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

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

*Dept. 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

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

ABSTRACT

Model checking is an established method for verifying behavioral properties of system models. But 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* at relatively low levels of abstraction.

In this paper, we demonstrate that formalized domain knowledge can be reused to raise the abstraction level of model and property specifications, and hence the effectiveness of model checking. We describe a novel method for domain-specific model checking called *cascading verification* that uses composite reasoning over high-level system specifications and formalized domain knowledge to synthesize *both* low-level system models and their behavioral properties for verification. In particular, model builders use a high-level *domain-specific language* (DSL) based on YAML to express system specifications that can be verified with *probabilistic model checking*. Domain knowledge is encoded in the Web Ontology Language (OWL), the Semantic Web Rule Language (SWRL) and Prolog, which are used in combination to overcome their individual limitations. A compiler then synthesizes models and properties for verification by the probabilistic model checker PRISM. We illustrate cascading verification for the domain of *uninhabited aerial vehicles* (UAVs), for which we have constructed a prototype implementation. An evaluation of this prototype reveals nontrivial reductions in the size and complexity of input 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

ESEC/FSE'13, August 18–26, 2013, Saint Petersburg, Russia
Copyright 2013 ACM 978-1-4503-2237-9/13/08...$15.00
http://dx.doi.org/10.1145/2491411.2491454

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, for example by employing partial order reduction and by exploiting symmetries and other state-space properties, thereby enabling model builders to verify larger, more elaborate models. But the complexity associated with specifying models and properties in the first place has yet to be addressed sufficiently. Popular model checkers tend to support low-level modeling languages that require intricate models to represent even the simplest systems. For example, PROMELA—the language of the model checker Spin—is essentially a dialect of the low-level programming language C. Another example is the modeling language used by the probabilistic model checker PRISM, whose lack of control structures forces model builders to pollute model components with counter variables that explicitly encode the components' state transitions.

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. In this paper we demonstrate that formalized domain knowledge can be reused to raise the abstraction level of model and property specifications, and hence the effectiveness of model checking.

Previous research has demonstrated that formal domain knowledge can decrease specification and verification costs. For instance, 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 knowl-

edge 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 limitations in its expressive and reasoning abilities reduce its utility in semantic model checking. The Semantic Web Rule Language (SWRL)—a W3C-approved OWL extension—addresses some of these limitations, but neither OWL nor SWRL can reason effectively with negation. The logic programming language Prolog does support negation, but then Prolog lacks some desirable expressive features of OWL, including support for equivalence and disjointness. Thus, it is necessary to use OWL, SWRL and Prolog in combination in order to overcome their individual limitations for semantic model checking.

This paper describes a novel method for domain-specific model checking called *cascading verification*, which uses composite reasoning over high-level system specifications and formalized domain knowledge to synthesize *both* low-level system models and their behavioral properties for verification. In particular, model builders use a high-level *domain-specific language* (DSL) based on YAML [10] (a human-friendly data serialization format) to express system specifications that can be verified with *probabilistic model checking*. A *compiler* uses automated reasoning to verify the consistency of each specification with domain knowledge encoded in OWL+SWRL and Prolog, and then explicit and inferred domain knowledge are used by the compiler to synthesize a *discrete-time Markov chain* (DTMC) model and the *probabilistic computation tree logic* (PCTL) properties from domain-specific templates. Finally, the probabilistic model checker PRISM [11] verifies the model against the properties. Thus, verification *cascades* through several stages of reasoning, analysis and synthesis.

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

In this paper we illustrate cascading verification with a prototype we built for the domain of *uninhabited aerial vehicle* (UAV) mission plans. We used this prototype to analyze 58 mission plans, which were based on real-world mission scenarios developed by DARPA [12] and the Defense Research and Development Canada (DRDC) agency [13]. UAVs are a particularly interesting and important domain, and the stochastic nature of UAV missions makes probabilistic model checking a natural choice for their verification.

This paper thus presents three contributions. First, we describe a novel method of model checking 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 specification comprising 92 tokens, cascading verification synthesized 104 lines of PRISM code comprising 744 tokens and three behavioral properties. 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.

