Cascading Verification: An Integrated Method for Domain-Specific Model Checking

Fokion Zervoudakis* David S. Rosenblum†
Sebastian Elbaum‡ Anthony Finkelstein*

*Department of Computer Science
University College London
London, UK
{f.zervoudakis, a.finkelstein}@cs.ucl.ac.uk

†School of Computing
National University of Singapore
Republic of Singapore
david@comp.nus.edu.sg

‡Department of Computer Science & Engineering
University of Nebraska–Lincoln
Lincoln NE, USA
elbaum@cse.unl.edu

ABSTRACT

Model checking is an established formal method for verifying the desired behavioral properties of system models. But popular model checkers tend to support low-level modeling languages that require intricate models to represent even the simplest systems. Modeling complexity arises in part from the need to encode *domain knowledge*—including domain objects and concepts, and their relationships—at relatively low levels of abstraction. We will demonstrate that, once formalized, domain knowledge can be reused to enhance the abstraction level of model and property specifications, and the effectiveness of probabilistic model checking.

This paper describes a novel method for domain-specific model checking called *cascading verification.* The method uses composite reasoning over high-level system specifications and formalized domain knowledge to synthesize *both* low-level system models and the behavioral properties that need to be verified with respect to those models. In particular, model builders use a high-level *domain-specific language* (DSL) to encode system specifications that can be analyzed with model checking. Domain knowledge is encoded in the *Web Ontology Language* (OWL), the Semantic Web Rule Language (SWRL) and Prolog, which are combined to overcome their individual limitations. Synthesized models and properties are analyzed with the probabilistic model checker PRISM. Cascading verification is illustrated with a prototype system that verifies the correctness of *uninhabited aerial vehicle* (UAV) mission plans. An evaluation of this prototype reveals non-trivial reductions in the size and complexity of input system specifications compared to the artifacts synthesized for PRISM.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Model checking; D.2.13 [**Reusable Software**]: Domain engineering

General Terms

Languages, Verification

Keywords

Composite reasoning, domain model, model checking, OWL, PRISM, Prolog, SWRL, UAVs

1. INTRODUCTION

Model checking is an established formal verification method whereby a model checker systematically explores the state space of a system model to verify that each state satisfies a set of desired behavioral properties [1].

Research in model checking has focused on enhancing the efficiency and scalability of verification by employing partial order reduction, and by exploiting symmetries and other state space properties. This research is important because it mitigates the complexity of model checking algorithms, thereby enabling model builders to verify larger, more elaborate models. But the complexity associated with model and property specification has yet to be sufficiently addressed. Popular model checkers tend to support low-level modeling languages that require intricate models to represent even the simplest systems. For example, PROMELA, the modeling language used by the model checker Spin, is essentially a dialect of the relatively low-level programming language C. Due to lack of appropriate control structures, the modeling language used by the probabilistic model checker PRISM forces model builders to pollute model components with variables that act as counters. These variables are manipulated at runtime to achieve desirable control flow from otherwise unordered commands.

Modeling complexity arises in part from the need to encode *domain knowledge*—including domain objects and concepts, and their relationships—at relatively low levels of abstraction. We will demonstrate that, once formalized, domain knowledge can be reused to enhance the abstraction level of model and property specifications, and the effectiveness of probabilistic model checking.

Leveraged appropriately, formal domain knowledge can decrease specification and verification costs. On the verification side, the model checking framework Bogor achieves significant state space reductions in model checking of program code by exploiting characteristics of the program code's deployment platform [2]. On the specification side, *semantic model checking* supplements model checking with semantic reasoning over domain knowledge encoded in the *Web Ontology Language* (OWL). Semantic model checking has been used to verify Web services [3, 4]; Web service security requirements [5]; probabilistic Web services [6]; Web service interaction protocols [7]; and Web service flow [8]. Additionally, multi-agent model checking has been used to verify OWL-S process models [9].

OWL is a powerful knowledge representation formalism, but expressive and reasoning limitations constrain its utility in the context of semantic model checking; for example, OWL cannot reason about triangular or self-referential relationships. As a W3C-approved OWL extension, the Semantic Web Rule Language (SWRL) addresses some of these limitations by integrating OWL with Horn-like rules. But OWL+SWRL cannot reason effectively with negation. The *logic programming* (LP) language Prolog can be used to overcome problems that are intractable in OWL+SWRL. Prolog, however, lacks several of the expressive features afforded by OWL including support for equivalence and disjointness.

This paper describes a novel method for domain-specific model checking called *cascading verification.* Our method uses composite reasoning over high-level system specifications and formalized domain knowledge to synthesize *both* low-level system models and the behavioral properties that need to be verified with respect to those models. In particular, model builders use a high-level *domain-specific language* (DSL) to encode system specifications that can be analyzed with model checking. A *compiler* uses automated reasoning to verify the consistency between each specification and domain knowledge encoded in OWL+SWRL and Prolog, which are combined to overcome their individual limitations. If consistency is deduced, then explicit and inferred domain knowledge is used by the compiler to synthesize a *discrete-time Markov chain* (DTMC) model and *probabilistic computation tree logic* (PCTL) properties from template code. PRISM subsequently verifies the model against the properties. Thus, verification *cascades* through several stages of reasoning and analysis.

Our method gains significant functionality from each of its constituent technologies. OWL supports expressive knowledge representation and efficient reasoning; SWRL extends OWL with Horn-like rules that can model complex relational structures and self-referential relationships; Prolog extends OWL+SWRL with the ability to reason effectively with negation; DTMC introduces the ability to formalize probabilistic behavior; and PCTL supports the elegant expression of probabilistic properties.

Cascading verification is illustrated with a prototype system that verifies the correctness of *uninhabited aerial vehicle* (UAV) missions. We use the prototype to analyze 58 mission plans, which are based on real-world mission scenarios developed independently by the Defense Advanced Research Projects Agency (DARPA) [10] and the Defence Research and Development Canada (DRDC) agency [11]. UAVs are contextualized by a particularly interesting and important experimental domain. The stochastic nature of UAV missions led us to select *probabilistic model checking*, and in particular the popular tool PRISM [12], for the verification of UAV mission plans.

This paper presents three research contributions. First, we describe a novel model checking method that leverages domain knowledge to realize a non-trivial reduction in the effort required to specify system models and behavioral properties. For example, from 23 lines of YAML code comprising 92 tokens, cascading verification synthesizes 104 lines of PRISM code comprising 744 tokens and three behavioral properties (with our prototype, model builders encode mission specifications in a domain-specific dialect of the human-readable YAML format [13]).

Second, we describe a composite inference mechanism that supports the synthesis of system models and their desired behavioral properties for probabilistic model checking. Third, we present a prototype system that uses our method to verify UAV mission plans, thereby demonstrating the utility of cascading verification in the context of a significant application domain.

The remainder of this paper is structured as follows. Section 2 presents background material on the technologies that constitute cascading verification, and on the UAV domain. Section 3 presents an overview of cascading verification, and an example UAV mission that underpins subsequent discussion and examples. Section 4 describes how domain knowledge is encoded in OWL+SWRL, Prolog, and DTMC and PCTL templates. Our prototype implementation of cascading verification is described in Section 5. An evaluation, which considers the portability of our method, and related work are presented in Section 6 and Section 7, respectively. Section 8 concludes the paper and discusses future work.

2. BACKGROUND

The research problem outlined in the previous section cannot be addressed with a single technology. Our solution was to develop a method that integrates OWL+SWRL, Prolog and DTMC and PCTL. OWL was chosen because it is an established knowledge representation formalism, and the ontology specification language recommended by the W3C [14]. OWL limitations motivate several contending extensions including SWRL, CARIN, $\mathcal{AL}$-log, DL-safe rules, $\mathcal{DL}$+log, and many others [15]. Hybrid knowledge representation systems that integrate OWL+SWRL and Prolog have also been proposed [16, 17, 18, 19, 20, 21, 22]. In the interest of space, we cannot offer a comprehensive review of all these formalisms. We chose to address OWL limitations with SWRL and Prolog; the former is an OWL extension approved by the W3C, while the latter is one of the most prominent logic-based knowledge representation languages.

