Cascading Verification:

An Integrated Method for Domain-Specific Model Checking

by

Fokion Zervoudakis

Submitted for the degree of Doctor of Philosophy at

University College London

February 2014

I, Fokion Zervoudakis, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis.

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 thesis 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.

Acknowledgments

First and foremost, I would like to thank my supervisors, David S. Rosenblum and Anthony Finkelstein, for their support during the past three years. I am indebted to David for his help and patience; for providing me the freedom to pursue interesting research; and for considering with me over long Skype calls the finer points of OWL-LP integration in the context of domain-specific and probabilistic model checking. Anthony's advice on authoring compelling narratives that clarify complex research, and his encouragement and strategic guidance, have been invaluable for my development as a researcher.

I have been fortunate to visit the Nebraska Intelligent MoBile Unmanned Systems (NIMBUS) Lab at the University of Nebraska-Lincoln (UNL). I would like to thank Sebastian Elbaum, co-founder of NIMBUS, for working with me to focus the research in this thesis, and for his contribution throughout my PhD.

I would also like to thank Emmanuel Letier for his insightful feedback during my first and second year vivas. And I would like to thank faculty and staff at the Department of Computer Science at University College London (UCL) for providing me with the resources to develop this thesis. In particular, I would like to thank Dean Mohamedally, Donald Lawrence, Graham Roberts, Jens Krinke, John Dowell, Mark Harman, Philip Treleaven and Westley Weimer for numerous valuable conversations. Last but not least, I have had the pleasure of working with talented colleagues including Jan Grochmalicki, Michal Galas and Panagiotis Papakos at UCL, and Charlie Lucas at UNL.

Research for this thesis was sponsored by the European Office of Aerospace Research & Development (EOARD), a detachment of the Air Force Office of Scientific Research (AFOSR), under agreement number FA8655-10-1-3007. Any conclusions, findings, opinions and recommendations expressed in this thesis are those of the author and do not necessarily reflect the views of AFOSR or EOARD.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

The increasing complexity of *uninhabited aerial vehicle* (UAV) missions, which may involve the execution of sophisticated (swarming) tactics by semi- or even fully-autonomous UAVs, is overwhelming mission development personnel. The development process is supported technologically by advanced, academic and commercially-oriented, mission planning systems and simulation environments. Orbit Logic's UAV Planner, for example, is a commercial simulation environment that provides resource, and task order, definition and management; manual and automated planning; and interactive visualizations [1]. In the academic space, MissionLab from Georgia Tech is a tool for specifying and controlling multi-agent missions [2]. Nowak et al. have developed a system named SWARMFARE, which executes self-organizing swarm-based simulations to test search and destroy scenarios [3]. Multi-agent self-organization has also been used to model UAV swarms and thereby facilitate solutions to problems of multi-objective optimization [4, 5].

Mission optimization constitutes a multi-objective problem because UAV missions are characterized by multiple and conflicting properties that form complex, and poorly understood, associations. The problem of multi-objective optimization has been studied extensively with evolutionary algorithms [6]. With respect to UAV missions, evolutionary algorithms are used to form robust UAV-based air-to-ground communication networks [7], and to generate a set of optimized and coordinated flight routes for surveillance missions [8]. The weight configurations that prioritize self-organizing rules in SWARMFARE are evolved with a genetic algorithm [3]. Rosenberg et al. use a simulation environment to evaluate optimized air campaign mission plans generated by a *multi-objective evolutionary algorithm* (MOEA) [9]. Pohl and Lamont use a MOEA to optimize the concurrent routing of multiple UAVs to multiple targets [10].

The problem of UAV scheduling/routing has also been modeled as a minimum cost network flow problem [11]; a traveling salesman problem; and a dynamic programming problem [12]. Alver et al. implement a route planning algorithm based on the time-oriented nearest neighbor heuristic [13]. Kamrani and Ayani use a special-purpose simulation tool named S2-Simulator to generate flight routes for the surveillance of moving targets [14]. In other simulation-related research, Corner and Lamont use a parallel discrete event simulation to explore emergence in UAV swarms [15]. Lian and Deshmukh use *Markov decision processes* (MDPs) to model UAVs in a dynamic multi-agent system

where agents have to manage constraints, negotiate flight paths and avoid enemy positions in order to execute a set of goals [16]. Hamilton et al. emphasize the importance of valid simulation models, and propose a testbed to empirically validate simulations against historical data [17].

When integrated, planning and simulation software should ideally provide information superiority and, ultimately, competitive advantage [18]. But these systems are currently challenged by the size and complexity of the UAV domain [19, 20, 21]. The state space of any given mission is potentially enormous. Mission correctness is contingent on multiple factors including UAV flight trajectories, interoperability and conflict resolution capabilities; and a dynamic environment that encompasses variable terrain, unpredictable weather and moving targets. In this context, mission plans may contain errors that compromise mission correctness. We propose to support mission developers by analyzing UAV mission plans with software verification methods—and in particular, probabilistic model checking—that can detect mission-critical errors before real-world execution. To this end, the following thesis describes a novel method for domain-specific model checking called *cascading verification* [22], and the application of that method to the probabilistic verification of complex UAV mission plans.

1.1 Research Problem and Scope

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 [23].

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 [24]. 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 [25, 26]; Web service security requirements [27]; probabilistic Web services [28]; Web service interaction protocols [29]; and Web service flow [30]. Additionally, multi-agent model checking has been used to verify OWL-S process models [31].

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.

In summary, model checking is a prominent formal verification method. Contemporary modeling languages induce modeling complexity, which is exacerbated by the need to encode domain knowledge at relatively low levels of abstraction. Semantic model checking uses semantic reasoning over domain knowledge encoded in OWL to supplement model checking and thereby decrease specification costs. But expressive and reasoning limitations constrain OWL and, by extension, the potential of semantic model checking.

1.2 Thesis Contributions

This thesis presents four 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. The method uses a composite inference mechanism to reason about high-level system specifications and thereby synthesize low-level PRISM code for probabilistic model checking.

Second, we use domain-specific software development to structure formal verification for the UAV domain. Third, we implement a prototype system that exploits our method to verify mission plans. Fourth, we evaluate our prototype to demonstrate the utility of cascading verification in the context of a significant and novel application domain. The following sections elaborate these contributions.

1.2.1 Cascading Verification

We have designed 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 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) [32] and the Defence Research and Development Canada (DRDC) agency [33]. 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 [34], for the verification of UAV mission plans.

As an implementation of cascading verification, our prototype realizes 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 [35]).

1.2.2 Domain-Specific Modeling for the UAV Domain

When compared with general-purpose programming languages, DSLs provide increased expressivity and usability for software development in the context of specific application domains [36]. A DSL is defined by concrete syntax and an abstract syntax metamodel [37]. Abstract syntax specifies language concepts and their relationships, while concrete syntax specifies the notation that represents those concepts. *Domain-specific modeling* (DSM) is a model-driven software development process that uses DSLs to encode system aspects. Syntax-oriented DSM avoids the behavioral properties of DSL models and metamodels. Several proposals attempt to specify these properties and

thereby enable the verification of DSL-based systems with model checking and other formal analysis techniques.

OWL supports the development of domain-specific languages and systems by representing knowledge in a manner that is unambiguous for humans and computers [38, 39]. For example, OWL can be used to formalize domain knowledge that constitutes the DSL metamodel [39]. By encompassing OWL ontologies, cascading verification is a method with the facility to support (probabilistic) model checking for DSL-based systems. In this context, we exploit OWL to formalize a subset of the UAV domain. The resulting ontology underpins, and thereby constrains and structures, the concrete syntax of a YAML DSL. For each DSL-based mission specification, consistency between concrete syntax and the abstract syntax metamodel is enforced by a compiler prior to the synthesis of PRISM code.

Cascading verification encompasses a DSL to enhance the abstraction level of model and property specifications. Model builders use this DSL to encode system specifications for probabilistic model checking. If UAV mission specifications encoded with the YAML DSL are also scheduled for real-world execution (via some process that is outside the scope of our method), then our prototype implementation of cascading verification will in essence be supporting DSM for the UAV domain. This observation implies a link between mission and software development, thereby justifying to some extent our motivation to analyze complex missions with software verification methods.

1.2.3 Prototype Design and Implementation

To investigate the feasibility of cascading verification, we designed and implemented a prototype that uses our method to verify UAV missions. DSM structured the development of 58 mission plans that underpin both our research effort and the evaluation of our method and prototype. Mission plans are encoded with the concrete syntax of a bespoke YAML DSL, which was established for this project. The abstract syntax metamodel of that DSL forms part of a knowledge base that ultimately comprises semantic, rule-based and behavioral models, and a set of behavioral properties. Models and properties, which describe different aspects of the UAV domain, are encoded with OWL+SWRL, Prolog, and PRISM DTMC and PCTL templates. The reasoning methods supported by OWL, SWRL and Prolog are combined to form a composite inference mechanism that achieves OWL-LP integration.

1.2.4 Prototype Evaluation

With the evaluation of our prototype, we aim to demonstrate the utility of cascading verification in the context of a non-trivial application domain. To this end, the evaluation is primarily concerned with abstraction and effectiveness. Abstraction is analyzed by comparing the *lines of code* (LOC) and numbers of lexical tokens required to specify UAV missions in YAML against the LOC and tokens that constitute synthesized PRISM code. We analyze effectiveness by presenting errors that can only be effectively eliminated with

the automated synthesis of PRISM artifacts. We also consider the utility of the DTMC and PCTL artifacts synthesized by our prototype. Finally, we discuss the portability of cascading verification. While the evaluation presented in this thesis is preliminary, it is also a substantive analysis supported by two case studies (involving tactical and traffic surveillance missions) and 58 mission plans (involving mission characteristics borrowed from DARPA and DRDC).

1.3 Thesis Outline

The remainder of this thesis is structured as follows.

Chapter 2 presents background material on the technologies—including OWL+SWRL, Prolog, DTMC and PCTL—that constitute cascading verification. This chapter also provides an overview of the UAV domain and complex UAV missions.

Chapter 3 presents a high-level overview of cascading verification. This chapter also specifies Mission A, an example UAV mission that underpins the discussion in subsequent chapters.

Chapter 4 describes how domain knowledge can be encoded in OWL+SWRL, Prolog, and DTMC and PCTL templates. The application of these technologies is illustrated with respect to a running example (Mission A) and two case studies.

Chapter 5 describes our prototype implementation of cascading verification for the UAV domain by tracing verification from high-level system specifications, which are encoded in a domain-specific YAML dialect, to probabilistic model checking with PRISM. This chapter also presents the technologies and implemented components that constitute our prototype.

Chapter 6 evaluates the benefits afforded by our prototype, and the utility and portability of cascading verification.

Chapter 7 summarizes our contributions to semantic model checking and discusses directions for future work.

Appendix A presents a framework of equations for establishing the occurrence, and calculating the duration, of *threat area incursions* committed by UAVs.

Appendix B contains an ontology encoded in OWL+SWRL that formalizes a subset of the UAV domain. This ontology encompasses both generic mission concepts and specialized knowledge related to tactical and traffic surveillance missions.

Appendix C contains Prolog rules that augment the ontological domain knowledge encoded in OWL+SWRL.

Appendix D contains DTMC and PCTL templates encoded in the programming language Ruby that formalize probabilistic behavioral knowledge.

Appendix E contains a schema definition for the YAML DSL presented in this thesis.

Appendix F contains verification artifacts—including system specifications encoded in YAML, synthesized DTMC and PCTL artifacts, and PRISM output—for representative UAV mission plans.

Source code for the software presented in this thesis is available online at: https://github.com/fokionzervoudakis/mission-verification-framework

Chapter 2

Background

The research problem outlined in Section 1.1 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 [40]. OWL limitations motivate several contending extensions including SWRL, CARIN, $\mathcal{AL}$-log, DL-safe rules, $\mathcal{DL}$+log, and many others [41]. Hybrid knowledge representation systems that integrate OWL+SWRL and Prolog have also been proposed [42, 43, 44, 45, 46, 47, 48]. 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 MDP models [23]. 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.

The remainder of this chapter is structured as follows. Section 2.1 investigates the formal logics underpinning OWL+SWRL and Prolog to determine the advantages and limitations of each formalism, and their compatibility with each other. Section 2.2 presents model checking, probabilistic model checking, and the DTMC and PCTL formalisms. Section 2.3 provides an overview of the UAV domain and complex UAV missions with a focus on UAV *performance specifications*, *mission aspects* and *mission scenarios*. This chapter is summarized in Section 2.4.

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* [40]. DL concepts

and individuals are roughly comparable to classes and objects, respectively, in object-oriented programming, while roles are comparable to UML associations.

An interpretation $\mathcal{I}$ is a model of a TBox $\mathcal{T}$ and an ABox $\mathcal{A}$ *iff* $\mathcal{I}$ satisfies all axioms and assertions in $\mathcal{T}$ and $\mathcal{A}$, respectively. More formally, an interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ comprises a non-empty set $\Delta^{\mathcal{I}}$ (the domain of $\mathcal{I}$) and a function $\cdot^{\mathcal{I}}$ that maps:

- every concept A_i to a subset $A_i^{\mathcal{I}}$ of $\Delta^{\mathcal{I}}$ ($A_i^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$);
- every role R_i to a binary relation $R_i^{\mathcal{I}}$ over $\Delta^{\mathcal{I}}$ ($R_i^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$);
- and every individual a_i to an element $a_i^{\mathcal{I}}$ of $\Delta^{\mathcal{I}}$ ($a_i^{\mathcal{I}} \in \Delta^{\mathcal{I}}$).

DLs support inferences that deduce the logical implications of ontological axioms with respect to concept *satisfiability*, *subsumption*, *equivalence* and *disjointness* [49]. TBox-supported inference services can be formalized as follows:

- a concept C is satisfiable with respect to a TBox $\mathcal{T}$ *iff* there exists some model $\mathcal{I}$ of $\mathcal{T}$ such that $C^{\mathcal{I}} \neq \emptyset$;
- C is subsumed by a concept B with respect to $\mathcal{T}$ *iff* $C^{\mathcal{I}} \subseteq B^{\mathcal{I}}$ for all $\mathcal{I}$ of $\mathcal{T}$;
- C is equivalent to B with respect to $\mathcal{T}$ *iff* $C^{\mathcal{I}} = B^{\mathcal{I}}$ for all $\mathcal{I}$ of $\mathcal{T}$;
- C is disjoint from B with respect to $\mathcal{T}$ *iff* $C^{\mathcal{I}} \cap B^{\mathcal{I}} = \emptyset$ for all $\mathcal{I}$ of $\mathcal{T}$.

OWL is an ontology specification language based on the modern description logic $\mathcal{SHOIN}(\mathbf{D})$ [40]. 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 [41]; nor the self-referential relationship that references an individual to itself [50]. SWRL addresses some of these limitations by extending OWL with Horn-like rules [51]. But OWL+SWRL cannot reason effectively with negation [41, 52]. Problems that are intractable in OWL+SWRL can be addressed with the programming language Prolog [41, 53].

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.

The decision to augment OWL+SWRL with Prolog is in part a consequence of Prolog's ability to support *negation as failure*, an inference rule that deduces falsity from the absence of truth. Negation as failure is related to the *closed world assumption* (CWA), which presumes a statement to be false unless that statement is known to be true. CWA is seemingly incompatible with the *open world assumption* (OWA) underpinning OWL+SWRL [41]. Contrary to CWA, OWA presumes a statement to be true unless that statement is known to be false. OWL+SWRL deficiencies with respect to negation derive from OWA, which is closely related to FOL. As described in the preceding paragraph, FOL underpins both OWL+SWRL and Prolog. The mechanism that enables Prolog to support CWA reasoning in the context of FOL is beyond the scope of our research.