2. BACKGROUND

Cascading verification employs a carefully selected suite of modeling and verification technologies—OWL+SWRL, Prolog, DTMCs and PCTL. In this section, we describe these technologies in greater detail and discuss the rationale for combining them into a single method for domain-specific model checking. We also present some background about the UAV domain supported by our prototype implementation of cascading verification.

We note that many people have discussed the practical limitations of OWL for formalizing domain knowledge and have sought to extend its capabilities with Prolog or extensions like SWRL [14]. These same limitations, discussed below, arose in our work with the UAV domain, which led us to devise a similar combination of technologies.

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* [15]. DL concepts and individuals are roughly comparable to classes and objects, respectively, in object-oriented programming, while roles are comparable to UML associations. DLs reason over TBox axioms to infer key properties of concepts, including *satisfiability*, *subsumption*, *equivalence* and *disjointness* [16].

OWL is an ontology specification language based on the description logic $\mathcal{SHOIN}(\mathbf{D})$ [15] that supports the automated processing of formalized knowledge. But OWL is constrained by expressive and reasoning limitations inherent in $\mathcal{SHOIN}(\mathbf{D})$. For example, OWL can model object relations that form tree-like patterns, but more complex relationships—such as the triangular relationship that exists between a child, the child's father, and the father's brother—are not expressible in OWL [14]. Furthermore, OWL cannot model the self-referential relationship between an individual and itself [17]. SWRL is one of several proposed extensions to OWL, and it extends OWL with Horn clause-like rules to address some of OWL's expressive limitations [18]. But like OWL, SWRL cannot reason with negation in an effective manner [14, 19]. Modeling problems that are inexpressible in OWL+SWRL can be addressed with Prolog [14, 20].

Unlike OWL, Prolog is a rule-based language, whereby rules and facts constitute a compiled knowledge base that can support efficient querying. And unlike OWL+SWRL, Prolog can reason effectively with negation. But Prolog is not without its own limitations. Prolog is based on a FOL subset expressed with first-order Horn clauses [20]. OWL can be translated into formulas of a 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.

The combination of OWL, SWRL and Prolog we use in cascading verification overcomes the individual limitations of each formalism and thus provides a rich expressive basis for domain modeling.

2.2 Probabilistic Model Checking

Probabilistic model checking is a method for verifying behavioral properties of stochastic systems, which are systems affected by message delays, failure rates and other random phenomena [1, 21], and PRISM is arguably the most widely used probabilistic model checker. To model these systems, 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 timesteps. 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; $\mathbf{P} : S \times S \rightarrow [0, 1]$ is the transition probability function, such that $\forall s : \sum_{s' \in S} \mathbf{P}(s, s') = 1$; $\iota_{init} : S \rightarrow [0, 1]$ is the initial state distribution, such that $\sum_{s' \in S} \iota_{init}(s) = 1$; and $L : S \rightarrow 2^{AP}$ is a labeling function that maps each state to a subset of AP, which is a set of atomic propositions that abstract key characteristics of a modeled system.

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 is 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 traces induced by a DTMC, and J specifies a closed sub-interval of the interval $[0, 1]$ 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 missions over civilian airspace [22, 23, 24].

The UAV domain exhibits complexity at many levels. In particular, UAVs contain sophisticated payloads, multiple sensors and increasing computational power. In time, these capabilities will enable UAV swarms to execute complex multi-task missions with reduced human supervision [25]. For a given mission, a UAV 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 [26]. These factors combine to form a complex stochastic state space that makes UAV mission plans particularly well suited to the use of probabilistic model checking.

3. METHOD OVERVIEW

In this section we first present an overview of cascading verification from a domain-independent viewpoint, and then we present an example UAV mission plan that we use as a running example for the rest of the paper. Subsequent sections describe the elements of the method in greater detail, with the running example used to illustrate key concepts.