Probabilistic model checking is supported by various software tools including ProbVerus and FMurφ, which analyze DTMC models; ETMCC and MRMC (the successor of ETMCC), which analyze DTMC and CTMC models; and LiQuor and Rapture, which analyze *Markov decision process* (MDP) models [1]. But PRISM is, in our opinion, preferable because it supports both model types, thereby extending the potential of our method and prototype. PRISM also supports PCTL, a formalism that can express a large class of properties in an elegant manner.

We proceed to elaborate the technologies that constitute cascading verification.

2.1 OWL+SWRL and Prolog

Description logics (DLs) are a family of knowledge representation languages based on *first-order logic* (FOL) that can be used to construct logically valid knowledge bases. DLs describe a domain in terms of *concepts* or *classes* (specified as axioms in a TBox), *individuals* (specified as assertions in an ABox) and *properties* or *roles* [14]. DL concepts and individuals are roughly comparable to classes and objects, respectively, in object-oriented programming, while roles are comparable to UML associations. DLs support inferences that deduce the logical implications of ontological axioms with respect to concept *satisfiability*, *subsumption*, *equivalence* and *disjointness* [23].

OWL is an ontology specification language based on the modern DL $\mathcal{SHOIN}(\mathbf{D})$ [14]. The OWL language structures, and thereby supports the automated processing of, formalized knowledge. But OWL is constrained by the expressive and reasoning limitations inherent in $\mathcal{SHOIN}(\mathbf{D})$. For example, OWL can be used to model object relations that form tree-like patterns, but not the triangular relationship that exists between a child, the child's father, and the father's brother [15]; nor the self-referential relationship that references an individual to itself [24]. SWRL addresses some of these limitations by extending OWL with Horn-like rules [25]. But OWL+SWRL cannot reason effectively with negation [15, 26]. Problems that are intractable in OWL+SWRL can be addressed with the programming language Prolog [15, 27].

Prolog is based on a FOL subset, which is expressed with first-order Horn clauses comprising facts, queries and rules. Unlike OWL+SWRL, Prolog can reason effectively with negation. But Prolog is not without limitations: OWL can be translated into formulas of a general FOL subset, but this subset overlaps only partially with the FOL subset underpinning Prolog. Consequently, some OWL primitives cannot be expressed efficiently in Prolog. For example, Prolog does not provide an equivalence predicate; Prolog's native syntax cannot encode the OWL primitives `disjointWith` and `differentIndividualFrom`, which denote concept and individual disjointness, respectively; and Prolog cannot encode the OWL primitive `oneOf`, which defines a concept by enumerating all individuals belonging to that concept.

This paper does not propose a definitive or optimal OWL-LP integration framework. Nor do we attempt to take sides in the ongoing debate regarding OWL-LP integration [15]. Our exclusive objective is to support domain-specific probabilistic model checking by combining well established logic systems [15, 28]. We propose to achieve this objective via *loose* (rather than *full*) integration, whereby DL and LP components are connected through a minimal interface [29].

2.2 Probabilistic Model Checking

Probabilistic model checking is a method for verifying the behavioral properties of systems affected by stochastic processes [1, 30]. In order to model these processes, which may include message delays, failure rates and other phenomena, finite-state transition systems are enriched with probabilities. *Markov chains* are transition systems where the successor of each state is chosen probabilistically and independently of preceding events (i.e., Markov chains are memoryless). DTMCs are Markov chains that represent time in discrete time-steps. A DTMC can be formalized as a tuple $\mathcal{M} = (S, \mathbf{P}, \iota_{init}, AP, L)$ where S is a countable set of states; $\mathbf{P} : S \times S \to [0,1]$ is the transition probability function, such that for all states s: $\sum_{s' \in S} \mathbf{P}(s, s') = 1$; $\iota_{init} : S \to [0,1]$ is the initial state distribution, such that $\sum_{s' \in S} \iota_{init}(s) = 1$; and $L : S \to 2^{AP}$ is a labeling function that maps each state to a subset of AP, a set of atomic propositions that abstract key characteristics from modeled systems.

Probabilistic model checking can be used to verify both quantitative and qualitative DTMC properties. The former constrain probabilities to specific thresholds, while the latter associate desirable and undesirable behavior with probabilities of one and zero, respectively. PCTL is an extension of the branching-time *computation tree logic* (CTL), and a prominent formalism for expressing probabilistic properties. PCTL supports the probabilistic operator $\mathbb{P}_J(\varphi)$, where φ specifies a constraint over the set of paths that constitute a Markov chain, and J specifies a closed interval between one and zero that bounds the probability of satisfying φ.

2.3 UAV Missions

UAVs, or *uninhabited aerial systems* (UASs), are aircraft capable of either autonomous or remote controlled flight. Primarily oriented toward (dull, dirty, or dangerous) military missions, UAVs are increasingly relied upon to perform agricultural, scientific, industrial and law-enforcement tasks over civilian airspace [31, 32, 33].

The UAV domain exhibits complexity at different levels of granularity. UAVs incorporate sophisticated payloads, multiple sensors and increasing computational power. These capabilities could, in time, enable UAV swarms to execute complex multi-task missions with reduced human supervision [34]. For any given mission, autonomous UAVs may be required to execute tasks synchronously and in real-time; with local, incomplete and/or noisy knowledge; and in the context of a dynamic environment [35]. These factors combine to form a complex stochastic state space that motivates the probabilistic verification of UAV mission plans.

3. METHOD OVERVIEW

Figure 1 illustrates a high-level, domain-agnostic schematic of our method and prototype. Domain experts, who are the method's primary stakeholders, use OWL to define domain concepts and their relationships; SWRL and Prolog to define rules; and PRISM's modeling and property specification languages to define, respectively, DTMC and PCTL templates. Model builders, who are also primary stakeholders, use a high-level DSL to encode system specifications for model checking. We note that domain knowledge is formalized once and subsequently reused to support the verification of multiple system specifications.

3.1 An Example Mission

With our prototype, model builders use a domain-specific YAML dialect to encode mission plans comprising UAV *assets* and the *action workflows* assigned to those assets. The YAML code in Listing 1 specifies Mission A, an example mission that is representative of the 58 mission plans developed for this project.

```
Action:
  TraversePathSegmentAction:
```

