a semantically enhanced virtual learning environment. In a sense, these models constitute a simulation of the M4 rifle. A 3D environment is used to interact with this semantic simulation in order to perform a given procedural task. The steps taken by the student are automatically assessed and compared to the "gold standard" solution. There are several possible directions for future work.

Currently, the Exercise UI allows the user to try any action on any objects. With little to no modifications to our modeling, we could use the semantic models to show a user only the actions that are physically *possible* in a given situation, or the ones that are *allowed*, *required*, etc. This could help users better understand the environment as well as the task they are supposed to learn. In some contexts it may prove *too* helpful, by telling the student exactly what to do. For actions that are not possible or allowed, we could show explanations of *why* that is the case. This feature could be implemented using Sunflower's tracing and natural language capabilities, described in [2]. It would also be interesting to examine the use of semantics for *discovering* relevant ontologies or classes during the annotation phase. Finally, modeling a second domain would demonstrate the generalizability of our work.

Acknowledgements. This material is based upon work supported by the United States Government under Contract No. W911QY-14-C-0023. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Government. Development of the Sunflower IDE was funded in part by the U.S. Office of the Assistant Secretary of Defense for Readiness under the Open Netcentric Interoperability for Training and Testing (ONISTT) project, and by TRMC (Test Resource Management Center) T&E/S&T (Test and Evaluation/Science and Technology) Program under the NST Test Technology Area. We are also indebted to the research community for developing and maintaining the open source language and software components on which Sunflower depends, especially Flora-2 (a.k.a. Ergo Lite), XSB Prolog, and InterProlog.

References

1. Soldier's manual of common tasks - warrior skills level 1. Technical report, Headquarters Department of the Army, September 2012
2. Ford, R., Denker, G., Elenius, D., Moore, W., Abi-Lahoud, E.: Automating financial regulatory compliance using ontology+rules and Sunflower. In: Proceedings of SEMANTICS (2016). (to appear)
3. Greuel, C., Myers, K.: Assessment and content authoring in semantically enabled virtual environments. In: Proceedings of Interservice/Industry Training, Simulation and Education Conference (2016). (submitted)
4. Kessing, J., Tutenel, T., Bidarra, R.: Designing semantic game worlds. In: Proceedings of the The Third Workshop on Procedural Content Generation in Games, PCG 2012, ACM, New York, NY, USA (2012). http://doi.acm.org/10.1145/2538528.2538530
5. Maderer, J., Gütl, C., AL-Smadi, M.: Formative assessment in immersive environments: a semantic approach to automated evaluation of user behavior in open wonderland. In: Proceedings of Immersive Education (iED) Summit, June 2013
6. Myers, K., Gervasio, M.: Solution authoring via demonstration and annotation: an empirical study. In: Proceedings of International Conference on Advanced Learning Technologies (2016). (submitted)
7. Nau, D., Ghallab, M., Traverso, P.: Automated Planning: Theory & Practice. Morgan Kaufmann Publishers Inc., San Francisco (2004)
8. Riehemann, S., Elenius, D.: Ontological analysis of terrain data. In: Liao, L. (ed.) ACM International Conference Proceeding Series on COM.Geo, p. 10. ACM (2011)
9. Vujosevic, R., Ianni, J.: A taxonomy of motion models for simulation and analysis of maintenance tasks. Technical report, United States Air Force Armstrong Laboratory, January 1997