3.1 From Specification to Verification

Figure 1 presents a high-level schematic of cascading verification. Domain experts, which are one of the method's primary stakeholders, use OWL to define domain concepts and their relationships, and SWRL to supplement the specification with rules. The result is a TBox that formalizes domain concepts. Domain experts also use Prolog to define additional rules, and they use PRISM's modeling and property specification languages to define DTMC and PCTL templates, respectively. The OWL+SWRL, Prolog, DTMC and PCTL artifacts constitute the formal domain knowledge available to model builders. The model builders—also primary stakeholders—use a high-level DSL to specify system specifications that can be verified eventually with model checking. We note that domain knowledge thus is formalized once and subsequently reused to support the construction of any number of system models.

Given a high-level system model expressed by a model builder in the DSL, a *cascading verification compiler* (CVC) synthesizes both the DTMC and PCTL artifacts corresponding to that specification, as follows:

1. The CVC transforms the model into ABox assertions.
2. A reasoning engine for OWL+SWRL verifies the consistency of the generated ABox with respect to the TBox defined previously in OWL+SWRL by domain experts. In doing this, the reasoner ensures that model constructs specified in the DSL are consistent with OWL+SWRL axioms. Inconsistencies signify an invalid model, which causes the compilation process to terminate with an error. If the model is found to be consistent with the TBox, then the reasoner generates additional inferences from the TBox to aid the eventual synthesis of DTMC and PCTL artifacts.
3. The CVC transforms the ontological knowledge inferred previously into Prolog facts. A Prolog engine inputs the generated facts plus the Prolog rules defined previously by domain experts and then proceeds to generate further inferences.
4. The CVC uses Prolog inferences in conjunction with explicit and inferred domain knowledge to synthesize DTMC and PCTL artifacts from the templates predefined by domain experts.

PRISM inputs the synthesized artifacts, verifies the system model against its behavioral properties (including possibly additional properties provided by the model builders), and returns logical and probabilistic results from the verification. If these results are deemed acceptable by the model builders, then the model can be implemented and deployed for real-world execution (via some process outside the scope of our method).

3.2 An Example UAV Mission

In our implementation of cascading verification for the UAV domain, model builders use a dialect of YAML we designed as their DSL for specifying mission plans. Mission plans

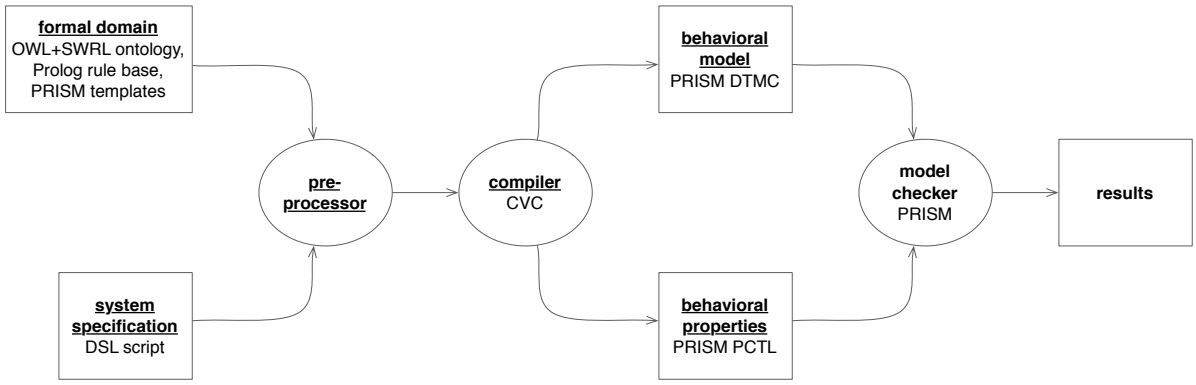

Figure 1: A high-level schematic of our method and prototype. Rectangular and oval shapes represent data and processes, respectively; bold and normal text distinguishes method from prototype, respectively; and underlined text represents our research contributions with respect to the UAV domain.