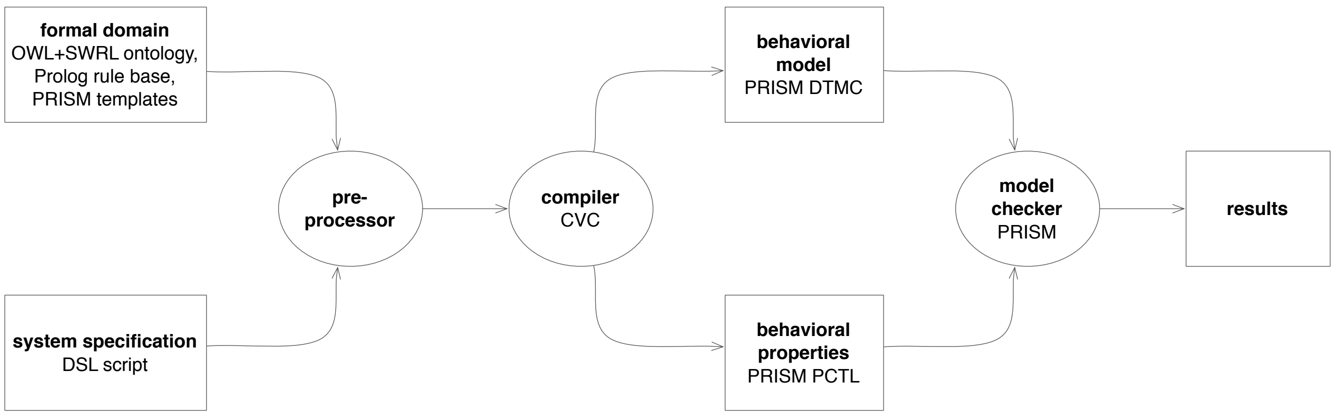


Figure 1: A high-level, domain-agnostic schematic of our method and prototype. Rectangular and oval shapes represent data and processes, respectively; and bold and normal text distinguishes method from prototype, respectively.

```
    - id: TPSA1                                   3
      duration: 60                                4
      coordinates: [-118.27017, 34.04572,         5
        -118.27279, 34.04284]                     6
    - id: TPSA2                                   7
      duration: 60                                8
      coordinates: [-118.2739, 34.03928]          9
      preconditions: [TPSA1, TPSA3]              10
    - id: TPSA3                                  11
      duration: 60                               12
      coordinates: [-118.26482, 34.03332,        13
        -118.27383, 34.03824]                    14
    - id: TPSA4                                  15
      duration: 60                               16
      coordinates: [-118.28204, 34.0376]         17
      preconditions: [TPSA3]                     18
  PhotoSurveillanceAction:                       19
    - id: PSA5                                   20
      duration: 50                               21
      preconditions: [TPSA3]                     22
Asset:                                           23
  Hummingbird:                                   24
    - id: H1                                     25
      actions: [TPSA1, TPSA2]                    26
    - id: H2                                     27
      actions: [TPSA3, TPSA4, PSA5]              28
```

Listing 1: YAML code for Mission A

Mission A comprises two *Hummingbird* assets (lines 24–28 in Listing 1); a single *photo surveillance action* (lines 19–22), which is a type of *sensor action*; and four *path segment traversal actions* (lines 2–18), which are *kinetic actions*. A path segment traversal action instructs the executing UAV to traverse a path between two waypoints. For each such action, the latitudes and longitudes of the delineating waypoints are stored in an array and indexed in succession; in other words, the latitude of waypoint one is followed by the longitude of waypoint one, which is in turn followed by the latitude of waypoint two, etc. We note that the end coordinates of an action a constitute the start coordinates of an action b if a precedes b, and both a and b are assigned to the same asset; for example, the end coordinates of action `TPSA1` (line 6) constitute the implied start coordinates of action `TPSA2`, which succeeds `TPSA1` in the sequence of kinetic actions assigned to asset `H1` (line 26).

The mission concepts described above will be elaborated as we proceed to analyze our method and prototype.

3.2 From Specification to Verification

For any given mission specification, a *cascading verification compiler* (CVC) synthesizes both the DTMC and PCTL artifacts corresponding to that specification. Artifacts are synthesized as follows:

1. Mission specifications encoded in YAML are transformed by the CVC into ABox assertions. During this preprocessing phase, the CVC uses geographic coordinates from mission specifications, and data (pertaining to operational environments) from external sources, to perform geodetic calculations.[1] The equations that support these calculations are hard-coded in the CVC; for example, the compiler comprises geodesic equations that establish the occurrence, and calculate the duration, of threat area incursions committed by UAVs. Geographic information resulting from preprocessing is integrated with the generated ABox.

2. We use Pellet, a sound and complete semantic reasoner [36], to verify the generated ABox against the TBox defined by domain experts. In doing so, the reasoner ensures that mission constructs encoded in YAML are consistent with OWL+SWRL axioms. Inconsistencies between TBox and ABox signify an invalid mission specification, which causes the compilation process to terminate with an error. If consistency is deduced, then the reasoner proceeds to generate inferences from explicitly encoded domain knowledge; for example, if geodetic calculations establish the occurrence of a threat area incursion, then the asset committing that incursion is inferred to be a *threatened asset*.

3. Inferred ontological knowledge is transformed by the CVC into Prolog facts. The compiler for SWI-Prolog—an open source implementation of Prolog [37]—inputs

[1]Geodetics is a branch of applied mathematics that deals with the size and shape of the Earth.

the generated fact-base, and the Prolog rule-base defined by domain experts, and proceeds to generate inferences; for example, the last kinetic action in an action workflow is inferred to be a *default terminal action*. The CVC uses Prolog inferences, in conjunction with explicit and inferred ontological knowledge, to synthesize DTMC and PCTL artifacts from predefined templates.

PRISM inputs the synthesized artifacts, verifies the system model against its desired behavioral properties, and returns logical and probabilistic results from the verification. If the results are deemed acceptable by the model builder(s), then the mission can be scheduled for real-world execution (via some process that is outside the scope of our method).

4. DOMAIN MODELING

This section describes the process of encoding of domain knowledge in OWL+SWRL, Prolog, and DTMC and PCTL templates. The application of these technologies is illustrated with respect to the UAV domain and, in particular, Mission A, the example mission presented in the previous section.

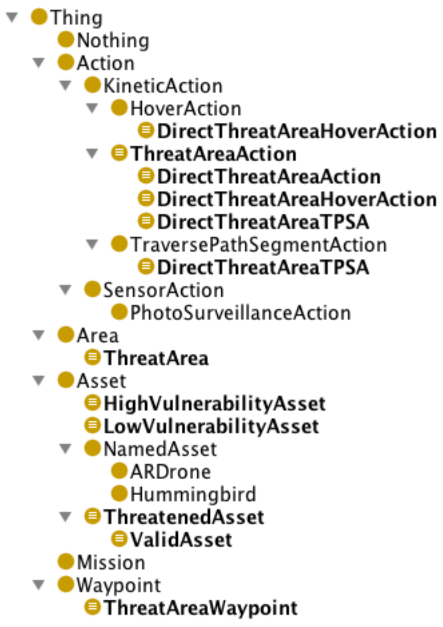
4.1 Semantic Modeling

With cascading verification, domain experts use ontologies to formally define domain concepts and their relationships. For our prototype, we have developed a *complex missions ontology* (CEMO) that formalizes a subset of the UAV domain. Figure 2 illustrates CEMO's class hierarchy, where each OWL class represents a grouping of individuals with similar characteristics. Five classes—`Action`, `Area`, `Asset`, `Mission` and `Waypoint`—inherit directly from the built-in OWL class `Thing`, which represents the set of all individuals. (The built-in OWL class `Nothing` represents the empty set.)

Having defined the concept of an asset we specify that every `Asset` individual must be associated with an `Action` individual via the *object property* `hasAction`; in other words, every asset must execute at least one action. `Asset` individuals are also associated with *data properties* that describe asset cost, endurance and speed. Class `Asset` is extended by classes `ARDrone` and `Hummingbird`. The latter classes represent quadcopter UAVs that have informed our research. Classes `Action`, `Asset` and `Hummingbird` are comprised in Mission A (lines 1, 23 and 24, respectively, in Listing 1).

During a mission, assets execute actions that may be associated with other actions via *preconditions*. An action *a* is a precondition to an action *b* if the end of *a* must precede the beginning of *b* in the sequence of actions that constitute an action workflow. Mission A comprises three preconditions (lines 10, 18 and 22 in Listing 1), where each precondition relates actions assigned to the same asset. A fourth precondition (line 10) associates action `TPSA2` with action `TPSA3`, thereby coupling the behavior of the assets to which those actions are assigned (`H1` and `H2`, respectively).

Class `Action` is extended by classes `KineticAction`, whose members are associated with a data property describing action duration, and `SensorAction`. Class `KineticAction` is in turn extended by classes `HoverAction` and `TraversePathSegmentAction`. The OWL code in Listing 2 uses *Manchester OWL syntax*—a human-friendly ontology representation language [38]—to formally define class `KineticAction`.