This thesis 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 [41]. Our exclusive objective is to support domain-specific probabilistic model checking by combining well established logic systems [41, 54]. We propose to achieve this objective via *loose* integration, whereby DL and LP components are connected through a minimal interface [55].

2.2 Probabilistic Model Checking

As an established formal verification method, model checking has been applied to the analysis of hardware—including self-timed sequential circuits, a synchronous pipeline circuit and a bus adapter—and software [23]. Software analysis with model checking encompasses communication protocols including the ISDN User Part protocol, the IEEE Futurebus+ standard and the Needham-Schroeder protocol; and safety-critical systems including NASA's Mars Pathfinder and Deep Space 1 spacecraft. Traditional model checking is extended by probabilistic model checking, a method that verifies the behavioral properties of systems affected by stochastic processes [23, 56]. The verification of these *probabilistic systems* accommodates a flexible notion of correctness that contrasts with the absolute correctness verified by traditional model checking [23].

In order to model stochastic 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. One of several formal models for probabilistic model checking, 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.

Figure 2.1 uses a directed graph, or digraph, to illustrate the (discrete-time) Markov chain for a six-sided die simulated via a fair coin. Nodes and edges in the digraph represent states and transitions between those states, respectively. Edges from states s to s' are labeled with transition probabilities in the interval $[0, 1]$ if and only if $P(s, s') > 0$. Reflexive edges indicate terminating states.

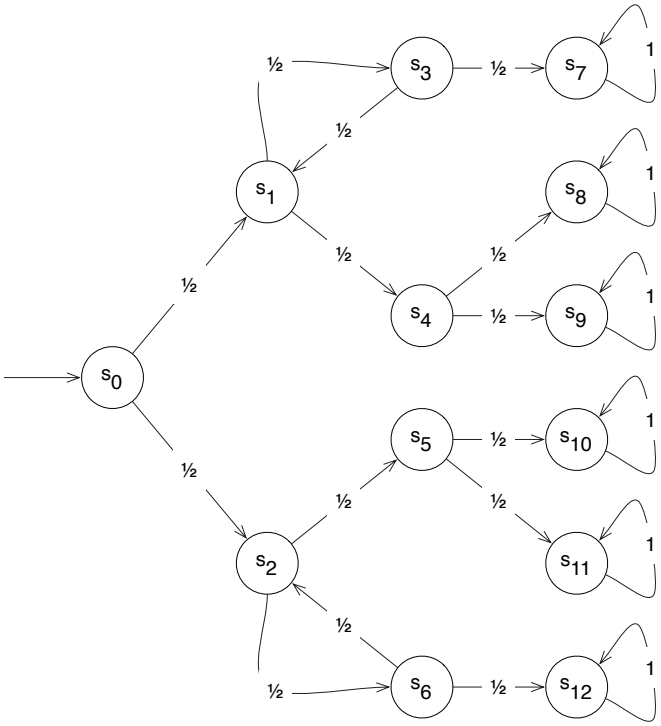

Figure 2.1: The (discrete-time) Markov chain for a six-sided die simulated via a fair coin [23].

The DTMC in Figure 2.1 comprises an initial node s_0—such that $\iota_{init}(s_0) = 1$ and $\forall s \neq s_0 : \iota_{init}(s) = 0$—and twelve states $S = \{s_1, \ldots, s_{12}\}$. Six inner states $\{s_1, \ldots, s_6\}$ and six terminating states $\{s_7, \ldots, s_{12}\}$ represent the tossing of a coin and possible die outcomes, repsectively. A Markov chain path, which can be rendered as a directed path in the underlying digraph, is a non-empty sequence of states $\pi = s_0\ s_1\ s_2 \cdots \in S^\omega$ such that $\forall i \geq 0 : P(s_i, s_{i+1}) > 0$.

For a given DTMC, the transition probability function $\mathbf{P}$ and the initial distribution ι_{init} can be represented, respectively, by a transition matrix $(\mathbf{P}(s, s'))_{s \in S}$ and an initial distribution vector $(\iota_{init}(s))_{s \in S}$. The matrix specifies for each state $s \in S$ the probability $\mathbf{P}(s, s')$ of transitioning from s to s' in a discrete time-step. The vector specifies for each state $s \in S$ the probability $\iota_{init}(s)$ that the system evolution begins in s. Figure 2.2 provides a partial transition matrix and a partial initial distribution vector for the DTMC in Figure 2.1.

Probabilistic model checking can be used to verify both quantitative and qualitative DTMC properties. The former constrain probabilities to specific thresholds, while the

$$\mathbf{P} = \begin{array}{c} \\ s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ \vdots \end{array} \begin{array}{c} \begin{array}{cccccccccc} s_0 & s_1 & s_2 & s_3 & s_4 & s_5 & s_6 & s_7 & s_8 & s_9 & \cdots \end{array} \\ \begin{bmatrix} 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots \\ 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & \cdots \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & \cdots \\ 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & \cdots \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \end{array} \quad \iota_{init} = \begin{array}{c} \\ s_0 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ \vdots \end{array} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ \vdots \end{bmatrix}$$

Figure 2.2: The partial transition matrix and partial initial distribution vector for a six-sided die simulated via a fair coin, as illustrated in Figure 2.1.

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 and dangerous) military missions, UAVs are increasingly relied upon to perform agricultural, scientific, industrial and law-enforcement tasks over civilian airspace [57, 58, 59].

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 [60]. Figure 2.3 illustrates various levels of UAV autonomy, ranging from remotely guided drones to fully autonomous swarms, which correspond to levels of automation originally proposed by Sheridan and Verplank [3]. 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 [61]. These factors combine to form a complex stochastic state space that motivates the probabilistic verification of UAV mission plans.

This section provides a concise overview of what is an expansive application domain. Chapter 4 explores a subset of that domain in greater detail.

2.3.1 UAV Performance Specifications

UAVs can be classified with respect to their performance specifications, and the mission requirements that those specifications are meant to address [63]. UAV performance specifications include the following:

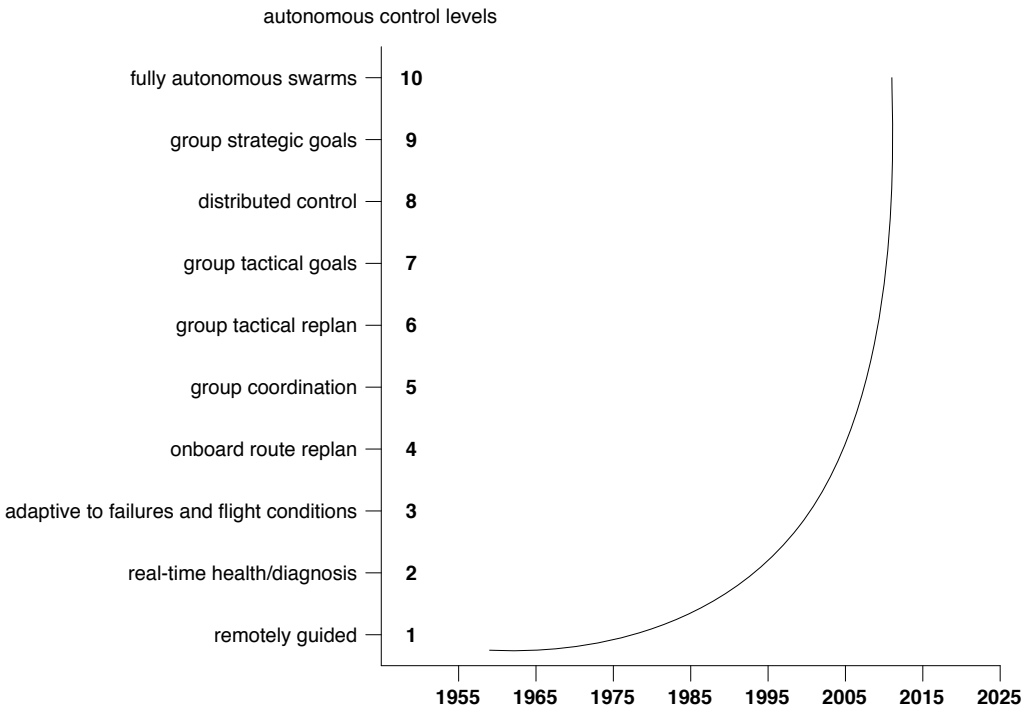

Figure 2.3: A hierarchy of UAV autonomy levels [33, 62].

- *ceiling* (measured in meters), "an aircraft's maximum pressure height" [64];
- *cost*;
- *endurance* (hours), "the length of time an aircraft can stay in the air without refueling" [64];
- *engine type*;
- *payload* (kilograms), "the load carried by an aircraft" [64];
- *power* (kilowatt), "the propulsive power needed to produce thrust" [64];
- *range* (kilometers), "the maximum distance an aircraft can fly on a given amount of fuel" [64];
- *speed* (kilometers per hour);
- *weight* (kilograms);
- *wing loading* (kg/m^2), "the weight of an aircraft per unit wing area" [64];
- *wing span* (meters), "a measurement from the tip of one wing to the tip of the other wing" [64].

Table 2.1 lists the performance specifications for various active (at the time of writing) military- and commercial-grade UAVs [63, 65, 66, 67]. The range of values encompassed

UAV	*manufacturer*	*ceiling* (m)	*endurance* (hr)	*payload* (kg)	*range* (km)	*speed* (km/h)	*weight* (kg)	*wing loading* (kg/m^2)	*wing span* (m)
BQM-147 Dragon	BAI Aerosystems	3,048	3	11	148	160	41	22	2
Crecerelle	SAGEM	3,353	6	35	59	246	120	9	3
Heron (Machatz-1)	Israel Aerospace Industries	10,000	40	227	3,300	207	1,087	70	17
Luna X-2000	EMT Penzberg	4,000	4	10	80	160	40	40	4
MQ-1 Predator	General Atomics	7,920	20	600	740	217	1,020	89	15
MQ-8 Fire Scout	Northrop Grumman	6,096	6	90	400	231	1,159	69	9
MQ-9 Reaper	General Atomics	15,200	24	3,000	1,500	405	4,500	83	20
RQ-2 Pioneer	AAI Corporation	4,572	5	64	373	175	125	34	5
RQ-4 Global Hawk	Northrop Grumman	20,000	30	900	22,000	636	11,600	199	35
RQ-7 Shadow	AAI Corporation	4,270	5	75	125	204	149	79	4
RQ-11 Raven	AeroVironment	4,267	4	17	100	204	84	57	3
RQ-14 Dragon Eye	AeroVironment	305	1	0	5	65	2	5	1
RQ-15 Neptune	DRS Technologies	2,440	4	10	75	156	36	74	2
AR.Drone 1.0	Parrot	6–50	0.2	0	0.05–0.335	18	0.38–0.42	n/a	n/a
Falcon 8	AscTec	2,500	0.27–0.3	0.5	0.15	10.8–54	0.8	n/a	n/a
X6	Draganfly	2,438	n/a	0.5	n/a	50	1	n/a	n/a

Table 2.1: Performance specifications for various military- and commercial-grade UAVs (top and bottom, respectively) including the established MQ-1 Predator and MQ-9 Reaper from General Atomics; and the MQ-8 Fire Scout helicopter from Northrop Grumman.

specification	*category*	*range*	*UAV*
ceiling (m)	low	<1000	RQ-14 Dragon Eye
	medium	1000–10,000	MQ-1 Predator
	high	>10,000	RQ-4 Global Hawk
endurance (hr)	low	<5	RQ-14 Dragon Eye
	medium	5–24	MQ-1 Predator
	high	>24	RQ-4 Global Hawk
range (km)	low	<100	RQ-14 Dragon Eye
	medium	100–1500	MQ-1 Predator
	high	>1500	RQ-4 Global Hawk
weight (kg)	micro	<5	RQ-14 Dragon Eye
	light	5–50	RQ-15 Neptune
	medium	50–200	RQ-11 Raven
	heavy	200–2000	MQ-1 Predator
	super heavy	>2000	RQ-4 Global Hawk
wing loading (kg/m^2)	low	<50	RQ-14 Dragon Eye
	medium	50–100	MQ-1 Predator
	high	>100	RQ-4 Global Hawk

Table 2.2: Performance specification categories with their respective ranges, and illustrative UAV classifications.

by each performance specification can be divided into categories [63]. Table 2.2 uses these categories to classify a subset of the UAVs listed in Table 2.1.

UAVs can also be classified with respect to the following mission aspects, which categorize mission requirements.

- *Aerial delivery and resupply* requires UAVs to supply special forces teams covertly and precisely with small quantities of cargo including batteries, water and leaflets for psychological operations [62].
- *Combat* requires highly maneuverable *unmanned combat aerial vehicles* (UCAVs) to engage in both air-to-air and air-to-surface combat [63].
- *Intelligence, surveillance, target acquisition and reconnaissance* (ISTAR) requires UAVs to enhance situational awareness by collecting battlefield information.
- *Multi-purpose* requires UAVs to conduct armed reconnaissance against critical and perishable targets.
- *Radar and communication relay* requires *airborne communication nodes* (ACNs) to ensure information superiority by extending and enhancing tactical intra-theater communications [62].

- *Vertical take-off and landing* (VTOL) requires UAVs to generate sufficient downward thrust to takeoff, hover and land within very limited space.

The performance specification categories that classify each UAV address a particular set of mission aspects [63]. For example, the MQ-1 Predator is classified as a medium altitude (1000–10,000 meters), medium endurance (5–24 hours), medium range (100–400 km) and heavy weight (200–2000 kg) UAV. These classifications determine the Predator to be highly-desirable for ISTAR and multi-purpose missions, and unsuitable for all remaining mission aspects. Table 2.3 classifies the military-grade UAVs listed in Table 2.1 with a rating scale of zero to four, where zero indicates the inability of a UAV to perform a specific mission aspect; and values in the range of one to four indicate the degree of compatibility, from lowest to highest, respectively, between performance specifications that parameterize UAVs and the performance specifications required by mission aspects.

	delivery	*UCAV*	*ISTAR*	*multi-purpose*	*ACN*
BQM-147 Dragon	0	0	0	0	0
Crecerelle	0	0	0	0	0
Heron (Machatz-1)	0	0	3	0	0
Luna X-2000	0	0	1	0	0
MQ-1 Predator	0	0	3	4	0
MQ-8 Fire Scout	0	0	0	0	2
MQ-9 Reaper	0	0	0	0	0
RQ-2 Pioneer	0	0	1.5	0	0
RQ-4 Global Hawk	0	0	4	0	0
RQ-7 Shadow	0	0	1.5	0	0
RQ-11 Raven	0	0	0	0	0
RQ-14 Dragon Eye	0	0	1	0	0
RQ-15 Neptune	0	0	1	0	0

Table 2.3: UAV classifications with respect to the degree of compatibility between performance specifications that parameterize UAVs and the performance specifications required by (abbreviated) mission aspects. Compatibility is rated with a scale of zero to four, where zero indicates the inability of a UAV to perform a specific mission aspect.

2.3.2 UAV Mission Hierarchy

We distinguish mission aspect from mission; the former is a conceptualization of related mission requirements, the latter a structured collection of interrelated tasks. Table 2.4 presents a tabular hierarchy of military and commercial UAV missions [68]. High-level missions, which correspond partially to the mission aspects presented in Section 2.3.1, include the following:

- *communication*, the relay of voice and data between units, and from units to a higher command;
- *drones*, the imitation of fighter aircraft or other objects for the purposes of training or (enemy) deception;
- *extraction*, the removal (including search and rescue) of personnel and cargo from a specified target;
- *insertion*, the delivery of lethal and non-lethal payloads (for example, emergency supplies) to a specified target;
- *intelligence*, the accumulation, analysis and dissemination of enemy, terrain and weather information in areas of interest or operation;
- *reconnaissance*, the exploration or inspection of a specific area for the purpose of information gathering;
- *surveillance*, the (often clandestine) monitoring of behavior and activities;
- *transport*, the movement or transfer of personnel and cargo between two locations.