comprise UAV *assets* and their assigned *action workflows*. The YAML specification in Listing 1 models Mission A, an example mission that is representative of a suite of 58 mission plans we developed for this research:

```
Action:                                                  1
  TraversePathSegmentAction:                             2
    - 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 specification 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 its UAV to traverse a path between two waypoints. The latitudes and longitudes of the waypoints are given in an array, with the latitude of waypoint one followed by the longitude of waypoint one, which in turn is followed by the latitude of waypoint two and then the longitude of waypoint two. In a sequence of path segment traversal actions associated with some asset, the end coordinates of one action serve implicitly as the start coordinates of the subsequent action. Thus, only the first action in the sequence is specified with four coordinates (for its start and end waypoints), while each remaining action is specified with two coordinates (for its end waypoint). For example, the actions `TPSA1` and `TPSA2` are specified for asset `H1`, and so the end coordinates of action `TPSA1` in line 6 are the (implicit) start coordinates of action `TPSA2` in line 9.

Before a mission plan can be compiled by the CVC, it must undergo a preprocessing phase. Such preprocessing will be needed for any domain in order to account for domain-specific elements in system models. During preprocessing for the UAV domain, the CVC uses the geographic coordinates appearing in mission plans (such as the waypoint coordinates described above), plus data from external sources characterizing expected operational environments, to perform geodetic calculations.[1] The equations that support these calculations are hard-coded in the preprocessor. Geographic information resulting from preprocessing—which may also include, for example, the durations of *threat area incursions* undertaken by UAVs—is integrated with the ABox generated by the CVC.

Once this preprocessed information is part of the ABox, then it is available to support the inferences performed by the CVC as described in steps 2 and 3 of Section 3.1. For example, in step 2, if geodetic calculations establish that a specified action is a *threat area action* (because the path of the action intersects the boundary of a threat area), then the asset undertaking that incursion is inferred to be a *threatened asset*. And in step 3, the last kinetic action in an action workflow is inferred to be a *default terminal action*.

The following two sections present in detail the steps of Section 3.1 and illustrate them using Mission A described above.

4. DOMAIN MODELING

This section describes the encoding of domain knowledge in OWL+SWRL, Prolog, and DTMC and PCTL templates. We illustrate the application of these technologies to the UAV domain using Mission A, the example mission presented in the previous section.

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

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

4.1 Semantic Modeling

With cascading verification, experts from some domain of interest use an ontology to formally define domain concepts and their relationships. For our prototype, we have developed a *complex missions ontology* (CEMO) formalizing the UAV domain. Figure 2 depicts CEMO's hierarchy of domain concepts, or classes. Five classes—including `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. Inheritance is indicated by indentation. For instance, class `Action` is extended by classes `KineticAction` and `SensorAction`, and `KineticAction` is extended in turn by classes `HoverAction` and `TraversePathSegmentAction`.

The formal definition of a class includes definitions of its *object properties* and *data properties*, which specify relationships between individuals of the class and, respectively, individuals of other classes and data values. For instance, below is the formal definition of class `KineticAction` using *Manchester OWL syntax*, a human-friendly ontology representation language [27]:

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

The listing specifies that class `KineticAction` is a subclass of class `Action` and has two properties, `hasDurationInSections` and `hasPrecondition`. The former property carries a value of the built-in data type `int` and must be present in every `KineticAction` individual (as indicated by the keyword `some`). The latter property is optional for a `KineticAction` individual (as indicated by the keyword `only`) and carries as its value an individual of class `Action`.

The other classes of Figure 2 are defined formally in a similar fashion. For instance, 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 used 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 some action workflow. Mission A comprises three preconditions (lines 10, 18 and 22 in Listing 1) relating actions assigned to the same asset. A fourth precondition (line 10) associates action `TPSA2` with action `TPSA3`, thereby coupling the behavior of assets `H1` and `H2`.

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 mainly to support automated reasoning, tactical concepts are not available to model builders 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 appear ultimately in the synthesized DTMC models (as explained further in Section 4.3).

4.2 Rule-Based Modeling

OWL is a powerful knowledge representation formalism, but it has fundamental expressive and reasoning limitations. As an example, OWL cannot model *cross-cutting actions* of UAVs. We consider an action a to be cross-cutting if a is a precondition to an action b, but a and b are assigned to different assets. By this definition, action `TPSA3` in Mission A is a cross-cutting kinetic action. We can use OWL to capture some but not all aspects of this relationship, as shown in the following OWL code:

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

The problem is that we cannot express in OWL the requirement that the `Asset` mentioned in line 3 be different from the `Asset` in line 6. We therefore resort to SWRL to specify this additional requirement, as in the following SWRL rule, which uses the built-in OWL property `DifferentFrom` (where question-marks denote variables).[2]

[2] `DifferentFrom` is an OWL construct used to differentiate individuals that are members of a class. It can be used in the definition of an individual, and in SWRL rules, but it cannot be used to define a class of individuals, which is the problem raised in the definition of `CrossCuttingKineticAction`.

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

Thus, SWRL rules extend the expressive power of OWL, but like OWL, SWRL cannot reason effectively with negation. When formalizing the UAV domain, we often found it desirable to state that something is not true or must not occur, rather than always try to state that something is true or must occur. It is surely the case that such negative statements arise naturally in many domains. We therefore use Prolog to encode rules that must reason with negation. The Prolog rule below formally defines rule `terminal`, which comprises three negated clauses (lines 3–5):