- Thing
  - Nothing
  - Action
    - KineticAction
      - HoverAction
        - **DirectThreatAreaHoverAction**
      - **ThreatAreaAction**
        - **DirectThreatAreaAction**
        - **DirectThreatAreaHoverAction**
        - **DirectThreatAreaTPSA**
      - TraversePathSegmentAction
        - **DirectThreatAreaTPSA**
    - SensorAction
      - PhotoSurveillanceAction
  - Area
    - **ThreatArea**
  - Asset
    - **HighVulnerabilityAsset**
    - **LowVulnerabilityAsset**
    - NamedAsset
      - ARDrone
      - Hummingbird
    - **ThreatenedAsset**
      - **ValidAsset**
  - Mission
  - Waypoint
    - **ThreatAreaWaypoint**

Figure 2: CEMO's class hierarchy as presented by the Protégé ontology editor.

```
Class: KineticAction                               1
  SubClassOf: Action                               2
    and hasDurationInSeconds some int              3
    and hasPrecondition only Action                4
    ...                                            5
```

Listing 2: OWL code for class KineticAction

Listing 2 contains the keywords *some* (line 3) and *only* (line 4), which represent, respectively, existential and universal restrictions in OWL. Existential restrictions describe classes of individuals that must participate in at least one relationship, along a specified property, with individuals that are members of a specified class [39]. Universal restrictions describe classes of individuals that may, and can only, participate in relationships along a specified property with individuals that are members of a specified class.

Assets may be required to commit threat area incursions, thereby compelling mission developers to consider the impact of asset *survivability* on the probability of mission success. To accommodate this requirement, CEMO comprises classes that describe *tactical* missions (Figure 2 highlights these classes in bold). Used exclusively to support automated reasoning, tactical concepts are not available to mission developers via the YAML DSL. We note that the asset subtypes `HighVulnerabilityAsset` and `LowVulnerabilityAsset` are associated with a data property describing asset vulnerability. The specific value assigned to each subclass is used by the CVC to calculate probabilities that are ultimately integrated into DTMC models (the integration process will be elaborated in Section 4.3).

4.2 Rule-Based Modeling

OWL is a powerful knowledge representation formalism, but expressive and reasoning limitations constrain its utility; for example, OWL cannot model *cross-cutting actions*. We consider an action a to be cross-cutting if a is a precondition to an action b, and b is assigned to an asset that is different from the asset to which a is assigned. By this definition, action `TPSA3` in Mission A is a cross-cutting kinetic action. The OWL code in Listing 3 presents an incomplete definition of class `CrossCuttingKineticAction`.

```
Class: CrossCuttingKineticAction                    1
  EquivalentTo: KineticAction                       2
    and (isActionOf some Asset)                     3
    and (hasPrecondition some                       4
      (Action                                       5
        and (isActionOf some Asset)))               6
```

Listing 3: OWL code for class CrossCuttingKineticAction

The definition in Listing 3 is lacking because the asset in line 3 cannot be differentiated from the asset in line 6. The SWRL code in Listing 4 uses the built-in OWL property `differentFrom` to appropriately define class `CrossCuttingKineticAction` (question mark prefixes denote variables).

```
Rule:                                               1
    KineticAction(?a), KineticAction(?b),           2
    Asset(?x), Asset(?y),                           3
    hasAction(?x, ?a), hasAction(?y, ?b),           4
    hasPrecondition(?b, ?a),                        5
    DifferentFrom(?x, ?y)                           6
  -> CrossCuttingKineticAction(?a)                  7
```

Listing 4: SWRL code for rule CrossCuttingKineticAction

SWRL rules extend the expressive power of OWL; but like OWL, SWRL cannot reason effectively with negation. Rules that must reason with negation are therefore encoded in Prolog. Domain knowledge is negated during the ontology development process if Pellet reasoning with respect to that knowledge is deemed unattainable. The Prolog code in Listing 5 formally defines rule `terminal`, which comprises three negated atoms (an atom is the basic building block of a Prolog rule). Rule `terminal` encapsulates both explicit and inferred ontological knowledge; explicit knowledge is encoded with the atom `has_action` (line 2), while the three negated atoms (lines 3–5) encode knowledge inferred by Pellet.

```
terminal(X) :-                                      1
    has_action(A, X),                               2
    not(is_precondition_to(X, _)),                  3
    not(single_action_asset(A)),                    4
    not(zero_action_asset(A)).                      5
```

Listing 5: Prolog code for rule terminal

The transition of inferred knowledge from OWL to Prolog can be illustrated with the atom `is_precondition_to` (line 3 in Listing 5). The OWL code in Listing 6 formally defines the object property `isPreconditionTo`, a relationship that inverts the object property `hasPrecondition`. This inversion constitutes a Pellet inference.

```
ObjectProperty: isPreconditionTo                    1
  InverseOf: hasPrecondition                        2
  ...                                               3
```

Listing 6: OWL code for the object property isPreconditionTo

Mission A comprises several instances of the DSL construct `preconditions` (lines 10, 18 and 22 in Listing 1). The CVC transforms knowledge contained in this construct into knowledge encoded as property `hasPrecondition`. Knowledge inferred by Pellet with respect to `hasPrecondition`, via the inverted property `isPreconditionTo`, is transformed by the CVC into knowledge encoded with the atom `is_precondition_to`, which in turn supports SWI-Prolog inferences.

4.3 Behavioral Modeling

OWL+SWRL and Prolog are appropriate formalisms for encoding semantic and rule-based knowledge. Likewise, the state-based PRISM modeling language is an appropriate formalism for encoding behavioral knowledge as DTMC models. The PRISM language can thus form the basis for reusable DTMC templates.

The code in Listing 7 uses string interpolation—denoted by code snippet `#{ ... }`—to encapsulate parameters in the context of a DTMC template for asset modules (the *module* is a fundamental PRISM language construct). The asset module template encodes knowledge that describes the behavior of the `Asset` concept encoded in CEMO.

```
module #{asset.class}#{asset.id}                    1
  e#{asset.id} : [0..#{asset.endurance}]            2
      init #{asset.endurance};                      3
  FOR action IN asset.kinetic_actions               4
  [#{action.type}]                                  5
      e#{asset.id}>0 &                              6
      d#{action.id}>0                               7
      -> (e#{asset.id}'=e#{asset.id}-1);            8
  END                                               9
  [#{asset.last_action.type}]                       10
      e#{asset.id}=0 |                              11
      d#{asset.last_action.id}=0                    12
      -> true;                                      13
endmodule                                           14
```

Listing 7: DTMC template for asset modules

Arguments for the parameters in Listing 7 are derived from explicit and inferred domain knowledge. Specifically, arguments for `asset.class` (line 1) and `asset.id` (lines 1, 2, 5, 7, and 9) are derived from mission plans; arguments for `asset.endurance` (lines 2 and 3) are derived from explicit domain knowledge encoded in CEMO; and arguments for `action.type` (line 5), `asset.last_action.type` (line 9) and `asset.last_action.id` (line 10) are derived from knowledge inferred by Pellet and the Prolog compiler (via a process that will be described in Section 5).

A module definition contains *variables* and *commands*. Asset module states are stored in variable `e#{asset.id}` (lines 2, 5, 7 and 9 in Listing 7), which represents asset endurance. Asset behavior is formalized with two or more commands, where each command assumes the form `[action] guard -> update`. A command becomes *enabled* for execution when its *guard* is satisfied by a specific model state. Commands encompass one or more updates, where each *update* transitions a module, with a given probability, from one state to the next. A probability of one is assumed, and can therefore be omitted, for commands with single updates. Each command may be labeled with an *action*, which forces two or more modules to transition states simultaneously (i.e., to synchronize).

Lines 4 and 8 in Listing 7 use meta-code, which is highlighted with purple text, to define a for loop. Each asset module command generated by this loop represents the execution of a kinetic action. The `true` keyword in the last command (line 11) denotes the end of execution for a specific module. This type of command prevents deadlocks that are inconsequential to mission correctness from terminating the verification process with an error.