Table 2.4 divides the *intelligence, surveillance, and reconnaissance* (ISR) mission space, which is itself a subset of the ISTAR mission aspect, into *intelligence/reconnaissance* and *surveillance* missions. When considered from a different perspective, airborne ISR missions can also be divided into mutually exclusive mission segments including *standoff*, which requires ISR platforms to respect the sovereign airspace of other nations; *over flight*, which authorizes ISR platforms to perform low-risk violations of sovereign airspace, with or without consent from the nation whose airspace is being violated; and *denied*, which exposes ISR platforms to a potentially hostile airspace [62].

UAV missions in general can be divided into planning, management, and re-planning segments to identify functions that will be assumed by human operators [68]. For example, drone mission segments comprise the following tasks:

- *mission planning*, the use of 1) a scheduling mechanism to plan health and status reports, 2) threat area and no-fly zone information to designate the area of deployment, and 3) a decision support mechanism to designate loiter locations;
- *mission management*, the use of indicators to monitor the health, status and progress of a UAV;
- *mission re-planning*, the use of path planning to re-designate deployment areas.

Given these tasks, drone missions require human operators to supervise path planning, and to monitor the health and status of UAVs. Transport missions extend drone missions by requiring operators to monitor the health and status of passengers. Insertion missions, which are more demanding, require operators to supervise path planning;

level 1	*level 2*	*level 3*
drones		
	decoy	
	target practice	
communication		
extraction		
insertion		
	electronic warfare	
		electronic attack
		electronic protection
	payload delivery	
		lethal
		non-lethal
intelligence/reconnaissance		
	BDA	
	mapping	
	target acquisition	
		dynamic target
		static target
	target designation	
transport		
	cargo	
	passengers	
surveillance		
	geospatial surveillance	
		dynamic target
		static target
	listening	
	NBC sensing	

Table 2.4: A tabular UAV mission hierarchy, which includes *battle damage assessment* (BDA) and *nuclear, biological and chemical* (NBC) sensing missions.

monitor the health and status of the UAV; monitor the status of weapons; identify targets; allocate and schedule resources; and negotiate with, and notify, other stakeholders. Nehme et al. provide a comprehensive discussion on UAV missions, and the function of human operators with respect to those missions [68].

We have presented UAV missions from multiple perspectives, which support the analysis of mission scenarios at different levels of granularity (from high-level mission objectives with tactical and political implications to low-level tasks performed by human

operators). Section 2.3.3 and Section 2.3.4 introduce two illustrative mission scenarios.

2.3.3 DARPA Mission Scenario

UAVForge was a collaborative initiative by DARPA and the Space and Naval Warfare Systems Center Atlantic (SSC Atlantic) [32]. The aim of the initiative was to crowd-source the design and development of a remotely operated micro-UAV that could be transported in a rucksack by a single person traveling on foot. Submitted UAVs were evaluated in the context of a mission scenario, which involved the observation of suspicious activities occurring in the vicinity of two nondescript urban buildings. Total mission time did not exceed five and a half hours including three hours for the observation task. UAVs were required to demonstrate the following capabilities:

- vertical take-off(s) from, and landing(s) to, different stationary locations;
- safe flight within an altitude window of 5–1000 feet, with up to 15 mile per hour winds and 40–100°F temperatures;
- point-to-point navigation within 500 feet of defined flight corridors;
- operation within an assigned airspace and avoidance of defined no-fly zones;
- (preferably autonomous) avoidance of static and (potentially) dynamic obstacles—including buildings, water towers, and trees—during approach into an observation area;
- landing in an area similar to a rooftop structure containing *heating, ventilation, and air conditioning* (HVAC) equipment, communications gear, satellite dishes and poles;
- identification of a vantage point—up to 2.0 miles beyond line-of-sight from the starting location—from which to conduct the observation task
- observation by landing on, adhering to, hanging over, and/or hovering above or below physical structures;
- real-time transmission of images or video depicting static and mobile items of interest located up to 100 feet from the observing UAV;
- data link frequency management and data transmission in compliance with rules and authorized frequencies stipulated, respectively, by the Academy of Model Aeronautics (AMA) and the U.S. Federal Communications Commission (FCC).

Figure 2.4 illustrates the UAVForge mission scenario, which underpins, and thereby lends credibility to, the 58 UAV mission plans developed for this project. These plans in turn support the development and evaluation of our method and prototype.

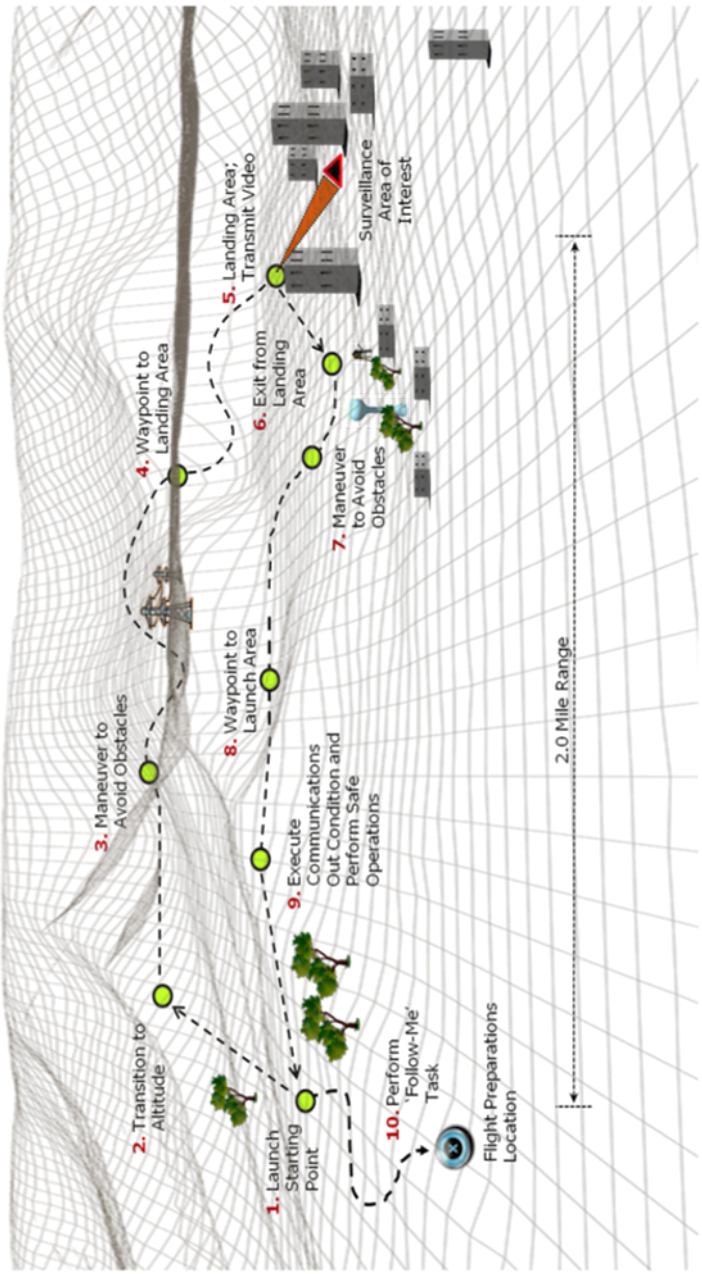

Figure 2.4: An illustration of DARPA's UAVForge mission scenario [32].

2.3.4 DRDC Mission Scenario

The DRDC produces mission scenarios to further knowledge on the intersection of UAV/UCAV devices and *intelligent adaptive interfaces* (IAIs), which are technologies for enhancing performance in complex socio-technical environments such as multi-UAV operations [69, 70]. Mission scenarios employ a suite of UAV platforms—including *medium altitude long endurance* (MALE) UAVs (for example, the MQ-1 Predator[1]), VTOL Tactical UAVs (VTUAVs), and mini UAVs—to evaluate the impact of IAI constructs on operations within the UAV domain in Canada [33]. We will focus on a mission scenario involving a *fishery patrol* (FISHPAT) and a *counter-drug operation* (CD OP), which are representative of domestic ISR missions undertaken by Canadian Forces in peacetime (additional domestic operations include disaster recovery, homeland security, long-duration law enforcement surveillance, and search and rescue).

The CD OP is coordinated from the Maritime Forces Atlantic (MARLANT) Headquarters at Canadian Forces Base (CFB) Halifax with support from HMCS Winnipeg. A *ship of interest* (SOI) is initially tracked by US Customs aircraft and a US Coast Guard vessel. In conjunction with this activity, HMCS Montreal, which carries four VTUAVs, is conducting a FISHPAT approximately 180 nautical miles south of Burin in Newfoundland; and HMCS Kingston, which carries two mini UAVs, is conducting a coastal patrol 150 nautical miles southwest of Burin. Mission correctness is complicated by several factors. Tasked assets operate remotely and at great distances from their home bases. Extended transit times and accurate, timely intelligence reports are therefore critical to mission planning, and asset positioning and availability. In addition, the *area of operation* (AOO) experiences weather conditions—including frequent and extensive fog, low cloud cover and precipitation—that limit the utility of *electro-optical* (EO) and *infrared* (IR) sensors.

A timeline of major mission events is divided into seven segments, which are deemed conducive to IAI experimentation. These segments identify periods of excessive operator workload resulting from "the simultaneous receipt of sensor data ... [transmitted by] multiple UAVs; dynamic re-tasking of UAVs; transfer of UAV control between agencies; and concurrent control of multiple UAVs." [33] Mission segments include the following (all times are Eastern Standard):

0210–0232 hrs: HMCS Montreal and Viper 01 (a MALE UAV) are re-tasked from a FISHPAT and a training exercise, respectively, to the CD OP. Viper 01 is vectored into position by a *mission control element* (MCE) located at CFB Greenwood. Alpha 51, a manned patrol aircraft, is also re-tasked from a training exercise in order to track the SOI.

0325–0702 hrs: HMCS Montreal maintains simultaneous control of two VTUAVs (Bingo 61 and Bingo 62) and receives *synthetic aperture radar* (SAR) imagery from Viper 01.

[1] In contrast to Youngson et al., Arjomandi classifies the MQ-1 Predator as a medium endurance UAV (see Table 2.2).

0750–0812 hrs: Ship board and airborne MCEs—for example, HMCS Kingston and Alpha 52, respectively—execute multiple UAV payload (i.e., EO, IR, and SAR) control transfers.

1110–1140 hrs: Alpha 52 controls, dynamically re-tasks and monitors sensor data from two UAVs (Mike 91 and Mike 92) while receiving SAR imagery from Viper 01. This involves Alpha 52 launching Mike 91, and assuming control of Mike 92 from HMCS Kingston.

1200–1432 hrs: HMCS Montreal controls and monitors sensor data from three UAVs (Bingo 63, Bingo 64 and Mike 92) while receiving SAR imagery from Viper 01. This involves HMCS Montreal launching Bingo 64 to replace Bingo 63, and assuming control of Mike 92 from Alpha 52.

1440–1525 hrs: Alpha 52, HMCS Kingston and HMCS Montreal transfer control of UAVs in conjunction with the surveillance of multiple targets.

1620–1700 hrs: UAV sensor operations and frequency of reporting accelerate in response to an increased operational tempo.

As with UAVForge, the DRDC scenario underpins mission plans that in turn support the development and evaluation of our method and prototype.

2.4 Summary

This chapter presents background material on the technologies that constitute cascading verification. We also provide a broad, concise overview of the UAV domain to contextualize, and convey the inherent complexity of, UAV missions. Some of the information in this overview is never exploited by our work; for example, our prototype implementation of cascading verification does not incorporate the performance specifications and mission hierarchy presented in Section 2.3.1 and Section 2.3.2, respectively. But this information is nevertheless pertinent because it has informed our research and development efforts. The scope of our prototype with respect to its application domain will be elaborated in subsequent chapters.

Chapter 3

Method Overview

Figure 3.1 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.

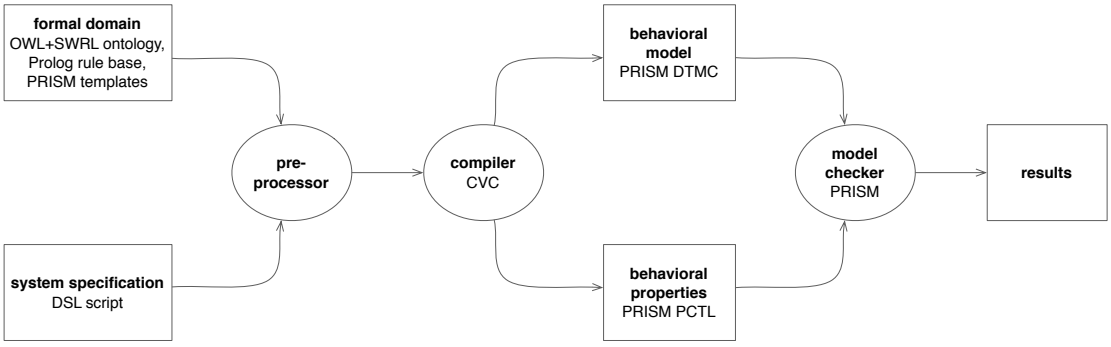

Figure 3.1: A high-level, domain-agnostic schematic of our method and prototype. Rectangular and oval shapes represent data and processes, respectively; and bold and normal text distinguishes method from prototype, respectively.

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 3.1 specifies Mission A, an example mission that is representative of the 58 mission plans developed for this project.

Mission A, which is illustrated in Figure 3.2, comprises two *Hummingbird* assets (lines 24–28 in Listing 3.1); 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

Listing 3.1: YAML code for Mission A

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

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).

With the exception of asset endurance, mission concepts specified in Listing 3.1 correspond directly to the elements illustrated in Figure 3.2. Asset endurance can be optionally omitted from mission specifications, and was therefore omitted for assets `H1` and `H2` in Mission A, because the default endurance for Hummingbird assets is specified in an OWL+SWRL ontology, which will be presented in Chapter 4. The following section describes the mechanism that integrates mission specifications with ontological knowledge.

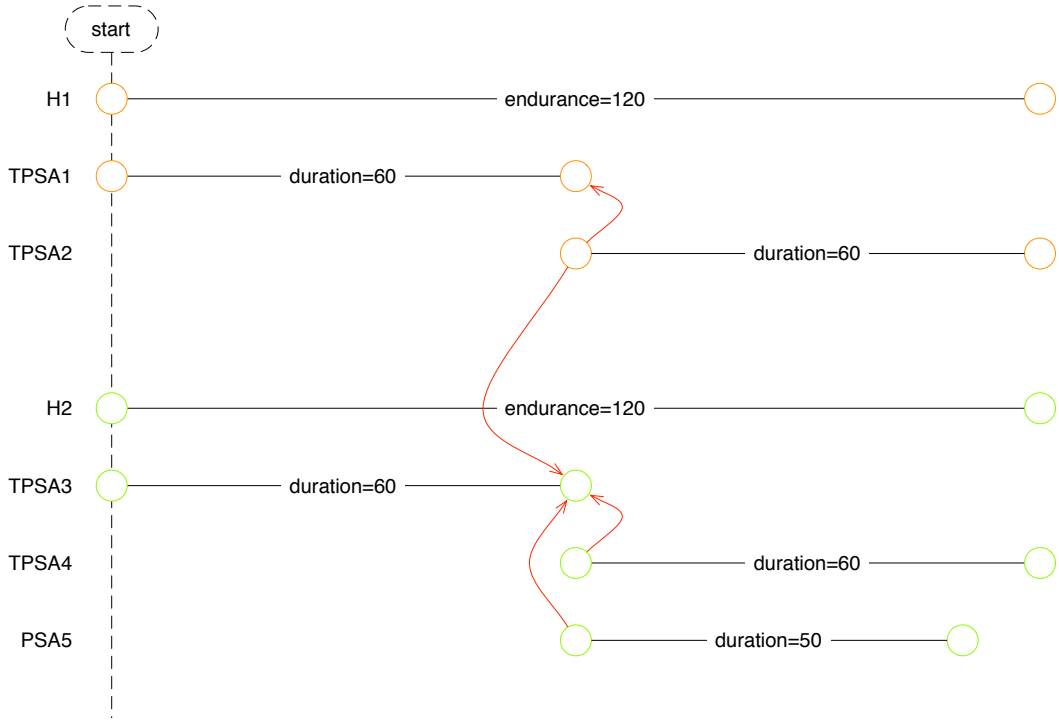

Figure 3.2: An illustration of Mission A, with red arrows representing action dependencies and horizontal lines representing the passage of time. Color coded circles delineate the operation and execution of assets and actions, respectively, and group actions and the assets to which those actions are assigned.