```
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
```

Rule `terminal` encapsulates both explicit and inferred ontological knowledge; explicit knowledge is encoded as predicate `has_action(A, X)` (line 2), while the three negated clauses encode knowledge inferred by the OWL+SWRL engine. Predicate `is_precondition_to(X, _)` (line 3) provides an example for the transition of inferred knowledge from OWL to Prolog. The following OWL code formally defines the object property `isPreconditionTo`, an inferred relationship that inverts the object property `hasPrecondition`:

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

Mission A comprises several instances of the DSL construct `preconditions` (lines 10, 18 and 22 in Listing 1). The CVC transforms information contained in these constructs into knowledge encoded as property `hasPrecondition`. Knowledge inferred with respect to `hasPrecondition`, via the inverted property `isPreconditionTo`, is transformed by the CVC into knowledge encoded as Prolog predicate `is_precondition_to(X, _)`, thus making this knowledge available for Prolog-based inferences.

4.3 Behavioral Modeling

As shown in the previous section, OWL+SWRL and Prolog are suitable formalisms for encoding domain knowledge. Likewise, the state-based modeling language of PRISM is a suitable formalism for formalizing system behavior. We use PRISM's DTMC sub-language as the basis for reusable behavior templates.

As an example of a PRISM template, the PRISM template below represents the behavior of `Asset` individuals 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}] e#{asset.id}>0                        5
      & d#{action.id}>0                                  6
      -> (e#{asset.id}'=e#{asset.id}-1);                 7
  END                                                    8
  [#{asset.last_action.type}] e#{asset.id}=0             9
      | d#{asset.last_action.id}=0                       10
      -> true;                                           11
endmodule                                                12
```

The template is specified as a *module*, the PRISM language construct for a single-threaded unit of behavior. The template uses string interpolation—denoted by code snippets `#{ ... }`—to designate template parameters, and arguments for instantiating the parameters are derived by the CVC 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 expressed in YAML; 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 the reasoning engines for OWL+SWRL and Prolog (via a process 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 above), 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 a command having just one update. Each command may be labeled with an *action*, which forces two or more modules to transition their states simultaneously.

Lines 4 and 8 in the above listing use meta-code highlighted with purple text to instruct the CVC on how to complete the template. In this case, the meta-code instructs the CVC to execute a for loop that generates one guarded command for each matching value of `action`. Each asset module command generated by this loop represents the execution of a kinetic action. The keyword `true` in the last command (line 11) denotes the end of execution for a specific module. This type of command prevents spurious deadlocks from arising solely from the synthesis process.

Note that the commands of the asset template above comprise only single updates and thus contain no probabilities. In contrast, the PRISM code below formally defines a template for asset survivability modules, which contain commands with multiple updates (lines 12–15):

```
FOR action IN asset.threat_area_actions                  1
formula actn#{action.id}_tai =                           2
    d#{action.id}>finish#{action.id} &                   3
    d#{action.id}<=start#{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
```

The probability of execution for each update is expressed as `asset.vulnerability` parameters; arguments for the parameters are inferred from explicit domain knowledge encoded in CEMO (as described in Section 4.1). The vulnerability values we use in CEMO were chosen arbitrarily and thus eventually would be replaced with data that represent real asset capabilities, terrain types and weather conditions.