Asset module commands comprise single updates that omit probabilities. The code in Listing 8 formally defines a template for asset survivability modules, which contain commands with multiple updates (lines 12–15). The probability of execution for each update is derived from `asset.vulnerability`; arguments for this parameter are derived via inferences from explicit domain knowledge encoded in CEMO (as described in Section 4.1). Current vulnerability values are arbitrary, and should eventually be calculated from real-world data related to asset capabilities, terrain types and weather conditions.

```
FOR action IN asset.threat_area_actions                    1
formula actn#{action.id}_tai =                             2
    d#{action.id}<=start#{action.id} &                     3
    d#{action.id}>finish#{action.id};                      4
END                                                        5
                                                           6
module #{asset.class}#{asset.id}_Survivability             7
  a#{asset.id}d : bool init false;                         8
  FOR action IN asset.threat_area_actions                  9
  [#{action.type}] !a#{asset.id}d &                       10
      actn#{action.id}_tai                                11
      -> #{1-asset.vulnerability}                         12
          :(a#{asset.id}d'=false) +                       13
         #{asset.vulnerability}                           14
          :(a#{asset.id}d'=true);                         15
  [#{action.type}] a#{asset.id}d |                        16
      !actn#{action.id}_tai                               17
      -> true;                                            18
  END                                                     19
endmodule                                                 20
```

Listing 8: DTMC template for asset survivability

Each survivability module enables PRISM to calculate the probability of survival for a threatened asset that has also been inferred by Pellet to be a *valid asset*. The concept of a threatened asset was described in Section 3; a valid asset is a threatened asset that executes at least one sensor action during a threat area incursion, thereby *justifying* the risk assumed during that incursion.

The probability of survival for a valid asset is calculated with respect to *threat area actions*, which are kinetic actions executed by the asset during threat area incursions. Lines 1 and 5 in Listing 8 use meta-code to define a for loop that generates one PRISM formula with identifier `actn#{action.id}_tai` per each action assigned to the asset in the loop header. Each formula specifies an overlap between the prosecution of a threat area incursion and the execution of a threat area action with identifier `action.id`. This overlap is delineated by variables `start#{action.id}` and `finish#{action.id}` (lines 3 and 4, respectively), which are assigned geographic information calculated by the CVC during preprocessing (as described in Section 3).

The formula identifier `actn#{action.id}_tai` in Listing 8 (defined in line 2 and used in lines 11 and 17) represents the execution of a kinetic action during the prosecution of a threat area incursion. The boolean variable `a#{asset.id}d` (defined in line 8 and used in lines 10, 13, 15 and 16) represents the destruction of the asset with identifier `asset.id`. The constructs `actn#{action.id}_tai` and `a#{asset.id}d` combine to form logical expressions, including a logical conjunction and disjunction (lines 10–11 and 16–17, respectively), that constitute the guards in the asset survivability module. When these expressions are considered in union, their logical truth values clarify asset behavior during threat area incursions. Specifically, a valid asset exists in one of the following disjoint states: 1) *not* destroyed and prosecuting a threat area incursion (the logical conjunction); 2) destroyed, as a consequence of prosecuting a threat area incursion, or *not* prosecuting a threat area incursion (the logical disjunction).

Mission properties are encoded in PRISM's property specification language, which subsumes several probabilistic temporal logics including PCTL, CSL and LTL. The code snippet `P=? [ F d#{action.id}=0 ... ]` formally defines a mission property template. This generic property queries the probability that an action with identifier `action.id` will deplete its duration—as denoted by the assertion `d#{action.id}=0`—and thereby complete its execution. An ellipsis indicates the potential for synthesized properties to comprise multiple assertions. Output from PRISM templates will be presented in the following section.

5. CASCADING VERIFICATION

Cascading verification combines the technologies described in the previous section to enhance the abstraction level of model and property specifications, and the effectiveness of probabilistic model checking; our prototype implementation of cascading verification combines these technologies to support the probabilistic verification of UAV mission plans. This section describes the prototype by tracing verification from high-level mission specifications to probabilistic model checking with PRISM.

5.1 Verification with Semantic Reasoning

With cascading verification, model builders use a DSL to encode high-level system specifications. In the context of our prototype, model builders use YAML to encode mission plans and the geographic data associated with those plans. Mission specifications encoded in YAML are transformed by the CVC into ABox assertions. During this preprocessing phase, geodesic equations use geographic coordinates (described in Section 3.1) and threat area data from external sources to calculate supplementary geographic information. For example, if the boundary of a threat area is intersected by a flightpath, then the traverse path segment action corresponding to that flightpath is classified as a threat area action. Geodesic equations can be used to establish threat area incursions in this manner because both flightpath and threat area boundary are defined by *great circles*.[2]

Pellet verifies the generated ABox against the TBox defined by domain experts. Inconsistencies between TBox and ABox signify an invalid mission specification; for example, an asset that does not execute at least one kinetic action would be inconsistent with respect to the definition of class `Asset` presented in Section 4.1. The OWL code in Listing 9 specifies a valid ABox individual with identifier `H1`, which corresponds to the identifier of the Hummingbird asset specified in Mission A.

[2]A great circle is the intersection of the Earth's surface with a plane passing through the center of the Earth.

```
Individual: H1
  Types: Hummingbird
  Facts: hasAction TPSA1 and hasAction TPSA2
```

Listing 9: OWL code for the individual H1

With ABox consistency deduced, Pellet proceeds to generate inferences by, for example, reasoning about *realization*, which determines the direct types of each individual. Given realization, a threat area action that initiates or prolongs an incursion is classified by Pellet as a *direct threat area action* (DTAA). This inference triggers the synthesis of PRISM code that calculates a *risk acceptability factor* (RAF) for the threat area incursion comprising the inferred DTAA. Each RAF value quantifies the risk for a specific threat area incursion, with RAF values of zero and one indicating high- and low-risk incursions, respectively. A RAF numerator represents the duration of concurrent execution between sensor actions and DTAAs during a threat area incursion; the denominator represents the aggregate duration of DTAA executions during that incursion.

RAF values resulting from the execution of synthesized PRISM code are verified against a RAF threshold value, which is specified in CEMO and integrated by the CVC into synthesized mission properties. For example, property `P=? [ F raf>0.6 ]` queries the probability that the specified RAF value exceeds 60 percent. Current threshold values are arbitrary, and should eventually be calculated from real-world data.

5.2 Classification with Prolog

Some Pellet inferences are transformed by the CVC into Prolog rules; for example, the Prolog rule `terminal` comprises knowledge inferred by Pellet (as described in Section 4.2). The Prolog code in Listing 10 formally defines rule `terminal_constrained_observer`, which encapsulates the atom `terminal`.

```
terminal_constrained_observer(X) :-
  constrained_observer(X),
  terminal(X).
```

Listing 10: Prolog code for rule terminal_constrained_observer

SWI-Prolog classifies each kinetic action with respect to the relationships that exist between it and other kinetic actions; for example, an action that is the last kinetic action to be executed by an asset, and has as precondition at least one cross-cutting kinetic action, is classified by SWI-Prolog as a `terminal_constrained_observer`. The classification of a kinetic action affects the composition of the PRISM module for the asset to which that action is assigned; for example, the classification of action `TPSA2` in Mission A as a `terminal_constrained_observer` affects the PRISM module corresponding to asset `H1`. Affected asset module constructs include the action label of the command associated with `TPSA2` (recall that each asset module command is associated with a specific kinetic action) and, potentially, the action label of the last module command (see inferred arguments in Section 4.3).

Kinetic action classifications also generate mission properties; for example, as the last kinetic action to be executed by an asset, the successful execution of a `terminal_constrained_observer` is a desired mission property because it guarantees the successful execution of all preceding actions.