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 (these equations will be presented in Appendix A). Geographic information resulting from preprocessing is integrated with the generated ABox.

2. We use Pellet, a sound and complete semantic reasoner [71], 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

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

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 [72]—inputs 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).

In summary, cascading verification is a formal method that abstracts model checking for specific application domains. For a given domain of interest, experts in that domain implement a bespoke CVC once; and model builders use the implemented CVC to easily verify multiple system specifications.

We proceed to elaborate a subset of the UAV domain, which encompasses the concepts that constitute Mission A.

Chapter 4

Domain Modeling

This chapter describes the process of encoding 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 Section 3.1.

The remainder of this chapter is structured as follows. Section 4.1 uses OWL to formalize a subset of the UAV domain. The discussion is contextualized by case studies involving tactical and traffic surveillance missions, which are presented in Section 4.1.2 and Section 4.1.3, respectively. Section 4.2 uses SWRL to integrate complex relational structures into OWL, and Prolog to encode knowledge that can support effective reasoning with negation. Section 4.3 uses DTMC and PCTL formalisms to encode, respectively, probabilistic behavior and properties. DTMC and PCTL code is abstracted and presented in templates that support the synthesis of PRISM artifacts. Section 4.4 and Section 4.5 present related work and a summary of this chapter, respectively.

4.1 Semantic Modeling

With cascading verification, domain experts use OWL+SWRL 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. Domain concepts are integrated into a modular ontology architecture that supports flexible development and reuse [73, 74]. The ontology development process utilized prominent Semantic Web technologies including the Protégé ontology editor, the Pellet reasoner and the Semantic Web Stack, which comprises OWL+SWRL.

4.1.1 Building an OWL Ontology

Figure 4.1 illustrates CEMO's class hierarchy, where each OWL class represents a grouping of individuals with similar characteristics [75]. 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.) These concepts are accessible to all ontology modules that extend CEMO.

Figure 4.1: CEMO's class hierarchy as presented by the Protégé ontology editor.

The OWL code in Listing 4.1 uses *Manchester OWL syntax*—a human-friendly ontology representation language [76]—to formally defines class `Asset`, which is comprised in Mission A. Lines 2–5 specify four OWL properties, which are constructs for describing relationships. In particular, an OWL property with specified *domain* and *range* links individuals from the domain to individuals or data values from the range. The two main OWL property types are *object properties* and *datatype properties*. The former describe relationships between two individuals while the latter describe relationships between individuals and data values.

Listing 4.1: OWL code for class `Asset`

```
1 Class: Asset
2   SubClassOf: hasAction some KineticAction,
3     hasCostValue some xsd:integer,
4     hasEnduranceInSeconds some xsd:integer,
5     hasSpeedInKilometersPerHour some xsd:integer
```

Lines 1 and 2 in Listing 4.1 specify that every member of class `Asset` (the domain) must be associated with a member of class `KineticAction` (the range) via an instance of the object property `hasAction`. (For brevity, the reminder of this thesis will refer to *property instances* simply as properties.) Members of class `Asset` are also associated with datatype properties describing asset cost, endurance and speed (lines 3–5).

Class `Asset` is extended by class `NamedAsset`, which is in turn extended by classes `ARDrone` and `Hummingbird`. The latter classes represent quadcopter UAVs manufactured by Parrot USA and Ascending Technologies, respectively, that have informed our research. The OWL code in Listing 4.2 formally defines class `Hummingbird`, which is comprised in Mission A.

Lines 3–5 in Listing 4.2 associate members of class `Hummingbird` with three datatype properties—`hasCostValue`, `hasEnduranceInSeconds` and `hasSpeedInKilometersPerHour`. These properties are inherited from classes `NamedAsset` and, ultimately, `Asset`. Unlike class `Asset`, the ranges of the datatype properties that parameterize members of

Listing 4.2: OWL code for class `Hummingbird`

```
1 Class: Hummingbird
2   SubClassOf: NamedAsset,
3     hasCostValue some xsd:integer[>= 5000],
4     hasEnduranceInSeconds some xsd:integer[<= 120],
5     hasSpeedInKilometersPerHour some xsd:integer[<= 50]
6   DisjointWith: ARDrone
```

class `Hummingbird` restrict possible data values. Line 6 specifies that classes `ARDrone` and `Hummingbird` constitute *disjoint* sets of individuals. Consequently, and appropriately, a member of class `ARDrone` cannot also be a member of class `Hummingbird`.

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 3.1), 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). CEMO encodes preconditions with the object property `hasPrecondition`, which is formally defined in Listing 4.3.

Listing 4.3: OWL code for the object property `hasPrecondition`

```
1 ObjectProperty: hasPrecondition
2   Characteristics: Transitive
3   Domain: Action
4   Range: Action
5   InverseOf: isPreconditionTo
```

Line 2 in Listing 4.3 declares the object property `hasPrecondition` to be *transitive*. Transitivity is one of several *property characteristics* that can be used to qualify object properties. Given transitivity, if an action a is related via `hasPrecondition` to an action b, and b is related to an action c via the same property, then we can infer that a is related via `hasPrecondition` to c. In addition, `hasPrecondition` is inverted by the object property `isPreconditionTo` (line 5). The inverse relation between `hasPrecondition` and `isPreconditionTo` implies that if an action a is related via `hasPrecondition` to an action b, then b is related to a via `isPreconditionTo`. Figure 4.2 and Figure 4.3 illustrate transitivity and inversion, respectively.

Because preconditions associate actions with other actions, class `Action` constitutes both the domain and range of the object property `hasPrecondition` (lines 3 and 4, respectively, in Listing 4.3). Class `Action` is extended by class `KineticAction`, whose members are associated with a data property describing action duration. Class `KineticAction` is in turn extended by classes `HoverAction` and `TraversePathSegmentAction`. The OWL code in Listing 4.4 formally defines class `TraversePathSegmentAction`, which is comprised in Mission A. Lines 3–4 in Listing 4.4 specify that every

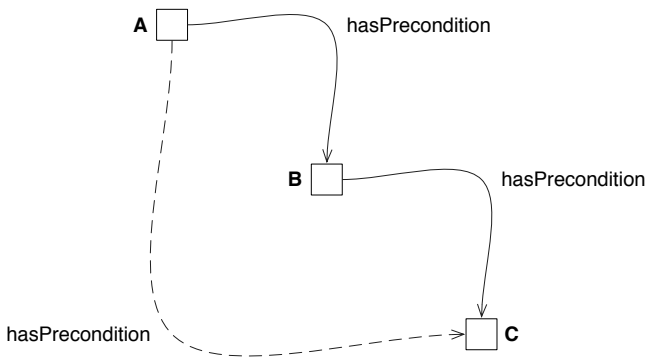

Figure 4.2: An example of the transitive property characteristic that qualifies the object property `hasPrecondition`. The dashed line represents an inferred relationship.

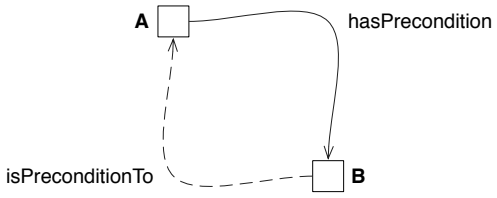

Figure 4.3: An example of inversion with respect to the object properties `hasPrecondition` and `isPreconditionTo`. The dashed line represents an inferred relationship.

member of class `TraversePathSegmentAction` must be associated via the object properties `hasStartPoint` and `hasEndpoint` with `Waypoint` individuals that designate the geographical start points and endpoints, respectively, of path segment traversal actions (hover actions are also designated geographically by waypoints).

Listing 4.4: OWL code for class `TraversePathSegmentAction`

```
1 Class: TraversePathSegmentAction
2   SubClassOf: KineticAction,
3     hasStartPoint some Waypoint,
4     hasEndpoint some Waypoint
5   DisjointWith: HoverAction
```

Class `Action` is also extended by class `SensorAction`, which is in turn extended by class `PhotoSurveillanceAction`. The OWL code in Listing 4.5 formally defines class `PhotoSurveillanceAction`, which is comprised in Mission A.

Listing 4.5: OWL code for class `PhotoSurveillanceAction`

```
1 Class: PhotoSurveillanceAction
2   SubClassOf: SensorAction,
3     hasDurationInSeconds some xsd:integer,
4     hasPrecondition only Action
```

The preceding code contains the keywords *some* (line 3) and *only* (line 4), which represent, respectively, existential and universal restrictions in OWL. With regard to object properties:

- 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 [77].
- 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.

Existential and universal restrictions, which can also be applied to datatype properties, are denoted in predicate logic by the existential ($\exists$) and universal ($\forall$) quantifiers, respectively.

4.1.2 Modeling Tactical Missions

During tactical missions, 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. The United States Department of Defense (USDOD) defines survivability as "the capability of a system ... to avoid or withstand a man-made hostile environment without suffering an abortive impairment in its ability to accomplish its designated mission." [78] To accommodate tactical mission requirements, we have developed a *complex tactical missions ontology* (Tactical-CEMO) that extends CEMO.

Figure 4.4 illustrates Tactical-CEMO's multiple inheritance class hierarchy; for example, class `DirectThreatAreaHoverAction` extends classes `HoverAction` and `ThreatAreaAction`. Classes describing tactical missions are highlighted in bold and thereby differentiate from classes encoded in CEMO. Used exclusively to support automated reasoning, tactical concepts are not available to mission developers via the YAML DSL.

We define class `ThreatArea` in Tactical-CEMO and specify that any waypoint related to a threat area be inferred, by Pellet or other semantic reasoners, a member of class `ThreatAreaWaypoint`. (The geographic information that relates waypoints to threat areas is determined by the CVC during preprocessing, as described in Chapter 3.) To enable the inference of class membership, class `ThreatAreaWaypoint` must be *defined* by domain experts using *necessary and sufficient* conditions [77]. We contrast necessary and sufficient conditions with *necessary* conditions, which describe all classes encoded in CEMO (as described in Section 4.1.1). These classes are known as *primitive*, and cannot be used to infer class membership. For example, because class `Asset` is described using only necessary conditions, an individual that is a member of class `Asset` must satisfy those conditions. However, class membership cannot be inferred for any (random) individual that satisfies the conditions describing class `Asset`. Figure 4.5 illustrates the type of reasoning supported by necessary conditions.

Unlike primitive classes, a *defined class*, which is a class comprising necessary and sufficient conditions, can be used to infer class membership. For example, because class `ThreatAreaWaypoint` is defined using necessary and sufficient conditions, class membership can be inferred for any (random) individual that satisfies those conditions. As

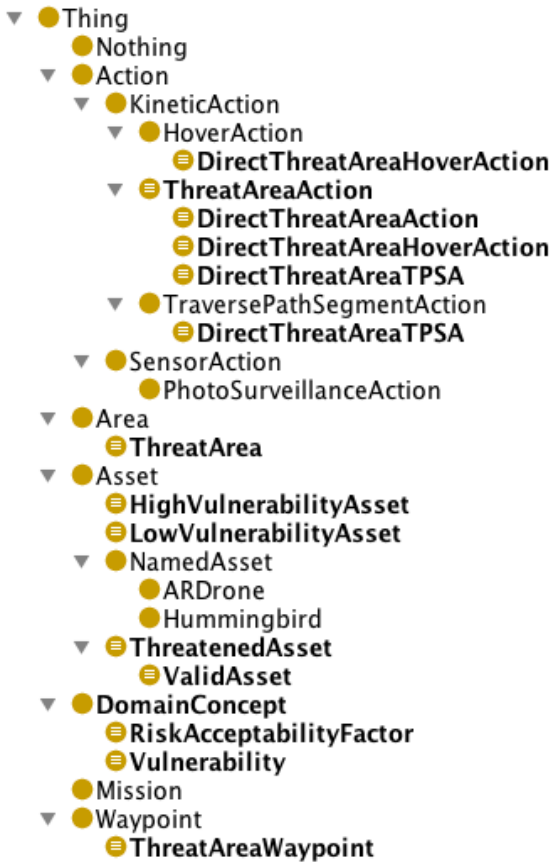

Figure 4.4: Tactical-CEMO's multiple inheritance class hierarchy as presented by the Protégé ontology editor.

with primitive classes, the conditions comprised by class `ThreatAreaWaypoint` must be satisfied by its members. In other words, class `ThreatAreaWaypoint` is defined using conditions that are necessary for, and sufficient to infer, class membership. Figure 4.6 illustrates the type of reasoning supported by necessary and sufficient conditions. The OWL code in Listing 4.6 formally defines class `ThreatAreaWaypoint`, with the keyword `EquivalentTo` establishing necessary and sufficient conditions for that class.

Listing 4.6: OWL code for class `ThreatAreaWaypoint`

```
1 Class: ThreatAreaWaypoint
2   EquivalentTo: Waypoint
3     and (isWaypointOf some ThreatArea)
```

Since members of class `Waypoint` are associated with `HoverAction` and `TraversePathSegmentAction` individuals (see Section 4.1.1), we specify that any kinetic action related to a threat area waypoint be inferred a member of class `ThreatAreaAction`. We assert the qualitative difference between *threat area actions* that initiate or prolong an incursion and threat area actions that terminate an incursion, and specify that the former be inferred members of class `DirectThreatAreaAction`. We further assert the qualitative difference between *direct threat area actions* (DTAAs) that execute exclusively,

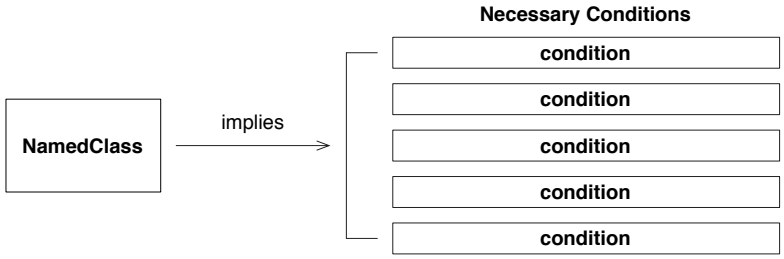

Figure 4.5: Necessary conditions, which describe primitive classes, cannot be used to infer class membership [77]. Contrast with Figure 4.6, which illustrates inference underpinned by necessary and sufficient conditions.