Each synthesized survivability module specifies the probability of survival for a threatened asset that has also been inferred by the OWL+SWRL engine 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 above use meta-code to define a for loop that instructs the CVC to generate one or more PRISM `formula` constructs. Each formula defines an overlap between the execution of a threat area incursion and the execution of a threat area action via the identifier `action.id`. The overlap is delineated by variables `start#{action.id}` and `finish#{action.id}`. These variables are assigned geographic information, which is calculated by the CVC during preprocessing (as described in Section 3).

The formula identifier `actn#{action.id}_tai` (defined in line 2 and used in lines 11 and 17) represents the execution of a specific threat area incursion. The Boolean variable `a#{asset.id}d` (defined in line 8 and used in lines 10, 13, 15 and 16) represents asset destruction. These constructs combine to form logical expressions in lines 10–11 and 16–17 that are used as the guards in the asset survivability module. When these expressions are considered in union, their logical truth values fully cover the possible asset behaviors during threat area incursions. Specifically, a valid asset exists in one of the following disjoint states: (1) *not* destroyed and executing a threat area incursion; or (2) destroyed as a consequence of executing a threat area incursion or else *not* executing a threat area incursion.

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 template for mission properties. 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 synthesizing additional clauses in the property.

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 for the UAV domain combines these technologies to support the probabilistic verification of UAV mission plans. This section describes in greater detail the steps of cascading verification presented in Section 3.1 and illustrates them with Mission A.

5.1 Verification with Semantic Reasoning

As described in Section 3, system models (in our case, mission plans) encoded in a DSL (in our case, YAML) are transformed by the CVC into ABox assertions. A semantic reasoner for OWL+SWRL verifies the consistency of the generated ABox with the TBox defined previously by domain experts. Inconsistencies signify an invalid mission plan. For example, an asset that does not execute at least one kinetic action would be inconsistent with the definition of class `Asset` presented in Section 4.1. The following OWL code specifies a valid ABox individual `H1` derived by the CVC from the declaration of `H1` in Mission A:

```
Individual: H1                                           1
  Types: Hummingbird                                     2
  Facts: hasAction TPSA1 and hasAction TPSA2             3
```

Once the OWL+SWRL engine establishes the consistency of the ABox with the TBox, it proceeds to generate a variety of inferences needed for eventual synthesis of the DTMC and PCTL artifacts. For example, the engine reasons 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 the engine as a *direct threat area action* (DTAA). This inference eventually triggers the synthesis of PRISM code that calculates a *risk acceptability factor* (RAF) for the threat area incursion that contains the inferred DTAA. RAF values quantify the risk associated with threat area incursions, where an optimal RAF value of one indicates a potentially risk-free incursion. The numerator of the RAF represents the duration of concurrent execution between sensor actions and DTAAs during a threat area incursion, while the denominator represents the aggregate duration of all DTAAs executed during the incursion. Thus, any time spent not performing sensor actions contributes to the *unjustifiable* risk of the threat area incursion.

RAF values computed in the synthesized PRISM code are verified against a RAF threshold value included in synthesized mission properties. For example, the PCTL property `P=? [ F raf>0.6 ]` queries the probability that the specified RAF value exceeds 60 percent. Like the probability values described in Section 4.3, these threshold values were chosen arbitrarily for our research and thus eventually would be calculated from real-world data.

5.2 Classification with Prolog

Some OWL-based inferences are transformed by the CVC into Prolog facts. Using the facts inferred for kinetic actions, Prolog classifies each kinetic action with respect to the relationships that exist between it and other kinetic actions. For example, a kinetic action that is the last to be executed by an asset and has as precondition at least one cross-cutting kinetic action is classified by Prolog as a `terminal_constrained_observer` according to the following inferred fact (which uses rule `terminal` presented in Section 4.2):

```
terminal_constrained_observer(X) :-                      1
  constrained_observer(X),                               2
  terminal(X).                                           3
```

The classification of a kinetic action affects the synthesis of the PRISM module for the asset to which that action is assigned. For example, the classification of action TPSA3 in Mission A would affect the PRISM module synthesized for asset H2. The asset module constructs that would be affected include the action label of the guarded command synthesized for TPSA3 and the action label of the last guarded command of the module.