5.3 Synthesized Models and Properties

The CVC uses explicit and inferred domain knowledge to synthesize DTMC and PCTL artifacts from predefined templates; for example, the asset module template presented in Section 4.3 underpins the synthesis of one module per each asset in Mission A. The PRISM code in Listing 11 specifies a synthesized asset module corresponding to asset `H2`, where the commands in lines 3 and 4 represent, respectively, the execution of actions `TPSA3` and `TPSA4` assigned to `H2`.

```
module Hummingbird2                              1
  e2 : [0..120] init 120;                        2
  [asst1] e2>0 & d3>0 -> (e2'=e2-1);             3
  [asst1] e2>0 & d4>0 -> (e2'=e2-1);             4
  [asst1] e2=0 | d4=0 -> true;                   5
endmodule                                        6
```

Listing 11: PRISM asset module code

Let us assume a threat area action classification for action `TPSA4` (via the preprocessing phase described in Section 5.1). Given this assumption, Pellet infers action `TPSA4` to be a DTAA (since `TPSA4` initiates the incursion). Asset `H2` is consequently inferred a valid asset (since `H2` executes at least one sensor action, namely `PSA5`, during the incursion). As a consequence of these inferences, the CVC synthesizes an asset survivability module to calculate the probability of survival for `H2` (as described in Section 4.3). The PRISM code in Listing 12 specifies the synthesized survivability module corresponding to asset `H2`, where variable `a2d` (lines 4–7) represents the asset's destruction.

```
formula actn4_tai = d4>0 & d4<=60;                       1
                                                         2
module Hummingbird2_Survivability                        3
  a2d : bool init false;                                 4
  [asst1] !a2d &  actn4_tai                              5
      -> 0.99:(a2d'=false) + 0.01:(a2d'=true);           6
  [asst1]  a2d | !actn4_tai                              7
      -> true;                                           8
endmodule                                                9
```

Listing 12: PRISM asset survivability code

At the conclusion of the synthesis process, the DTMC artifact that models Mission A is verified against the synthesized PCTL property specified in Listing 13. This property comprises variables `d2` and `d4`, which represent the durations of `TPSA2` and `TPSA4`, respectively; variable `a2d`, described above; and variable `raf2`, which represents the RAF value for the threat area incursion committed by asset `H2`. The property need not comprise variables to represent the durations of `TPSA1` and `TPSA3` because these actions precede `TPSA2` and `TPSA4`, respectively. In other words, the successful execution of `TPSA1` is implied by the successful execution of `TPSA2` (this implication is denoted by the classification of `TPSA2` as a `terminal_constrained_observer`), etc.

```
P=? [ F d2=0 & d4=0 & !a2d & raf2>0.6 ]
```

Listing 13: PRISM mission property code

Given the property in Listing 13, the probability of success for Mission A is calculated by PRISM to be approximately 0.299. In particular, the verification of Mission A assigns variables `d2` and `d4` with values of zero; variable `!a2d` is assigned a probability of approximately 0.299; and variable `raf2` is assigned a value of approximately 0.833.

5.4 Implementation

We have implemented several of the components that constitute our prototype. The ontology described in Section 4.1 and Section 4.2 was implemented in OWL+SWRL. The Prolog rule-base described in Section 4.2 was implemented in SWI-Prolog. The DTMC and PCTL templates presented in Section 4.3 were implemented in the programming language Ruby. And the DSL described in Section 3 was implemented in YAML, which is particularly compatible with Ruby. While integration, testing and extension of implemented components is currently ongoing, completed work was sufficient to support an evaluation, which is presented in the following section.

6. EVALUATION

We assert that by enhancing the abstraction level of model and property specifications, cascading verification also enhances the effectiveness of probabilistic model checking. To validate this assertion, we will demonstrate that, as an implementation of cascading verification for the UAV domain, the prototype presented in this paper benefits mission developers by simplifying the verification of UAV mission plans, and by augmenting PRISM's verification capabilities. Ultimately, we aim to show that our prototype benefits mission developers by improving the correctness of UAV mission specifications. We will also evaluate the portability of cascading verification, i.e., the usability of our method in the context of different application domains.

6.1 Abstraction

Because it was unfeasible to involve practitioners in the evaluation of our prototype's utility, we opted instead for a metrics-based analysis of 58 mission plans. These plans were based on real-world mission scenarios developed independently by DARPA and DRDC [10, 11]. We evaluated our approach by comparing the *lines of code* (LOC) and numbers of lexical tokens required to specify missions in YAML against the LOC and tokens in the combined DTMC and PCTL code synthesized by the CVC. On average, our prototype synthesizes PRISM code that is 3.127 and 4.490 times greater than the size of YAML input with regard to LOC and tokens, respectively. (The standard deviations were 52.4% and 95.4%, respectively.) These results, which are presented succinctly in the interest of space, provide preliminary evidence of non-trivial reduction in the effort required to produce mission models and properties.

We observe that tactical missions generate more LOC and tokens than *standalone mission plans*, which are mission plans not associated with the more specialized tactical subdomain. Specifically, tactical mission plans generate PRISM code that is on average 3.933 and 5.992 times greater than the size of YAML input with regard to LOC and tokens, respectively. (The standard deviations were 24.0% and 59.2%, respectively.) Because the effort required to synthesize PRISM code is proportional to the effort required to synthesize the LOC and tokens that constitute the code, tactical mission plans result in added value for mission developers. This observation suggests that, with respect to tactical missions, the utility of our prototype is proportional to the threat level associated with any given mission plan. More broadly, increased LOC and token output suggests that the utility of cascading verification may be proportional to the amount of automated reasoning required to synthesize pertinent artifacts, a conclusion that justifies our motivation to augment model checking with formalized domain knowledge.

6.2 Effectiveness

Because it cannot account for the intricate syntax of the PRISM language, a LOC- and token-based analysis offers limited insight into the inherent complexity of model and property specifications. We investigate complexity further by considering *behavioral modeling errors* specific to the PRISM language that can be eliminated with the automated synthesis of PRISM artifacts (at least for the segment of the mission space that we have explored thus far). Behavioral modeling errors include variable declarations with incorrect values; incorrect or missing command actions; incorrect command probabilities; and incorrect command updates (PRISM language constructs were introduced in Section 4.3). These errors are significant, perhaps more so than the errors uncovered during the model checking process, because they can mislead mission developers by causing PRISM to verify erroneous mission plans.

We also consider mission specification errors that are beyond the scope of PRISM's verification capabilities. These *domain-specific errors* are detected by either Pellet or the SWI-Prolog compiler during the synthesis process. We have identified 28 domain-specific errors, across six error classes, that impact the correctness of UAV missions. In the context of the OWL language and CEMO:

- *Disjoint class errors* occur when individuals are declared in system specifications to be instances of incompatible classes; for example, a hover action can also be a kinetic action, but not a sensor action.
- *Existential restriction errors* occur when individuals fail to participate in mandatory relationships, as specified by the OWL keyword `some`; for example, every asset must execute at least one kinetic action.
- *Data property value errors* occur when data property values declared in system specifications fall outside the ranges of corresponding data properties encoded in CEMO (data properties were described in Section 4.1); for example, the endurance of a Hummingbird is 1200 time-steps.
- *Data property domain and range errors* occur when data property domain and range types declared in system specifications are inconsistent with the domain and range types of the corresponding data properties encoded in CEMO; for example, CEMO specifies that every `Hummingbird` individual must be associated with a datatype property `hasCostValue` of type `int`.
- *Object property domain and range errors* occur when object property domain and range types declared in system specifications are inconsistent with the domain and range types of the corresponding object properties encoded in CEMO (object properties were described in Section 4.1); for example, CEMO specifies that the object property `hasAction` associates every member of

class `Asset` (the domain of `hasAction`) with a member of class `KineticAction` (the range).

- *Threatened asset errors* occur when mission plans comprise a threatened asset that is not also a valid asset (threatened and valid assets were described in Section 3 and Section 4.3, respectively).

Mission correctness can clearly be compromised by domain-specific and behavioral modeling errors, which occur during the design/implementation and verification phases, respectively, of the mission development process. Our prototype augments PRISM's effectiveness by preventing both of these error types.