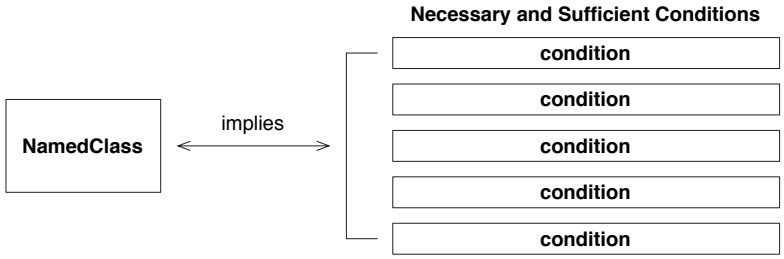

Figure 4.6: Necessary and sufficient conditions, which are comprised by defined classes, can be used to infer class membership [77]. Contrast with Figure 4.5, which illustrates inference underpinned by necessary conditions.

thereby endangering an asset seemingly without purpose, and those DTAAs that execute concurrently with one or more sensor actions. Given these assertions, we specify that assets with at least one assigned DTAA be inferred members of class `ThreatenedAsset`. We also specify that a threatened asset with assigned DTAAs be inferred a member of class `ValidAsset`, if at least one of those DTAAs executes concurrently with at least one sensor action assigned to the same asset. The OWL code in Listing 4.7 formally defines class `ValidAsset`.

Listing 4.7: OWL code for class `ValidAsset`

```
1 Class: ValidAsset
2   EquivalentTo: ThreatenedAsset
3     and (hasAction some
4       (SensorAction
5         and (hasSibling some DirectThreatAreaAction)))
```

The preceding code contains a nested class expression that is disambiguated with parentheses. This class expression can be understood as follows: A member of class `ValidAsset` is equivalent to a threatened asset associated, via the object property `hasAction`, to a member of class `SensorAction` that is in turn associated, via the object property `hasSibling`, to a member of class `DirectThreatAreaAction`. The OWL code in Listing 4.8 formally defines `hasSibling`.

Lines 2 and 3 in Listing 4.8 qualify the object property `hasSibling` with the asymmetric and irreflexive property characteristics, respectively (the transitive property char-

Listing 4.8: OWL code for the object property `hasSibling`

```
1 ObjectProperty: hasSibling
2   Characteristics: Asymmetric,
3     Irreflexive
4   Domain: SensorAction
5   Range: KineticAction
```

acteristic was introduced in Section 4.1.1). Because `hasSibling` is asymmetric[1], if an action a is related via `hasSibling` to an action b, then b cannot be related to a via the same property. Because `hasSibling` is irreflexive, if an action a is related via `hasSibling` to an action b, then a and b cannot be the same action. Figure 4.7 illustrates the asymmetric and irreflexive property characteristics that qualify `hasSibling`.

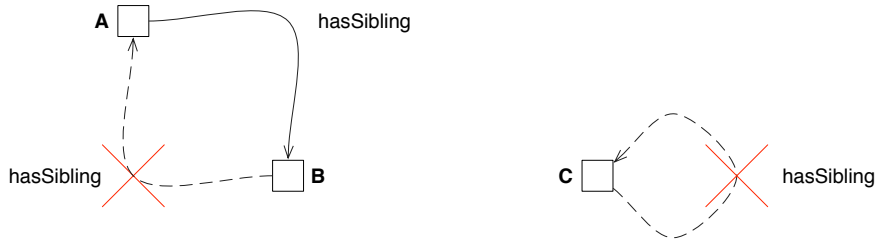

Figure 4.7: An example of the asymmetric (left) and irreflexive (right) property characteristics that qualify the object property `hasSibling`. The dashed lines represent inferred relationships.

Classes `ThreatenedAsset` and `ValidAsset` provide an initial mission verification mechanism in the context of our method. Specifically, a mission specification is inconsistent if it contains threatened assets that are not also *valid assets*. Once the validity of threatened assets has been inferred, the CVC synthesizes DTMC models that enable PRISM to compute the probability of survival for those assets. Accordingly, synthesized models encompass probabilities describing the *vulnerability* of real-world assets. The USDOD defines vulnerability as "the characteristic of a system that causes it to suffer a definite degradation ... [resulting from exposure] to a defined level of effects in a man-made hostile environment." [78] Vulnerability probabilities are derived from classes `HighVulnerabilityAsset` and `LowVulnerabilityAsset`, which extend class `Asset`. The OWL code in Listing 4.9 formally defines class `HighVulnerabilityAsset`.

Similar to class `Hummingbird`, the ranges of the datatype properties that parameterize members of class `HighVulnerabilityAsset`, including `hasCostValue` and `hasSpeedInKilometersPerHour` (lines 3 and 4, respectively, in Listing 4.9), restrict possible datatype values. The datatype property `hasEnduranceInSeconds`, which is inherited from class `Asset` (line 2), is not calibrated in a similar manner. The decision to calibrate two of the three inherited datatype properties implies that the calibrated properties are more likely to impact asset vulnerability. We note that, unlike class `Hummingbird`, the datatype properties in Listing 4.9 establish necessary and sufficient conditions (lines 1–4), and

[1]While potentially counterintuitive, an asymmetric sibling relationship is nevertheless consistent in the context of our domain model.

Listing 4.9: OWL code for class `HighVulnerabilityAsset`

```
1 Class: HighVulnerabilityAsset
2   EquivalentTo: Asset
3     and (hasCostValue some xsd:integer[<= 1000])
4     and (hasSpeedInKilometersPerHour some xsd:integer[<= 20])
5   SubClassOf:
6     hasRiskAcceptabilityFactor value HighRiskAcceptabilityFactor,
7     hasVulnerability value HighVulnerability
8   DisjointWith: LowVulnerabilityAsset
```

thereby support the inference of membership for class `HighVulnerabilityAsset`.

Lines 6 and 7 in Listing 4.9 use the keyword `value` to declare *hasValue* restrictions, which describe classes of individuals that must participate in at least one relationship, along a specified property, with a specific individual [77]. In particular, line 6 specifies that every member of class `HighVulnerabilityAsset` must be associated with the individual `HighRiskAcceptabilityFactor` via the object property `hasRiskAcceptability-Factor` (the concept of *risk acceptability* will be elaborated in Section 4.3.2). Line 7 specifies that every member of class `HighVulnerabilityAsset` must be associated with the individual `HighVulnerability` via the object property `hasVulnerability`. The individuals `HighRiskAcceptabilityFactor` and `HighVulnerability` belong, respectively, to the *enumerated classes* `RiskAcceptabilityFactor` and `Vulnerability`, which extend the generic class `DomainConcept`. An enumerated class is defined by listing precisely the individuals that are members of that class. The OWL code in Listing 4.10 formally defines class `Vulnerability`.

Listing 4.10: OWL code for class `Vulnerability`

```
1 Class: Vulnerability
2   EquivalentTo: DomainConcept
3     and ({HighVulnerability, LowVulnerability})
4   DisjointWith: RiskAcceptabilityFactor
```

Line 3 in Listing 4.10 specifies two individuals—`HighVulnerability` and `LowVul-nerability`—that constitute class `Vulnerability`. The OWL code in Listing 4.11 formally defines the individual `HighVulnerability`, which is associated with a datatype property describing a double-precision number (line 3). The value of this number is used by the CVC to calculate probabilities that are ultimately integrated into DTMC models representing high vulnerability assets. These modules enable PRISM to compute the probability of survival for high vulnerability assets during threat area incursions. The synthesis process will be elaborated in Section 4.3.

Listing 4.11: OWL code for the individual `HighVulnerability`

```
1 Individual: HighVulnerability
2   Types: Vulnerability
3   Facts: hasDoubleValue 0.1
```

The double-precision number encapsulated by the individual `HighVulnerability` is the raison d'être for that individual's existence in our ontology. In other words, because datatype properties link classes to data ranges (for example, class `Hummingbird`) and individuals to data values, we were compelled to create classes of *individuals* that could encapsulate risk acceptability and asset vulnerability *values*. And because the number of individuals per class was finite, enumeration enabled us to explicitly declare the completeness of those classes, and thereby create a finite set of risk acceptability and asset vulnerability grades.

The preceding discussion regarding class `HighVulnerabilityAsset` applies equally to class `LowVulnerabilityAsset`, which is formally defined in Listing 4.12. Datatype and object property ranges differentiate the two classes.

Listing 4.12: OWL code for class `LowVulnerabilityAsset`

```
Class: LowVulnerabilityAsset
  EquivalentTo: Asset
    and (hasCostValue some xsd:integer[>= 3000])
    and (hasSpeedInKilometersPerHour some xsd:integer[<= 60])
  SubClassOf:
    hasRiskAcceptabilityFactor value LowRiskAcceptabilityFactor
    hasVulnerability value LowVulnerability,
  DisjointWith: HighVulnerabilityAsset
```

4.1.3 Modeling Traffic Surveillance Missions

Having investigated tactical missions comprising threat area incursions, we considered a second case study involving traffic surveillance missions. In this scenario, which is conceptually similar to work presented by Heintz et al. [59], deployed UAVs are required to monitor freeway traffic and notify subscribers if traffic speeds exceed minimum and nominal thresholds. To accommodate traffic surveillance mission requirements, we have developed a *complex traffic surveillance missions ontology* (Traffic-CEMO) that extends CEMO.

Figure 4.8 illustrates Traffic-CEMO's class hierarchy. Classes describing traffic surveillance missions are highlighted in bold, and thereby differentiate from classes encoded in CEMO. Also highlighted in bold are classes defined in CEMO that have been declared disjoint from classes defined in Traffic-CEMO. Used mainly to support automated reasoning, traffic surveillance concepts are generally not available to mission developers via the YAML DSL. Concepts available to mission developers will be highlighted in the analysis that follows.

We define class `FreewaySection` in Traffic-CEMO and specify that two-lane freeway sections with high access ramp frequency be inferred, by Pellet or other semantic reasoners, members of `FreewaySection` subclass `LowSpeedFreewaySection`. We also specify that three-lane freeway sections with low access ramp frequency be inferred members of `FreewaySection` subclass `HighSpeedFreewaySection`. These inferred relationships

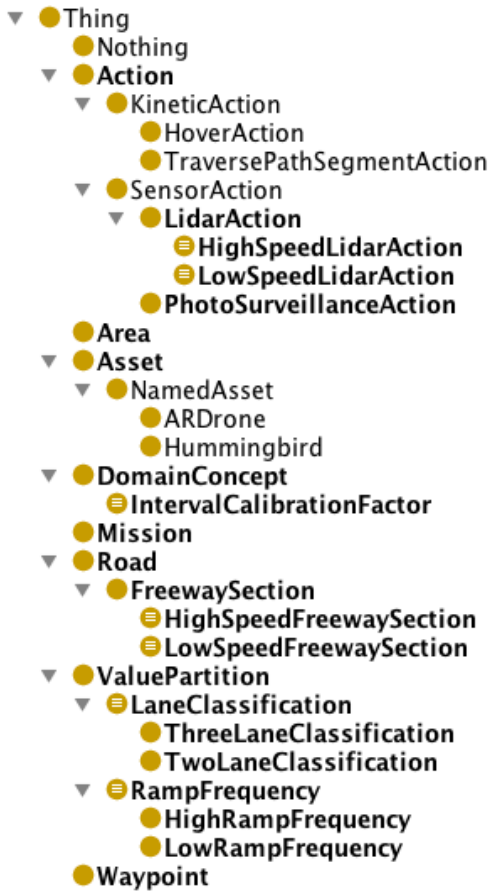

Figure 4.8: Traffic-CEMO's class hierarchy as presented by the Protégé ontology editor.

formalize the correlation between off-ramp over-saturation and freeway bottlenecks that disrupt traffic discharge rates [79]. The OWL code in Listing 4.13 formally defines class `FreewaySection`.

Listing 4.13: OWL code for class `FreewaySection`

```
1 Class: FreewaySection
2   SubClassOf: Road,
3     approachesMinimumSpeed some xsd:double,
4     exceedsMinimumSpeed some xsd:double,
5     exceedsNominalSpeed some xsd:double,
6     hasLaneClassification some LaneClassification,
7     hasRampFrequency some RampFrequency
```

Line 3 in Listing 4.13 associates members of class `FreewaySection` with the datatype property `approachesMinimumSpeed`, which describes the potential for minimum traffic speeds to be approached during the operation of freeway sections. The datatype properties `exceedsMinimumSpeed` (line 4) and `exceedsNominalSpeed` (line 5) likewise describe the potential for minimum and nominal traffic speeds, respectively, to be exceeded during the operation of freeway sections. Lines 6 and 7 specify that every member of class `FreewaySection` must be associated, via the object properties `hasLaneClassification` and `hasRampFrequency`, with members of classes `LaneClassification` and `RampFre-`

quency, respectively. These classes extend class ValuePartition, which represents the *value partition* design pattern [77].

We use value partitions to describe the concepts of *lane classification* and on/off *ramp frequency*, and to restrict the range of possible values for those concepts to an exhaustive list of mutually exclusive choices. For example, class LaneClassification restricts the range of the object property hasLaneClassification to classes TwoLaneClassification and ThreeLaneClassification. The OWL code in Listing 4.14 formally defines class LaneClassification.

Listing 4.14: OWL code for class LaneClassification

```
1 Class: LaneClassification
2   EquivalentTo: TwoLaneClassification
3     or ThreeLaneClassification
4   SubClassOf: ValuePartition
5   DisjointWith: RampFrequency
```

Lines 1–3 in Listing 4.14 specify a *covering axiom*. This type of axiom comprises a set of classes that *cover* a common superclass by virtue of their union; for example, LaneClassification subtypes TwoLaneClassification and ThreeLaneClassification form a union (lines 2 and 3) that covers their common superclass. We note that the union of classes TwoLaneClassification and ThreeLaneClassification establishes necessary and sufficient conditions for class LaneClassification (necessary and sufficient conditions were described in Section 4.1.2). The OWL code in Listing 4.15 formally defines class TwoLaneClassification.

Listing 4.15: OWL code for class TwoLaneClassification

```
1 Class: TwoLaneClassification
2   SubClassOf: LaneClassification
3   DisjointWith: ThreeLaneClassification
```

Line 3 in Listing 4.15 specifies that classes TwoLaneClassification and ThreeLaneClassification constitute disjoint sets of individuals. Because class LaneClassification is *covered* by the union of its two subclasses, and because those subclasses are disjoint, a member of class LaneClassification must also be a member of either TwoLaneClassification or ThreeLaneClassification. As with enumeration, which was described in Section 4.1.2, value partitions enabled us to create a finite set of possible values (in this case, lane classification and ramp frequency grades). The value partitions that describe lane classification and ramp frequency are used to parameterize FreewaySection subtypes including classes LowSpeedFreewaySection and HighSpeedFreewaySection. The OWL code in Listing 4.16 formally defines the latter class.

Lines 1–4 in Listing 4.16 use value partitions to establish necessary and sufficient conditions, and thereby support the inference of membership for class HighSpeedFreewaySection. Unlike class FreewaySection, the ranges of the datatype properties that parameterize members of class HighSpeedFreewaySection (lines 5–7) restrict possible data

Listing 4.16: OWL code for class `HighSpeedFreewaySection`

```
1 Class: HighSpeedFreewaySection
2   EquivalentTo: FreewaySection
3     and (hasLaneClassification some ThreeLaneClassification)
4     and (hasRampFrequency some LowRampFrequency)
5   SubClassOf: approachesMinimumSpeed some xsd:double[>= 0.2],
6     exceedsMinimumSpeed some xsd:double[>= 0.9],
7     exceedsNominalSpeed some xsd:double[>= 0.5]
8   DisjointWith: LowSpeedFreewaySection
```

values. These data values specify probabilities for traffic speed fluctuations during the operation of high-speed freeway sections. Traffic speed fluctuations affect the optimal surveillance of freeway traffic by airborne *light detection and ranging* (LIDAR) systems.

LIDAR is an optical remote sensing technology that can observe targets from a distance of 15 kilometers (in air) with sub-millimeter resolution [80]. The integration of LIDAR systems into rotary wing UAV platforms could enable, for example, a single hovering UAV with a five kilometer altitude ceiling to monitor traffic over an area spanning approximately 630 square kilometers. In the context of an urban environment, LIDAR-equipped UAVs offer a flexible and optimizable alternative to land-based cameras for the simultaneous surveillance of multiple freeway sections. The OWL code in Listing 4.17 models the concept of LIDAR systems by formally defining class `LidarAction` as a sensor action subtype.