Classifications of kinetic actions also trigger the synthesis of PCTL properties, such as a property requiring that the execution of a terminal_constrained_observer must not fail.

5.3 Synthesized Models and Properties

The CVC uses explicit and inferred domain knowledge to synthesize DTMC and PCTL artifacts from templates defined previously by domain experts. For example, the asset module template presented in Section 4.3 is used to synthesize one module for each asset in Mission A. The following PRISM code specifies the module synthesized for 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
```

Suppose that action TPSA4 is inferred to be a threat area action and thus part of a threat area incursion (as described in Section 3.2). The OWL+SWRL engine would then further infer action TPSA4 to be a DTAA (since TPSA4 initiates the incursion). Asset H2 consequently would be 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 would synthesize an asset survivability module to calculate the probability of survival for H2 (as described in Section 4.3). The following PRISM code specifies the synthesized survivability module for 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
```

At the conclusion of the synthesis process, the DTMC artifact that models Mission A is verified against a set of synthesized PCTL properties, such as the following:

```
P=? [ F d2=0 & d4=0 & !a2d & raf2>0.6 ]                1
```

This property contains variables d2 and d4, which denote the durations of TPSA2 and TPSA4, respectively; variable a2d, described above; and variable raf2, which represents the RAF for the threat area incursion executed by asset H2. The property need not constrain 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, and so on.

PRISM verifies the above property and thereby computes the probability of success for Mission A to be 0.299, with raf2 calculated to be 0.833.

5.4 Implementation

For our prototype implementation of cascading verification, the ontology described in Section 4.1 and Section 4.2 has been implemented in OWL and SWRL, with OWL+SWRL reasoning handled by Pellet, a sound and complete OWL reasoner [28]. The Prolog rules and Prolog reasoning described in Section 4.2 are handled by SWI-Prolog, an open source implementation of Prolog [29]. The CVC is implemented in Ruby and includes the preprocessing of YAML specifications, the synthesis of PRISM models and properties, and the coordination of calls to Pellet and SWI-Prolog. YAML was chosen as the basis for the DSL because of its compatibility with Ruby and the ease with which it can be preprocessed into OWL ABox assertions. Although integration, testing and extension of these components is still ongoing, the implementation is sufficiently complete to enable an evaluation of cascading verification, which we present in the following section.

6. PRELIMINARY 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 start validating this assertion, we demonstrate in this section that our prototype implementation of cascading verification simplifies the verification of UAV mission plans and augments PRISM's verification capabilities. Ultimately, we aim to show that our prototype benefits mission developers by improving the correctness of UAV mission specifications. We also discuss 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 infeasible at this stage of our research to evaluate our prototype with a full practitioner study, we opted instead for a metrics-based evaluation of 58 mission plans that are based on real-world missions developed by DARPA and DRDC [12, 13]. We evaluated our prototype 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 has, respectively, 3.127 and 4.490 times the LOC and tokens of the YAML models, with standard deviations of 0.524 and 0.954. These results provide some preliminary evidence of a non-trivial reduction in the effort required to produce mission models and properties.

We observe that five of the 58 mission specifications caused the CVC to make use of constructs for tactical missions (described previously in Section 4.1). These tactical mission plans generated PRISM code that was on average, respectively, 3.933 and 5.992 times the YAML LOC and tokens, with standard deviations of 0.240 and 0.592. Because the effort of synthesizing PRISM code is proportional to its LOC and tokens, tactical mission plans provide added value for mission planners. This suggests that, with respect to standalone and tactical missions, the utility of our prototype is proportional to the threat level associated with any given

mission plan. More broadly, the increased LOC and tokens suggest that the utility of cascading verification may be proportional to the amount of automated reasoning required to synthesize PRISM artifacts, a conclusion that justifies our motivation to augment model checking with formalized domain knowledge.