6.3 Probabilistic Verification

Finally, we consider PRISM's ability to meaningfully verify UAV mission plans (or rather, we consider the utility of the DTMC and PCTL artifacts synthesized by our prototype). For this part of the evaluation, eighteen of the 58 mission plans described above were seeded with errors, including deadlock and non-reachable states, that violated desirable behavioral properties. One mission plan failed (i.e., contained errors that resulted in a 0.0 probability of success) because of an unacceptably low RAF value; nine mission plans failed because kinetic or sensor action workflow durations exceeded the endurances of the assets to which those workflows were assigned; and eight mission plans contained action workflow errors that resulted in deadlock. These errors were successfully identified by our prototype. While the correctness of some mission plans was absolute (with a 0.0 or 1.0 probability of success) several mission plans, including plans comprising threat area incursions, were associated with variable probabilities of success. For example, the probability of success for Mission A is approximately 0.299 (as described in Section 5.3).

6.4 Discussion

By automating the synthesis of PRISM artifacts, and by providing multiple stages of reasoning and analysis, our prototype enhances the abstraction level of model and property specifications, and the effectiveness of probabilistic model checking, respectively. This cascading approach to verification improves mission correctness to a degree that is evidently unattainable by the individual components that constitute the prototype.

In conjunction with the complexity of the UAV domain, and the real-world DARPA and DRDC mission scenarios underpinning this evaluation, the above results suggest that cascading verification can be ported to different application domains. The portability of our method is supported by the general purpose of its constituent technologies including OWL+SWRL, Prolog, and DTMC and PCTL. Presently, we cannot make the same argument for the *connections* that link those technologies in the context of the method. But cascading verification is an extension of semantic model checking methods with identical or comparable constituent technologies (similarities to semantic model checking will be presented in Section 7). The successful application of these methods to the Web service domain further supports the portability of cascading verification.

We note that this evaluation is preliminary. Further work is required to determine the utility of our prototype in the context of a more sophisticated mission specification language and domain model; and the ability of cascading verification to support probabilistic model checking in the context of other non-trivial domains.

7. RELATED WORK

Cascading verification can be considered a form of semantic model checking, which has been studied exclusively in the context of the Web service domain.

Narayanan and McIlraith encode Web service capability descriptions and behavioral properties with DAML-S and Petri net formalisms, respectively [3]. DAML-S is a DAML+OIL ontology for describing Web services. For any given Web service, an implemented system generates the Petri net corresponding to the DAML-S description of that service. The resulting net is used by KarmaSIM, a modeling and simulation environment, to perform various analysis techniques including reachability analysis and deadlock detection.

The model checking algorithm presented by Di Pietro et al. uses a DL-based ontology to formalize the Web service domain [4]. The behavior of each Web service is modeled as a *state transition system* (STS), while behavioral requirements are encoded with CTL. Both STS and CTL formalisms are extended with semantic annotations. For any given Web service, the algorithm generates a finite STS corresponding to the annotated description of that service. The resulting model is verified with model checking. The same algorithm is used by Boaro et al. to verify Web service security requirements [5].

Oghabi et al. use OWL-S, an OWL ontology that supersedes DAML-S, to describe Web service behavior [6]. For any given Web service, an implemented system generates a PRISM model corresponding to the OWL-S description of that service. The resulting model is verified with PRISM. Ankolekar et al. translate OWL-S process models to equivalent PROMELA models, which are verified with the SPIN model checker [7]. Liu et al. extend OWL-S with multiple annotation layers for specifying Web service flow properties including temporal constraints [8]. Annotated OWL-S models are transformed to corresponding *time constraint Petri net* (TCPN) models, which are verified with model checking. Lomuscio and Solanki express OWL-S process models with the *interpreted systems programming language* (ISPL) [9]. ISPL is the system description language for MCMAS, a symbolic model checker tailored to the verification of multi-agent systems. In this context, Web services and Web service compositions are viewed as agents and multi-agent systems, respectively.

Our method is perhaps most compatible with the work presented by Oghabi et al. Similarities include the motivation to verify stochastic behavior and the development of a system that synthesizes PRISM models from OWL knowledge. But unlike our prototype, the system developed by Oghabi et al. does not synthesize behavioral properties, nor does it exploit inferred knowledge to support the synthesis of DTMC artifacts. Inferred knowledge is utilized in other work including that of Narayanan and McIlraith, Di Pietro et al. and Boaro et al. But existing work is exclusively concerned with the verification of Web services, and does not address the expressive and reasoning limitations that constrain OWL. The work presented in this paper is (to our knowledge) unique because it addresses semantic model checking limitations, and applies the resulting method to a novel domain.

8. CONCLUSIONS AND FUTURE WORK

This paper describes a novel cascading verification method that uses composite reasoning over high-level system specifications and formalized domain knowledge to synthesize both system models and their desired behavioral properties. With cascading verification, model builders use a high-level DSL to encode system specifications that can be analyzed with model checking. Domain knowledge is encoded in OWL+SWRL and Prolog, which are combined to overcome their individual limitations. Synthesized DTMC models and PCTL properties are analyzed with the probabilistic model checker PRISM. Cascading verification was illustrated with a prototype system that verified the correctness of UAV mission plans. An evaluation of this prototype revealed non-trivial reductions in the size and complexity of input system specifications compared to the artifacts synthesized for PRISM.

We have identified several promising directions for future work. Composite CVC inferences are currently unidirectional, with Prolog facts derived from knowledge encoded in OWL+SWRL. While conceptually and practically appealing, this inference *pipeline* constrains the reasoning process from refining Prolog inferences with ontological knowledge, and increases the potential for knowledge duplication. We aim to address these limitations by developing a knowledge representation framework that can support more flexible, iterative reasoning.

A second issue pertains to the artifacts that constitute the CVC knowledge base including CEMO, the Prolog rule-base, and the DTMC and PCTL templates. These artifacts should be extensible to reflect changes in domain knowledge. Extensions should in turn be verifiable to ensure that domain knowledge remains consistent across the entire knowledge base. This requirement provides impetus for the development of a mechanism that will automate the consistency management process.

Finally, we intend to further the evaluation of our method and prototype by enhancing the sophistication of the mission specification language and domain model presented in this paper.

9. ACKNOWLEDGEMENTS

The material presented in this paper is based on work supported by AFOSR grant FA9550-09-1-0687 and EOARD grant FA8655-10-1-3007. Any conclusions, findings, opinions and recommendations expressed in this paper are those of the author(s) and do not necessarily reflect the views of AFOSR or EOARD. The authors would like to thank the anonymous referees for their valuable feedback.

10. REFERENCES

[1] C. Baier and J.-P. Katoen, *Principles of Model Checking.* The MIT Press, 2008.

[2] Robby, M. B. Dwyer, and J. Hatcliff, "Bogor: An Extensible and Highly-Modular Software Model Checking Framework," in *Proceedings of the 9th European Software Engineering Conference Held Jointly With the 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-11, pp. 267–276, ACM, 2003.

[3] S. Narayanan and S. A. McIlraith, "Simulation, Verification and Automated Composition of Web Services," in *Proceedings of the 11th International Conference on World Wide Web*, WWW '02, pp. 77–88, ACM, 2002.

[4] I. D. Pietro, F. Pagliarecci, and L. Spalazzi, "Model Checking Semantically Annotated Services," *IEEE Transactions on Software Engineering*, vol. 38, pp. 592–608, May 2012.

[5] L. Boaro, E. Glorio, F. Pagliarecci, and L. Spalazzi, "Semantic Model Checking Security Requirements for Web Services," in *Proceedings of the 2010 International Conference on High Performance Computing and Simulation*, HPCS '10, pp. 283–290, IEEE, 2010.

[6] G. Oghabi, J. Bentahar, and A. Benharref, "On the Verification of Behavioral and Probabilistic Web Services Using Transformation," in *Proceedings of the 2011 IEEE International Conference on Web Services*, ICWS '11, pp. 548–555, IEEE Computer Society, 2011.