Listing 4.17: OWL code for class `LidarAction`

```
1 Class: LidarAction
2   SubClassOf: SensorAction,
3     hasIntervalInSeconds some xsd:integer,
4     hasIntervalCalibrationFactor some IntervalCalibrationFactor,
5     isConcurrentWith some HoverAction
6   DisjointWith: PhotoSurveillanceAction
```

The object property `hasIntervalCalibrationFactor` (line 4 in Listing 4.17) links LIDAR actions with the individuals `HighIntervalCalibrationFactor` and `LowIntervalCalibrationFactor`, which constitute the enumerated class `IntervalCalibrationFactor`. Members of class `IntervalCalibrationFactor` are associated with a datatype property describing an integer. The value of this integer, which is ultimately integrated into DTMC models that simulate LIDAR actions, calibrates intervals between LIDAR readings. LIDAR intervals are specified by mission developers with a construct from our DSL; the interval type that verifies mission plans is specified by the datatype property `hasIntervalInSeconds` (line 5). Every member of class `LidarAction` is also associated, via the object property `isConcurrentWith` (line 3), with a `HoverAction` individual. This relationship is specified by mission developers via the YAML DSL.

Class `LidarAction` is extended by class `HighSpeedLidarAction`, which is formally defined by the OWL code in Listing 4.18. `LidarAction` subtype individuals are associ-

ated with members of class `HighSpeedFreewaySection` via the object property `monitors` (line 3). Given the necessary and sufficient conditions established in Listing 4.18 (lines 1–3), LIDAR actions that monitor *high-speed freeway sections* are inferred members of class `HighSpeedLidarAction`. Thus `FreewaySection` subtypes support inferences that determine the classification of `LidarAction` individuals. The object property `monitors`, which encompasses the concurrency relationship specified by mission developers, will be elaborated in Section 4.2.

Listing 4.18: OWL code for class `HighSpeedLidarAction`

```
1 Class: HighSpeedLidarAction
2   EquivalentTo: LidarAction
3     and (monitors some HighSpeedFreewaySection)
4   SubClassOf: hasIntervalCalibrationFactor value
      HighIntervalCalibrationFactor
5   DisjointWith: LowSpeedLidarAction
```

Line 4 in Listing 4.18 links `HighSpeedLidarAction` individuals with a *high interval calibration factor*, which is integrated into DTMC models representing *high-speed LIDAR actions*. Thus `LidarAction` classifications support the synthesis of DTMC artifacts that model the behavior of LIDAR action individuals. The discussion regarding class `HighSpeedLidarAction` applies equally to class `LowSpeedLidarAction`, which is formally defined in Listing 4.19. Object property ranges differentiate the two classes.

Listing 4.19: OWL code for class `LowSpeedLidarAction`

```
1 Class: LowSpeedLidarAction
2   EquivalentTo: LidarAction
3     and (monitors some LowSpeedFreewaySection)
4   SubClassOf: hasIntervalCalibrationFactor value
      LowIntervalCalibrationFactor
5   DisjointWith: HighSpeedLidarAction
```

Appendix B presents CEMO, Tactical-CEMO and Traffic-CEMO in their entirety.

4.1.4 An Overview

Section 4.1 exploits OWL to formalize a subset of the UAV domain. With respect to the OWL constructs presented in Section 2.1: Listing 4.6 uses the equivalence predicate `EquivalentTo` to establish necessary and sufficient conditions for class `ThreatArea-Waypoint`. Listing 4.2 uses the primitive `DisjointWith` to specify that classes `ARDrone` and `Hummingbird` constitute disjoint sets. And Listing 4.10 uses enumeration to describe class `Vulnerability`. These and other OWL-based modeling methods described throughout Section 4.1 support the expressive representation of knowledge by domain experts.

OWL also supports the efficient reasoning afforded by Pellet. Two strands of reasoning are particularly compelling because they span multiple classes and relationships. Tactical-CEMO comprises the inferred concepts `DirectThreatAreaAction` and `ValidAsset`; these concepts support the synthesis or PRISM code from templates, which will be presented in Section 4.3.1 and Section 4.3.2. Traffic-CEMO also comprises inferred concepts including `HighSpeedFreewaySection` and `HighSpeedLidarAction`, which support the synthesis of PRISM code from templates presented in Section 4.3.3. The OWL-based inferences described throughout Section 4.1 enable the CVC to generate low-level system models and their desired behavioral properties from high-level system specifications encoded by model builders.

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 4.20 presents an incomplete definition of class `CrossCuttingKineticAction`.

Listing 4.20: OWL code for class `CrossCuttingKineticAction`

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

The definition in Listing 4.20 is lacking because the asset in line 3 cannot be differentiated from the asset in line 6. The SWRL code in Listing 4.21 uses the built-in OWL property `differentFrom` to appropriately define class `CrossCuttingKineticAction` (question mark prefixes denote variables). The OWL code in line 1 provides a rudimentary definition for the `CrossCuttingKineticAction` concept, which is augmented by the rule in lines 3–12.

The SWRL rule in Listing 4.21 augments the definition of an OWL class. Listing 4.22 uses a SWRL rule to augment the OWL object property `monitors`, which supports inferences described in Section 4.1.3. Lines 1–5 formally define `monitors` as an asymmetric and irreflexive object property with domain and range the classes `LidarAction` and `FreewaySection`, respectively (similar definitions have been introduced throughout this chapter). Line 14 specifies that a LIDAR action *?l* (defined in line 10) monitors a freeway section *?f* (defined in line 8) if the hover action *?h* (defined in line 9) that executes concurrently with *?l* (as specified in line 13) is associated with a waypoint *?w* (as specified in line 11) that also delineates *?f* (as specified in line 12).

Listing 4.21: OWL+SWRL code for rule `CrossCuttingKineticAction`

```
Class: CrossCuttingKineticAction

Rule:
    KineticAction(?a),
    KineticAction(?b),
    Asset(?x),
    Asset(?y),
    hasAction(?x, ?a),
    hasAction(?y, ?b),
    hasPrecondition(?b, ?a),
    DifferentFrom(?x, ?y)
  -> CrossCuttingKineticAction(?a)
```

Listing 4.22: OWL+SWRL code for rule `monitors`

```
ObjectProperty: monitors
  Characteristics: Asymmetric,
    Irreflexive
  Domain: LidarAction
  Range: FreewaySection

Rule:
    FreewaySection(?f),
    HoverAction(?h),
    LidarAction(?l),
    hasWaypoint(?f, ?w),
    hasWaypoint(?h, ?w),
    isConcurrentWith(?l, ?h)
  -> monitors(?l, ?f)
```

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 4.23 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.

Listing 4.23: Prolog code for rule `terminal`

```
terminal(X) :-
    has_action(A, X),
    not(is_precondition_to(X, _)),
    not(single_action_asset(A)),
    not(zero_action_asset(A)).
```

The transition of inferred knowledge from OWL to Prolog can be illustrated with the atom `is_precondition_to` (line 3 in Listing 4.23). The OWL code in Listing 4.24 formally defines the object property `isPreconditionTo`, a relationship that inverts the object property `hasPrecondition`. This inversion constitutes a Pellet inference (as described in Section 4.1).

Listing 4.24: OWL code for the object property `isPreconditionTo`

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

Mission A comprises several instances of the DSL construct `preconditions` (lines 10, 18 and 22 in Listing 3.1). 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.

Appendix B and Appendix C present, respectively, the SWRL and Prolog rule-bases in their entirety.

4.2.1 An Overview

Section 4.2 exploits SWRL to overcome OWL limitations and encode complex relational structures. SWRL rules in Listing 4.21 and Listing 4.22 augment class `CrossCuttingKineticAction` and the object property `monitors`, respectively. The inferred concept `CrossCuttingKineticAction` supports PRISM code synthesis via a process that will be described in Chapter 5. In conjunction with several inferred concepts, including `HighSpeedFreewaySection` and `HighSpeedLidarAction`, the object property `monitors` supports the synthesis of PRISM code from templates, which will be presented in Section 4.3.3.

Prolog is used to overcome OWL+SWRL limitations and encode knowledge that can support effective reasoning with negation. Listing 4.23 and Listing 4.24 illustrate composite FOL reasoning, which is reasoning supported by multiple and integrated FOL-based systems, with the transition of inferred knowledge from OWL to Prolog. The process that leverages composite reasoning to synthesize PRISM code will be described in Chapter 5.

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 4.25 uses string interpolation, which is denoted by the 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.

Listing 4.25: DTMC template for asset modules

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

A module definition contains *variables* and *commands*. Asset module states are stored in variable `e#{asset.id}` (lines 2, 4, 5 and 7 in Listing 4.25), which represents asset endurance. Line 2 declares `e#{asset.id}` to be an integer variable with range `[0..#{asset.endurance}]`. The upper limit for the range is derived from `asset.endurance`, which also initializes `e#{asset.id}` (as denoted by the keyword `init`).

Asset behavior is formalized with two or more commands, where each command assumes the form:

$$[action]\ guard \rightarrow P(update_1) : update_1 + \ldots + P(update_n) : update_n;$$

A command becomes *enabled* for execution when its *guard* is satisfied by a specific model state; for example, lines 4 and 5 in Listing 4.25 specify a command that becomes enabled when action duration, which is denoted by variable `d#{action.id}`, *and* asset endurance exceed zero. 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 3 and 6 in Listing 4.25 use meta-code, which is highlighted with yellow background, to define a for loop. The form assumed by each for loop is

```
FOR var IN collection DO  body  END
```

where `var` denotes a variable and `collection` denotes a collection of objects. Each asset module command generated by the for loop in Listing 4.25 represents the execution of a kinetic action. The `true` keyword in the last command (line 8) 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.

Arguments for the parameters in Listing 4.25 are derived from explicit and inferred domain knowledge. Specifically, arguments for `asset.class` (line 1) and `asset.id` (lines 1, 2, 4, 5 and 7) are derived from mission plans; arguments for `asset.endurance` (line 2) are derived from explicit domain knowledge encoded in CEMO; and arguments for `action.type` (line 4), `asset.last_action.type` (line 7) and `asset.last_action.id` (line 7) are derived from knowledge inferred by Pellet and the Prolog compiler (via a process that will be described in Chapter 5). Listing 4.25 uses blue text to highlight parameters that receive as arguments domain-specific knowledge encoded by domain experts and derived from inferences. Purple text highlights parameters that receive as arguments mission-specific knowledge encoded by mission developers and also derived from inferences. This color coding scheme will be used for the remainder of Chapter 4.

Mission properties are encoded in PRISM's property specification language, which subsumes several probabilistic temporal logics including PCTL, CSL and LTL. The code in Listing 4.26 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. Arguments for the parameter `action.id` are derived from knowledge inferred by Pellet and the Prolog compiler (via a process that will be described in Chapter 5).

Listing 4.26: PCTL template for action duration

```
1 P=? [ F d#{action.id}=0 ... ]
```

4.3.1 Modeling Survivability

Asset module commands comprise single updates that omit probabilities. The code in Listing 4.27 formally defines a template for asset survivability modules, which contain commands with multiple updates. Survivability modules enable PRISM to calculate the probability of survival for `ValidAsset` individuals with respect to the threat area actions executed by each asset (valid assets were described in Section 4.1.2). Module states are stored in the boolean variable `a#{asset.id}d` (defined in line 9 and used in lines 11–14), which represents the destruction of the asset with identifier `asset.id`. Command updates transition survivability modules from one state to the next. The probability of execution for each update is derived from the parameter `asset.vulnerability` (lines 12 and 13). Current vulnerability values are arbitrary, and should eventually be calculated from real-world data related to asset capabilities, terrain types and weather conditions.

Lines 1 and 6 in Listing 4.27 use meta-code to define a for loop that generates one *formula* with identifier `actn#{action.id}_tai` (defined in line 4 and used in lines 11 and 14) per each action assigned to the asset in the loop header (the formula is a PRISM

Listing 4.27: DTMC template for asset survivability

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

language construct used to avoid code duplication). 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` (threat area actions were described in Section 4.1.2). This overlap is delineated by variables `start#{action.id}` and `finish#{action.id}` (defined in lines 2 and 3, respectively), which are assigned geographic information calculated by the CVC during preprocessing (as described in Chapter 3).

The constructs `a#{asset.id}d` and `actn#{action.id}_tai` combine to form logical expressions, including a logical conjunction and disjunction (lines 11 and 14, respectively), that constitute the guards in the asset survivability module. When these expressions are considered in union, their logical truth values, which are highlighted in Table 4.1, clarify asset behavior and susceptibility during threat area incursions. Specifically, a valid asset exists in one of the following disjoint states:

- *not* destroyed and prosecuting a threat area incursion, whereby the logical conjunction is true;
- destroyed, as a consequence of prosecuting a threat area incursion, or *not* prosecuting a threat area incursion, whereby the logical disjunction is true.

d	tai	$!d \wedge tai$	$d \vee !tai$
true	true	false	true
true	false	false	true
false	true	true	false
false	false	false	true

Table 4.1: A truth table for asset survivability guards, with variables `a#{asset.id}d` and `actn#{action.id}_tai` in Listing 4.27 represented by variables d and tai, respectively.

Arguments for the parameters in Listing 4.27 are derived from explicit and inferred domain knowledge. Specifically, arguments for `action.id` (lines 4, 5, 11 and 14), `asset.class` (line 8) and `asset.id` (lines 8, 9 and 11–14) are derived from mission plans; arguments for `asset.vulnerability` (lines 12 and 13) are derived from knowledge inferred by Pellet; arguments for `action.start` (line 2), `action.finish` (line 3) are derived from geodetic calculations and knowledge inferred by Pellet; and arguments for `action.type` (lines 11 and 14) are derived from knowledge inferred by Pellet and the Prolog compiler.

The generic property in Listing 4.28 queries the probability that an asset with identifier `asset.id` will not be destroyed during its prosecution of threat area incursions. This mission property makes explicit the intuitive correlation between asset survivability and mission correctness.

Listing 4.28: PCTL template for asset survivability

```
1 P=? [ F !a#{asset.id}d ... ]
```

4.3.2 Modeling Risk Acceptability

We assert that, during threat area incursions, mission correctness is contingent on asset survivability *and* risk acceptability. As described in Section 4.1.2, a threat area action that initiates or prolongs an incursion is inferred a member of class `DirectThreatAreaAction`. 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.

The code in Listing 4.29 formally defines a template that enables PRISM to calculate risk acceptability with respect to the threat area incursions prosecuted by a `ValidAsset` individual. For a given asset, RAF values quantifying the incursions prosecuted by that asset are aggregated by the formula with identifier `raf#{asset.id}` (line 39). Numerator and denominator for this formula are derived, respectively, from the constructs `sad#{asset.id}` and `tkad#{asset.id}`.

Listing 4.29 uses a sensor action counter module (lines 16–37) to calculate the duration of concurrent execution between sensor actions and DTAAs during threat area incursions prosecuted by an asset with identifier `asset.id`. Module states are stored in variable `sad#{asset.id}` (defined in line 17 and used in lines 26, 27 and 39), which is incremented by command updates. Lines 18 and 36 use meta-code to define an outer for loop that generates two commands per each DTAA assigned to the asset in the loop header. Guards for these commands encapsulate the constructs `actn#{action.id}_tai` and `r#{sensor_action.id}`.