6.2 Effectiveness

Because a LOC- and token-based analysis cannot account fully for the complexities of the PRISM language, such analysis offers limited insight into the inherent complexity of model and property specifications. We investigate complexity further by considering some *behavioral modeling errors* specific to the PRISM language that can be eliminated by the automated synthesis of PRISM artifacts. Behavioral modeling errors include variable declarations with incorrect values; incorrect or missing command actions; incorrect command probabilities; and incorrect command updates. These errors are significant—perhaps more so than the errors uncovered during the model checking process—because they can cause PRISM to verify erroneous mission plans and thereby mislead mission planners.

We also consider mission specification errors that are beyond the scope of PRISM's verification capabilities. These *domain-specific errors* are detected either by Pellet or by 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:

- *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 the corresponding data properties in CEMO; for example, the maximum allowed endurance of a Hummingbird is 1200 timesteps.

- *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 specified in CEMO; for example, CEMO specifies that every `Hummingbird` 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 in CEMO; 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.

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

6.3 Probabilistic Verification

Finally, we consider PRISM's ability to verify UAV mission plans. 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 either 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 greater degree than is evidently possible by using PRISM on its own.

Along with the complexity of the UAV domain and the real-world DARPA and DRDC mission scenarios used for this evaluation, the above results suggest that cascading verification can be beneficial to other application domains. The portability of our method is supported by the general-purpose nature of its constituent technologies. Presently, we cannot extend this argument to the *connections* that link those technologies for cascading verification, although our attempt to isolate domain-specific processing in a preprocessor facilitates the portability of the connections. But cascading verification can be considered an extension of semantic model checking methods, which employs identical or comparable constituent technologies. The successful application of those methods to the Web service domain thus further supports the portability of cascading verification.

We note that this evaluation is preliminary. Further work is required to determine the ability of our prototype 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, as discussed in Section 1, has been studied exclusively in the context of the Web service domain. We discuss the body of results in semantic model checking below, but to the best of our knowledge, the work presented in this paper is unique, because it addresses limitations of existing semantic model checking techniques and applies the resulting improved method to a novel domain.

Narayanan et al. encode Web service capability descriptions and behavioral properties with DAML-S (an ontology encoded in DAML+OIL for describing Web services) and Petri net formalisms, respectively [3]. For a given Web service, their approach generates a Petri net corresponding to the DAML-S description of the service. The resulting Petri net is used by KarmaSIM—a modeling and simulation environment—to apply various analyses, including reachability analysis and deadlock detection.

The model checking algorithm of Di Pietro et al. uses a DL-based ontology to formalize the Web service domain [4]. The behavior of a Web service is modeled as a *state transition system* (STS), while behavioral requirements are encoded with CTL. Both STS and CTL are extended with semantic annotations. For a given Web service, their algorithm generates a finite STS corresponding to the annotated description of the service. The resulting model is verified with model checking. The same algorithm has been 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]. Their approach generates PRISM models from OWL-S descriptions of Web services. The resulting model is verified with PRISM. Ankolekar et al. map OWL-S models to PROMELA, and the PROMELA models are then verified with SPIN [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 et al. map OWL-S models to 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 most comparable to the work described above by Oghabi et al. Similarities include the motivation to verify stochastic behavior and the synthesis of PRISM models from domain knowledge encoded in OWL. 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 models. Inferred knowledge is utilized in the work described above by Narayanan et al., Di Pietro et al. and Boaro et al., but that other work is concerned exclusively with the verification of Web services and does not address the expressive and reasoning limitations that constrain OWL.

8. CONCLUSIONS AND FUTURE WORK

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

We have identified several promising directions for future work. First, the inferences performed by the CVC currently flow in a unidirectional fashion, with Prolog facts derived from the knowledge encoded in OWL+SWRL. While conceptually and practically appealing, this approach prevents the reasoning process from refining earlier inferences based on later inferences, and it 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 domain knowledge base, including CEMO, the Prolog rulebase, and the DTMC and PCTL templates. These artifacts should be extensible to reflect changes in domain knowledge. Extensions should in turn be verified to ensure that domain knowledge remains consistent across the entire knowledge base. Ideally, this verification of such evolutionary changes would be automated.

Finally, we intend to further evaluate our method and prototype by enhancing the sophistication of the mission specification language and domain model presented in this paper, and we intend to explore other domains in order to confirm the generality of our method.

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

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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