[7] A. Ankolekar, M. Paolucci, and K. Sycara, "Towards a Formal Verification of OWL-S Process Models," in *Proceedings of the 4th International Conference on the Semantic Web*, ISWC '05, pp. 37–51, Springer-Verlag, 2005.

[8] R. Liu, C. Hu, and C. Zhao, "Model Checking for Web Service Flow Based on Annotated OWL-S," in *Proceedings of the 2008 Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, SNPD '08, pp. 741–746, IEEE Computer Society, 2008.

[9] A. Lomuscio and M. Solanki, "Mapping OWL-S Processes to Multi Agent Systems: A Verification Oriented Approach," in *Proceedings of the 2009 International Conference on Advanced Information Networking and Applications Workshops*, WAINA '09, pp. 488–493, IEEE Computer Society, 2009.

[10] DARPA, "UAVForge." [online] Available at: `http://www.uavforge.net/`.

[11] G. Youngson, K. Baker, D. Kelleher, and S. Williams, "Project Support Services for the Operational Mission and Scenario Analysis for Multiple UAVs/UCAVs Control From Airborne Platform," March 2004.

[12] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker, "PRISM: A Tool for Automatic Verification of Probabilistic Systems," in *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '06, pp. 441–444, Springer-Verlag, 2006.

[13] C. C. Evans, "The Official YAML Web Site." [online] Available at: `http://yaml.org/`.

[14] I. Horrocks and P. F. Patel-Schneider, "Knowledge Representation and Reasoning on the Semantic Web: OWL," in *Handbook of Semantic Web Technologies* (J. Domingue, D. Fensel, and J. A. Hendler, eds.), ch. 9, pp. 365–398, Springer, 2011.

[15] B. Motik, I. Horrocks, R. Rosati, and U. Sattler, "Can OWL and Logic Programming Live Together Happily Ever After?," in *Proceedings of the 5th International*

Conference on the Semantic Web, ISWC '06, pp. 501–514, Springer-Verlag, 2006.

[16] M. Şensoy, G. de Mel, W. W. Vasconcelos, and T. J. Norman, "Ontological Logic Programming," in *Proceedings of the International Conference on Web Intelligence, Mining and Semantics*, WIMS '11, pp. 44:1–44:9, ACM, 2011.

[17] T. Matzner and P. Hitzler, "Any-World Access to OWL From Prolog," in *Proceedings of the 30th Annual German Conference on Advances in Artificial Intelligence*, KI '07, pp. 84–98, Springer-Verlag, 2007.

[18] N. Papadakis, K. Stravoskoufos, E. Baratis, E. G. M. Petrakis, and D. Plexousakis, "PROTON: A Prolog Reasoner for Temporal ONtologies in OWL," *Expert Systems With Applications*, vol. 38, pp. 14660–14667, November 2011.

[19] K. Samuel, L. Obrst, S. Stoutenberg, K. Fox, P. Franklin, A. Johnson, K. Laskey, D. Nichols, S. Lopez, and J. Peterson, "Translating OWL and Semantic Web Rules Into Prolog: Moving Toward Description Logic Programs," *Theory and Practice of Logic Programming*, vol. 8, pp. 301–322, May 2008.

[20] G. Lukácsy and P. Szeredi, "Efficient Description Logic Reasoning in Prolog: The DLog System," *Theory and Practice of Logic Programming*, vol. 9, pp. 343–414, May 2009.

[21] J. M. Almendros-Jiménez, "A Prolog-Based Query Language for OWL," *Electronic Notes in Theoretical Computer Science*, vol. 271, pp. 3–22, March 2011.

[22] D. Elenius, "SWRL-IQ: A Prolog-Based Query Tool for OWL and SWRL," in *Proceedings of OWL: Experiences and Directions Workshop 2012*, vol. 849 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2012.

[23] F. Baader, I. Horrocks, and U. Sattler, "Description Logics as Ontology Languages for the Semantic Web," in *Mechanizing Mathematical Reasoning, Essays in Honor of Jörg H. Siekmann on the Occasion of His 60th Birthday* (D. Hutter and W. Stephan, eds.), no. 2605 in Lecture Notes in Computer Science, pp. 228–248, Springer, 2005.

[24] M. Krötzsch and S. Speiser, "ShareAlike Your Data: Self-Referential Usage Policies for the Semantic Web," in *Proceedings of the 10th International Conference on the Semantic Web—Volume Part I*, ISWC '11, pp. 354–369, Springer-Verlag, 2011.

[25] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean, "SWRL: A Semantic Web Rule Language Combining OWL and RuleML." W3C Member Submission, May 2004.

[26] R. E. McGrath and J. Futrelle, "Reasoning About Provenance With OWL and SWRL Rules," in *AAAI Spring Symposium: AI Meets Business Rules and Process Management*, pp. 87–92, AAAI, 2008.

[27] R. Volz, S. Decker, and D. Oberle, "Bubo—Implementing OWL in Rule-Based Systems," in *Proceedings of the 12th International Conference on World Wide Web*, WWW '03, ACM, 2003.

[28] V. S. Costa, R. Rocha, and L. Damas, "The YAP Prolog System," *Theory and Practice of Logic Programming*, vol. 12, pp. 5–34, January 2012.

[29] S. Greco and F. A. Lisi, "Logic programming Languages for Databases and the Web," in *A 25-Year Perspective on Logic Programming* (A. Dovier and E. Pontelli, eds.), pp. 183–203, Springer-Verlag, 2010.

[30] M. Kwiatkowska and D. Parker, "Advances in Probabilistic Model Checking," in *Software Safety and Security: Tools for Analysis and Verification* (T. Nipkow, O. Grumberg, and B. Hauptmann, eds.), vol. 33 of *NATO Science for Peace and Security Series D: Information and Communication Security*, pp. 126–151, IOS Press, 2012.

[31] A. Rango, A. Laliberte, K. Havstad, C. Winters, C. Steele, and D. Browning, "Rangeland Resource Assessment, Monitoring, and Management Using Unmanned Aerial Vehicle-Based Remote Sensing," in *Proceedings of the IEEE International Geoscience & Remote Sensing Symposium*, IGARSS '10, pp. 608–611, IEEE, 2010.

[32] J. A. Jiménez-Berni, P. J. Zarco-Tejada, L. Suarez, and E. Fereres, "Thermal and Narrowband Multispectral Remote Sensing for Vegetation Monitoring From an Unmanned Aerial Vehicle," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 47, pp. 722–738, March 2009.

[33] F. Heintz, P. Rudol, and P. Doherty, "From Images to Traffic Behavior—A UAV Tracking and Monitoring Application," in *Proceedings of the 10th International Conference on Information Fusion*, FUSION '07, pp. 1–8, IEEE, 2007.

[34] S. Karaman and E. Frazzoli, "Complex Mission Optimization for Multiple UAVs Using Linear Temporal Logic," in *Proceedings of the 2008 American Control Conference*, ACC '08, pp. 2003–2009, IEEE, 2008.

[35] P. Tosic, M.-W. Jang, S. Reddy, J. Chia, L. Chen, and G. Agha, "Modeling a System of UAVs on a Mission," in *Proceedings of the 7th World Multiconference on Systemics, Cybernetics and Informatics*, SCI '03, pp. 508–514, 2003.

[36] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz, "Pellet: A practical OWL-DL Reasoner," *Journal of Web Semantics*, vol. 5, pp. 51–53, June 2007.

[37] "SWI-Prolog's home." [online] Available at: `http://www.swi-prolog.org/`.

[38] M. Horridge, N. Drummond, J. Goodwin, A. L. Rector, R. Stevens, and H. Wang, "The Manchester OWL Syntax," in *Proceedings of the OWLED*06 Workshop on OWL: Experiences and Directions*, CEUR-WS.org, 2006.

[39] M. Horridge, *A Practical Guide to Building OWL Ontologies Using Protégé 4 and CO-ODE Tools.* The University of Manchester, 1.3 ed., 2011.