Listing 4.29: DTMC template for risk acceptability

```
1  FOR action IN asset.direct_threat_area_actions DO
2  const int start#{action.id} = #{action.start};
3  const int finish#{action.id} = #{action.finish};
4  formula actn#{action.id}_tai = d#{action.id}>finish#{action.id} &
5      d#{action.id}<=start#{action.id};
6  formula duration#{action.id} = start#{action.id} - finish#{action.id};
7  END
8
9  formula tkad#{asset.id} =
10     FOR action IN asset.direct_threat_area_actions DO
11     duration#{action.id}
12     +  IF action != asset.direct_threat_area_actions.last
13     ;  IF action == asset.direct_threat_area_actions.last
14     END
15
16 module SensorActionCounter#{asset.id}
17   sad#{asset.id} : [0..tkad#{asset.id}] init 0;
18   FOR action IN asset.direct_threat_area_actions DO
19   [#{action.type}] actn#{action.id}_tai &
20       (
21            FOR sensor_action IN asset.sensor_actions DO
22            r#{sensor_action.id}
23            |  IF sensor_action != asset.sensor_actions.last
24            END
25       ) &
26       sad#{asset.id}<tkad#{asset.id}
27       -> (sad#{asset.id}'=sad#{asset.id}+1);
28   [#{action.type}] !actn#{action.id}_tai |
29       !(
30            FOR sensor_action IN asset.sensor_actions DO
31            r#{sensor_action.id}
32            |  IF sensor_action != asset.sensor_actions.last
33            END
34       )
35       -> true;
36   END
37 endmodule
38
39 formula raf#{asset.id} = sad#{asset.id} / tkad#{asset.id};
```

The formula identifier `actn#{action.id}_tai` in Listing 4.29 (defined in line 4 and used in lines 19 and 28) represents the execution of a kinetic action during the prosecution of a threat area incursion. In the context of a sensor action counter module, the scope of formula `actn#{action.id}_tai` is limited by the outer for loop. Specifically, the outer loop generates only those identifiers associated with the execution of DTAAs (as dictated by the loop header), which constitute a subset of threat area actions assigned to the asset in the loop header.

Variable `r#{sensor_action.id}` (lines 22 and 31 in Listing 4.29) couples the behavior of each sensor action counter module to the state of a sensor action module representing the execution of a sensor action with identifier `action.id`. Each counter module command is associated with a for loop (lines 21–24 and 30–33) that generates

one such variable per each sensor action assigned to the asset in the loop header. Two or more variables form a logical disjunction with operands generated by an if statement (lines 23 and 32). The form assumed by each if statement is

```
code IF expression
```

where the `code` is executed if (and only if) the `expression` evaluates to something other than `false` or `nil`.

The constructs `actn#{action.id}_tai` and `r#{sensor_action.id}` combine to form logical expressions, including a logical conjunction and disjunction (lines 19–25 and 28–34, respectively, in Listing 4.29), that constitute the guards in the sensor action counter module. When these expressions are considered in union, their logical truth values, which are highlighted in Table 4.2, clarify the conditions that increment variable `sad#{asset.id}`. Specifically, and in accordance with its specification as described above, the RAF numerator is incremented during the prosecution of a threat area incursion, and the concurrent execution of a direct threat area action and at least one sensor action, whereby the logical conjunction is true. The execution of the update in the second command prevents inconsequential deadlocks (as described in Section 4.3), whereby the logical disjunction is true. We note that the expression in line 26 prevents variable `sad#{asset.id}` from exceeding its upper limit. This expression, which is required by the PRISM language, does not alter the essence of the logical truth values presented in Table 4.2.

tai	r	$tai \wedge r$	$!tai \vee !r$
true	true	true	false
true	false	false	true
false	true	false	true
false	false	false	true

Table 4.2: A truth table for sensor action counter guards, with variables `actn#{action.id}_tai` and `r#{sensor_action.id}` in Listing 4.29 represented by variables tai and r, respectively. Variable r may also subsume a logical disjunction comprising two or more instances of variable `r#{sensor_action.id}`.

Lines 1 and 7 in Listing 4.29 use meta-code to define a for loop that generates one formula with identifier `duration#{action.id}` (line 6) per each action assigned to the asset in the loop header. Each formula uses variables `start#{action.id}` and `finish#{action.id}` (lines 2 and 3, respectively) to calculate the duration of overlap between the execution of a DTAA with identifier `action.id` and the prosecution of a threat area incursion by the asset to which that DTAA is assigned. Given its function as the RAF denominator, the formula identifier `tkad#{asset.id}` represents the aggregate duration of DTAA executions during incursions prosecuted by the asset with identifier `asset.id`. Formula `tkad#{asset.id}` (lines 9–14) comprises a for loop that increments

the RAF denominator by generating the formula identifier `duration#{action.id}` as an addition operand once per each action executed by the asset in the loop header.

RAF values resulting from the execution of synthesized PRISM code are verified against a RAF threshold value, which is specified in Tactical-CEMO and integrated by the CVC into synthesized mission properties. The generic property in Listing 4.30 queries the probability that the specified RAF value exceeds 60 percent. This threshold is encapsulated by the individual `LowRiskAcceptabilityFactor`, which belongs to the enumerated class `RiskAcceptabilityFactor` described in Section 4.1.2. Current threshold values are arbitrary, and should eventually be calculated from real-world data related to operational assessments.

Listing 4.30: PCTL template for risk acceptability

```
1 P=? [ F raf#{asset.id}>0.6 ... ]
```

Arguments for the parameters in Listing 4.29 are derived from explicit and inferred domain knowledge. Specifically, arguments for `action.id` (lines 2–6, 11, 19 and 28), `asset.id` (lines 9, 16, 17, 26, 27 and 39) and `sensor_action.id` (lines 22 and 31) are derived from mission plans; arguments for `action.start` (line 2), `action.finish` (line 3) are derived from geodetic calculations and knowledge inferred by Pellet; and arguments for `action.type` (lines 19 and 28) are derived from knowledge inferred by Pellet and the Prolog compiler.

4.3.3 Modeling Traffic Surveillance

Section 4.1.3 describes a case study whereby LIDAR-equipped UAVs monitor freeway traffic and notify subscribers if traffic speed exceeds specified thresholds. Traffic speed fluctuations in turn affect the optimal surveillance of freeway traffic by monitoring assets. Specifically, when traffic speed descends below the nominal speed threshold, and thereby approaches the minimum speed threshold, of a freeway section *fs*, the sample rate of the LIDAR action *la* monitoring *fs* is increased by a factor equal to the calibration value associated with the conjunction of *la* and *fs*. Increased LIDAR sample rates afford fine-grained traffic control (via some process that is outside the scope of our method), which is assumed to be an appropriate mechanism for mitigating traffic congestion and bottlenecks.

Conversely, when traffic speed ascends above the nominal speed threshold, and is thereby distanced from the minimum speed threshold, of freeway section *fs*, the sample rate of *la* is decreased by a factor equal to the calibration value associated with the conjunction of *la* and *fs*. Decreased LIDAR sample rates afford coarse-grained traffic control, which is assumed to be acceptable during periods of decreased freeway traffic. LIDAR calibration can thus optimize the surveillance of multiple freeway sections by a single UAV.

In this context, we consider the following mission scenario:

1. Traffic speed exceeds the nominal speed threshold of freeway section *fs*.
2. The system optimizes its surveillance operation by decreasing the sample rate of *la* with respect to *fs*.
3. Subsequently, and during what is now a prolonged *la* interval, traffic speed descends rapidly below the minimum speed threshold of *fs*.

Steps 2 and 3 in this scenario highlight potentially conflicting mission requirements. The system must achieve optimized performance with limited resources. Concurrently, to manage freeway traffic in real-time, the system must disseminate timely (and potentially low-latency) information from ISR assets to human and software traffic control agents. The latter requirement could presumably be compromised if *la* remains inactive for a considerable period of time following the occurrence of step 3.

Listing 4.31 formally defines a template that enables PRISM to quantify probabilistically the correlation between decreased LIDAR sample rates and increased latency in the dissemination of information related to traffic speed fluctuations. Lines 27 and 41 use meta-code to define a for loop that generates one freeway section module with identifier `FreewaySection#{section.id}` (line 28) per each freeway section assigned to the asset in the loop header (freeway sections are assigned to specific assets via inferences supported by the object property `monitors`, which was described in Section 4.1.3 and Section 4.2). Freeway section modules enable PRISM to simulate traffic speed fluctuations for high- and low-speed freeways. Module states representing disjoint traffic speed intervals are stored in variable `spd#{section.id}` (defined in line 29 and used in lines 14, 19 and 30–39). Three intervals describe minimum (denoted by the integer 0), nominal (1) and greater than nominal (2) traffic speeds.

Command updates transition freeway section modules from one state to the next as follows: The first command derives the probability of transitioning from minimum to nominal speed via the parameter `section.exceeds_minimum_speed` (line 32 in Listing 4.31). The negation of this parameter determines the probability of maintaining minimum speed (line 31).

The second command derives the probability of transitioning from nominal to minimum speed via the parameter `section.approaches_minimum_speed` (line 34 in Listing 4.31); the probability of transitioning from nominal to greater than nominal speed is derived via the parameter `section.exceeds_nominal_speed` (line 36). The sum of these parameters is negated to determine the probability of maintaining nominal speed (line 35).

The third command derives the probability of maintaining greater than nominal speed via the parameter `section.exceeds_nominal_speed` (line 39 in Listing 4.31). The negation of this parameter determines the probability of transitioning from greater than nominal to nominal speed (line 38). Arguments for `section.approaches_minimum_speed`, `section.exceeds_minimum_speed` and `section.exceeds_nominal_speed` are derived via inferences from explicit domain knowledge encoded in CEMO (as described in Sec-

Listing 4.31: DTMC template for traffic surveillance

```
1  FOR action IN asset.lidar_actions DO
2  const int i#{action.id} = #{action.interval};
3  const int icf#{action.id} = #{action.interval_calibration_factor};
4  formula ci#{action.id} = nse#{action.id}
5      ? i#{action.id} * icf#{action.id}
6      : i#{action.id};
7
8  module LidarAction#{action.id}
9    r#{action.id} : [0..#{action.concurrent_action.duration}] init 0;
10   nse#{action.id} : bool init false;
11   [#{action.concurrent_action.type}]
12       mod(d#{action.concurrent_action.id}, ci#{action.id})=0 &
13       r#{action.id}<#{action.concurrent_action.duration} &
14       spd#{action.section.id}=2
15       -> (r#{action.id}'=r#{action.id}+1) & (nse#{action.id}'=true);
16   [#{action.concurrent_action.type}]
17       mod(d#{action.concurrent_action.id}, ci#{action.id})=0 &
18       r#{action.id}<#{action.concurrent_action.duration} &
19      !spd#{action.section.id}=2
20       -> (r#{action.id}'=r#{action.id}+1) & (nse#{action.id}'=false);
21   [#{action.concurrent_action.type}]
22      !mod(d#{action.concurrent_action.id}, ci#{action.id})=0
23       -> true;
24 endmodule
25 END
26
27 FOR section IN asset.freeway_sections DO
28 module FreewaySection#{section.id}
29   spd#{section.id} : [0..2] init 1;
30   [#{section.action.concurrent_action.type}] spd#{section.id}=0
31       -> #{1-section.exceeds_minimum_speed}:(spd#{section.id}'=0) +
32          #{section.exceeds_minimum_speed}:(spd#{section.id}'=1);
33   [#{section.action.concurrent_action.type}] spd#{section.id}=1
34       -> #{section.approaches_minimum_speed}:(spd#{section.id}'=0) +
35          #{1-(section.exceeds_nominal_speed + section.approaches_minimum_speed)}:(
               spd#{section.id}'=1) +
36          #{section.exceeds_nominal_speed}:(spd#{section.id}'=2);
37   [#{section.action.concurrent_action.type}] spd#{section.id}=2
38       -> #{1-section.exceeds_nominal_speed}:(spd#{section.id}'=1) +
39          #{section.exceeds_nominal_speed}:(spd#{section.id}'=2);
40 endmodule
41 END
```

tion 4.1.3). Current probabilities are arbitrary, and should eventually be calculated from real-world data related to freeway surveillance operations.

Lines 1 and 25 in Listing 4.31 use meta-code to define a for loop that generates one LIDAR action module with identifier `LidarAction#{action.id}` (line 8) per each LIDAR action assigned to the asset in the loop header. Module states are stored in variables `r#{action.id}` (defined in line 9 and used in lines 13, 15, 18 and 20) and `nse#{action.id}` (defined in line 10 and used in lines 4, 15 and 20). The former variable represents LIDAR readings; for any given LIDAR action, the latter variable represents a transition to greater than nominal traffic speed for the freeway section monitored by

that action.

Line 9 in Listing 4.31 declares `r#{action.id}` to be an integer variable with range `[0..#{action.concurrent_action.duration}]`. This variable is incremented by updates in two commands (lines 11–20), which represent the execution of LIDAR actions; a third command uses the `true` keyword (line 23) to represent execution intervals. Each command is enabled, either primarily or exclusively, by a modulo operation (lines 12, 17 and 22) between the constructs `d#{action.concurrent_action.id}` and `ci#{action.id}`.

Variable `d#{action.concurrent_action.id}` (lines 12, 17 and 22 in Listing 4.31) represents the duration of the hover action that executes concurrently with the LIDAR action represented by variable `action`; formula `ci#{action.id}` (lines 4–6) calibrates the sample rate of this LIDAR action with variables `i#{action.id}` (defined in line 2 and used in lines 5 and 6) and `icf#{action.id}` (defined in line 3 and used in line 5). Specifically, the LIDAR action interval `i#{action.id}` is calibrated by the interval calibration factor `icf#{action.id}` (as described in Section 4.1.3) when `nse#{action.id}==true`, i.e., when traffic speed exceeds the nominal speed threshold of the freeway section monitored by the LIDAR action with identifier `action.id`. Variable `nse#{action.id}` is updated, by the same commands that update `r#{action.id}`, when variable `spd#{section.id}` transitions between values representing minimum/nominal and greater than nominal traffic speeds. The commands that update variables `r#{action.id}` and `nse#{action.id}` comprise expressions to prevent `r#{action.id}` from exceeding its upper limit (lines 13 and 18). These expressions, which are required by the PRISM language, do not alter the essence of the functionality afforded by LIDAR action modules.

Listing 4.31 labels all commands with actions that synchronize the execution of PRISM modules. Synchronization has been used throughout Section 4.3 to accurately model tightly-coupled units of behavior including LIDAR actions and the freeway sections monitored by those actions. The relationship between modeling accuracy and synchronization will be elaborated in Chapter 5.

Arguments for the parameters in Listing 4.31 are derived from explicit and inferred domain knowledge. Specifically, arguments for `action.id` (lines 2–6, 8–10, 12, 13, 15, 17, 18, 20 and 22), `action.interval` (line 2), `action.concurrent_action.duration` (lines 9, 13 and 18) and `action.concurrent_action.id` (lines 12, 17 and 22) are derived from mission plans; arguments for `action.interval_calibration_factor` (line 3), `action.section.id` (lines 14 and 19), `section.exceeds_minimum_speed` (lines 31 and 32), `section.approaches_minimum_speed` (lines 34 and 35) and `section.exceeds_nominal_speed` (lines 35, 36, 38 and 39) are derived from knowledge inferred by Pellet; arguments for `action.concurrent_action.type` (lines 11, 16 and 21) and `section.action.concurrent_action.type` (lines 30, 33 and 37) are derived from knowledge inferred by Pellet and the Prolog compiler; and arguments for `section.id` (lines 28–39) are derived from explicit domain knowledge encoded in Traffic-CEMO.

Lines 1 and 5 in Listing 4.32 use meta-code to define a for loop that generates one

property per each LIDAR action assigned to the asset in the loop header. Each property comprises a conjunction of three assertions, which verify the mission scenario described in the beginning of this section and reproduced below.

1. According to the assertion in line 2, traffic speed exceeds the nominal speed threshold of a freeway section that is monitored by the LIDAR action with identifier `action.id`.

2. Given the assertion in line 2, the system optimizes its surveillance operation by decreasing the sample rate of the monitoring LIDAR action.

3. The assertion in line 3 denotes that traffic speed descends below the minimum speed threshold of the freeway section with identifier `action.section.id`; in conjunction, the assertion in line 4 denotes that traffic speed decent occurs while the monitoring LIDAR action is in the process of what is now a prolonged (as a consequence of step 2) interval.

Listing 4.32: PCTL template for traffic surveillance

```
1 FOR action IN asset.lidar_actions DO
2 P=? [ F nse#{action.id} &
3          spd#{action.section.id}=0 &
4         !mod(d#{action.concurrent_action.id}, ci#{action.id})=0 ]
5 END
```

We note that the generic property encoded in Listing 4.32 is the raison d'être for the existence of variable `nse#{action.id}`. The DTMC template in Listing 4.31 declares and updates several occurrences of `nse#{action.id}`, which is used by the formula `ci#{action.id}`. In this context, `nse#{action.id}` duplicates data stored in, and could therefore be replaced by, variable `spd#{section.id}`. But the PCTL template uses `nse#{action.id}` (as an assertion) to verify the sequence of steps that constitute the traffic surveillance mission scenario. In particular, `nse#{action.id}` is used in conjunction with the assertion `spd#{action.section.id}=0` to verify that traffic speed exceeds the nominal speed threshold, and *subsequently* descends below the minimum speed threshold, of a specific freeway section.

The artifacts presented in this section enable PRISM to verify the occurrence of what is presumably a compromising scenario for traffic surveillance missions. This scenario is underpinned by three assertions, which signify failure of a LIDAR-based surveillance system to disseminate traffic speed data in a timely manner. Two assertions (lines 3 and 4 in Listing 4.32) verify the occurrence of the scenario unless traffic speed decent and reaction to that descent by the monitoring LIDAR action occur simultaneously (i.e., in the same time-step). Future work could extend PRISM artifacts to verify the occurrence of the scenario unless reaction to traffic speed decent by the monitoring LIDAR action occurs within a specified timeframe, whereby, for example, reaction time after calibration does not exceed reaction time before calibration. This type of analysis will probably

afford more valuable and pertinent insight into the impact of the compromising scenario on mission correctness. Future work could also develop models and properties to analyze the surveillance of multiple freeway sections by a single LIDAR action. Additional limitations and potential for future work will be considered in Chapter 7.

These considerations do not diminish the utility of the current model: PRISM uses artifacts that encode Mission 5b, one of the 58 mission plans supporting the evaluation in Chapter 6, to verify the occurrence of the scenario described above with probabilities of approximately 0.651 and 0.625 for high- and low-speed freeway sections, respectively. PRISM verifies the occurrence of a similar scenario, but one that does *not* calibrate LIDAR actions, with probabilities of approximately 0.468 and 0.525 for high- and low-speed freeway sections, respectively. These results quantify the intuitive correlation between decreased LIDAR sample rates and increased latency in the dissemination of information related to traffic speed fluctuations. Latency resulting from calibration is on average 1.391 and 1.19 times greater than latency sans calibration for high- and low-speed freeway sections, respectively. Assessing the impact of increased latency on traffic control and traffic discharge rates is beyond the scope of this thesis.

Appendix D presents DTMC and PCTL templates in their entirety. Output from these templates, and additional probabilistic results, will be presented in Chapter 5.

4.3.4 An Overview

Section 4.3 exploits PRISM's model and property specification languages to formally define DTMC and PCTL templates, respectively. PRISM templates support the synthesis of system models and the behavioral properties that need to be verified with respect to those models. The synthesis process receives data from a variety of heterogeneous sources including system specifications; explicit knowledge formalized in CEMO, and Tactical- and Traffic-CEMO; inferred knowledge derived exclusively from Pellet-based reasoning; and inferred knowledge derived from composite, Pellet- and Prolog-based, reasoning. Inferences are generated from domain- and mission-specific knowledge encoded by domain experts and mission developers, respectively.

Templates presented in Section 4.3.1 enable PRISM to calculate the probability of survival for valid assets. Each valid asset executes one or more direct threat area actions during the prosecution of a threat area incursion. PRISM quantifies the inherent risk for each incursion with synthesized code from the templates presented in Section 4.3.2. The synthesis of PRISM code from survivability and risk acceptability templates is supported, in part, by the inferred concepts `DirectThreatAreaAction` and `ValidAsset`, which are formalized in Tactical-CEMO.

Section 4.3.3 presents templates that enable PRISM to quantify the dynamic relationship between LIDAR actions and the freeway sections monitored by those actions. The synthesis of PRISM code from traffic surveillance templates is supported, in part, by the inferred concepts `HighSpeedFreewaySection` and `HighSpeedLidarAction`, and the inferred object property `monitors`, which are formalized in Traffic-CEMO.

4.4 Related Work

The work presented thus far has been broadly concerned with the (loose) integration of OWL+SWRL and Prolog, and with multi-dimensional, i.e., semantic, rule-based and behavioral, modeling in the context of the UAV domain. The following sections describe related work for each activity.

4.4.1 Integrating OWL and Prolog

Şensoy et al. integrate DL- and LP-based reasoning by embedding ontological terms—including classes, properties and individuals—in logic programs; the interpretation of ontological axioms is delegated to a semantic reasoner during the execution of those programs [42]. The resulting method, which retains the expressivity of both OWL and Prolog, has been used to support resource determination and allocation by intelligent agents [81]. With respect to the UAV domain, the method has been used by de Mel et al. to develop a solution for sensor assignment during ISTAR missions [82].

Matzner and Hitzler achieve DL-LP integration with a Prolog extension based on any-world semantics that uses special atoms to query the DL-reasoner KAON2 [43]. Programs written in this language are transformed to standard LP programs encoded in SWI-Prolog. Papadakis et al. transform OWL ontologies to SWI-Prolog code via subject-predicate-object triplets [44]. Samuel et al. achieve a similar transformation—from OWL+SWRL to Prolog—via a combination of XSLT templates and Prolog rules, which are used to enforce OWL primitives [45]. With their work, Samuel et al. claim to address OWL-LP integration issues related to, for example, class disjointness and enumeration that are considered unsolvable by Volz et al [53]. Obrst et al. use the resulting method to rapidly integrate *command-and-control* (C2) systems with new sources of information [83].

The algorithm presented by Zombori transforms a TBox encoded with the description logic $\mathcal{SHIQ}$ to a set of first-order clauses [84]. Lukácsy and Szeredi incorporate this algorithm into a reasoning system named DLog, which generates Prolog programs from first-order clauses [46]. Extensions to DLog introduce optimization techniques including partial evaluation, which avoids the execution of unary predicates with uninstantiated arguments [85]; and parallel query execution [86].

Almendros-Jiménez uses Prolog as a query language for OWL [47]. Each query, which is encoded as a Prolog goal, retrieves data and meta-data from a given ontology. Conversely, Elenius uses OWL+SWRL to generate queries for XSB Prolog, an open source implementation of the Prolog language [48]. Both approaches are limited to a subset of OWL.

As mentioned in Chapter 2, this thesis does not claim a contribution to the state of the art in OWL-LP integration. We have nevertheless developed a composite inference mechanism that is distinct from related work. This mechanism is perhaps most compatible with the method presented by Şensoy et al. Similarities include the delineation

of OWL and Prolog with respect to the syntactic features and reasoning capabilities afforded by each language. But our mechanism executes semantic and Prolog-based reasoning in strict succession while Şensoy et al. interlace semantic reasoning with the execution of Prolog programs. Related work that either transforms ontological axioms to Prolog code [44, 45, 46], or uses ontological axioms to query Prolog (or vice versa) [47, 48], is even further removed from our implementation of composite reasoning.

4.4.2 Modeling the UAV Domain

The technologies presented throughout this chapter have been used in related work to model aspects of the UAV domain. De Mel et al. and Preece et al. describe ontologies for UAV-based ISTAR missions [82, 87]. Schumann et al. use OWL to formalize civilian-oriented UAV missions comprising a combination of three possible goals: loitering, path traversal and search [88]. Valente et al. exploit OpenCyc, a general purpose knowledge base, to create ontologies for military UAV missions [89]. Schlenoff and Messina have developed a *robot ontology* that supports urban *search and rescue* (S&R) efforts by the U.S. Department of Homeland Security (DHS). The ontology encodes concepts that model *aerial robots* including *high altitude loiter robots*, *ledge access robots* and *rooftop payload drop robots* [90].

In a more general context of robotics, Amigoni and Arrigoni Neri use an ontology to achieve resource determination and allocation with respect to a multi-robot system and its assigned tasks [91]. Chella et al. describe a spatial ontology that models robotic knowledge regarding indoor office environments [92]. Ontologies are exploited by Schlenoff et al. to enhance the navigation capabilities of autonomous vehicles [93]. Gavshin and Shumik transform ontological knowledge into executable robot control code [94].

Bohn models search and destroy missions as a pursuer-evader game, where pursuing UAVs attempt to locate evasive targets [95]. Model checking is subsequently used to generate strategies that are favorable for pursuers, when those strategies exist. Webster et al. apply model checking to the verification of high-level decision making by autonomous UASs in civilian airspace [96]. Basic and more advanced UAS control systems are modeled with PROMELA and the autonomous agent language Gwendolen, respectively. The models are verified by Spin and the agent model checker AJPF against air traffic regulations issued by the UK Civil Aviation Authority.

Jeyaraman et al. use Kripke semantics to model cooperative search underpinned by Dubins paths[2] [97]. Kripke semantics are also used by Sirigineedi et al. to model UAV behavior during the execution of a distributed control strategy, where assets cooperate to monitor a road network [98]. The latter model is verified by the model checker SMV against mission properties expressed in CTL.

Our modeling effort overlaps, to some extent, with related work presented in this section. CEMO formally defines hover and path segment traversal actions, which are

[2] A Dubins path is the shortest path with bounded curvature between two points in the Euclidean plane.

similar (if not equivalent) to the loitering and path traversal goals described by Schumann et al. Tactical- and Traffic-CEMO encode military- and civilian-oriented mission concepts, respectively, and thereby resemble ontologies that formalize ISTAR and S&R missions [82, 87, 90]. But unlike related work aimed at enhancing UAV capabilities [93, 95], we use semantic and behavioral models to verify the correctness of UAV mission plans. And unlike Gavshin and Shumik, we transform ontological (and other) knowledge into PRISM code.

Ultimately, the divergence between our work and that of others is most compelling at the intersection of semantic and behavioral modeling. This divergence will be highlighted in Chapter 5.

4.5 Summary

This chapter is necessarily extensive. We exploit five formalisms—including knowledge representation languages (OWL and SWRL), a logic programming language (Prolog), a behavioral modeling language (PRISM DTMC), and a property specification language (PRISM PCTL)—to model a non-trivial application domain; in doing so, a subset of that domain is encoded precisely and from multiple perspectives. Syntactic and semantic features of each formalism, and the inherent limitations of OWL+SWRL, are described in the context of two case studies (involving tactical and traffic surveillance missions) and a running example (Mission A).

When encoded with different technologies, domain concepts serve to integrate those technologies and thereby highlight a core thesis contribution. Integrated technologies form the stack that constitutes our prototype implementation of the cascading verification method. This stack can be demarcated into three principal groups: OWL+SWRL, Prolog, and PRISM DTMC and PCTL. A loose integration of OWL+SWRL and Prolog produces strands of composite reasoning that are traced throughout this chapter. Composite inferences, and explicit knowledge encoded by domain experts and mission developers, converge to support the synthesis of system models and behavioral properties for probabilistic model checking.

The formalisms and modeling techniques that constitute our method and prototype are involved and diverse. At present, model builders are forced to develop system models with relatively low-level modeling languages. Model builders must also engage with at least some aspects of semantic modeling as they endeavor to integrate complex domain knowledge into the model checking process. The intersection of low-level modeling languages and complex domain knowledge is evidently a formidable challenge in the development of system models and their desired behavioral properties. This challenge can be mitigated with cascading verification, which leverages formal domain knowledge to support the probabilistic analysis of multiple system specifications. We proceed to elaborate our method.

Chapter 5

Cascading Verification

Cascading verification combines the technologies described in Chapter 4 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 chapter describes the prototype by tracing verification from high-level mission specifications to probabilistic model checking with PRISM. As with Chapter 4, cascading verification is illustrated with respect to Mission A, the example mission presented in Section 3.1.

The remainder of this chapter is structured as follows. Section 5.1 presents the YAML DSL used by model builders to encode mission plans and the geographic data associated with those plans. Section 5.2 describes the transformation of mission specifications into ABox assertions, and the verification of those assertions via automated semantic reasoning. Section 5.2 also describes inferred ontological knowledge, which is generated via automated reasoning from explicitly encoded domain knowledge. Section 5.3 describes the transformation of explicit and inferred ontological knowledge into Prolog facts, and the Prolog inferences resulting from those facts. Explicit domain knowledge—encoded in OWL+SWRL and Prolog—and knowledge inferred via composite reasoning is used by our prototype to synthesize DTMC and PCTL artifacts, which are presented in Section 5.4. This section also presents results from the probabilistic model checking underpinned by synthesized artifacts. The technologies and implemented components that constitute our prototype are presented in Section 5.5. Our work is related to semantic model checking, which is presented in Section 5.6. This chapter is summarized in Section 5.7.

5.1 High-Level Specifications in YAML

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. The prototype comprises a domain-specific YAML dialect that is consistent with the domain concepts formalized in CEMO. Consistency is enforced when mission specifications are transformed by the

CVC into ABox assertions, and verified via semantic reasoning with respect to the TBox defined by domain experts.

To better illustrate the YAML DSL, which specifies Mission A, we have encoded a schema definition in the Kwalify schema language [99]. This schema is partially defined in Listing 5.1.

Listing 5.1: Partial schema definition for the YAML DSL

```
1  type: map
2  required: yes
3  mapping:
4    "Action":
5      type: map
6      required: yes
7      mapping:
8        "HoverAction":
9        ...
10       "LidarAction":
11       ...
12       "PhotoSurveillanceAction":
13       ...
14       "TraversePathSegmentAction":
15       ...
16   "Asset":
17     type: map
18     required: yes
19     mapping:
20       "ARDrone":
21       ...
22       "Hummingbird":
23       ...
```

The Kwalify schema language is itself a dialect of YAML. Lines 1–3 in Listing 5.1 declare a YAML mapping, which is a primary logical structure in a YAML document (lines 5–7 and 17–19 declare similar data structures). The Kwalify keyword `required` is used in lines 2, 6 and 18 to denote constraints on each of the declared mappings.

According to Listing 5.1, the YAML DSL affords mission developers two top-level elements—`Action` and `Asset` (declared in lines 4 and 16, respectively)—that must be specified for each mission. The former contains four optional elements—`HoverAction`, `LidarAction`, `PhotoSurveillanceAction` and `TraversePathSegmentAction` (declared in lines 8–14)—while the latter contains optional elements `ARDrone` and `Hummingbird` (declared in lines 20 and 22, respectively). The YAML code in Listing 5.2 specifies the DSL element `Hummingbird`.

Lines 2 and 3 in Listing 5.2 declare a YAML sequence, which is a primary logical structure in a YAML document (lines 13–15 declare a similar data structure). According to Listing 5.2, the element `Hummingbird` contains the required element `id` and the optional element `endurance` (declared in lines 7 and 10, respectively), both of which assume values of type `int`. The element `Hummingbird` also contains the required element `actions` (declared in line 12), which is a sequence of `int` values that identify

Listing 5.2: Schema definition for the DSL element `Hummingbird`

```
"Hummingbird":
  type: seq
  sequence:
    - type: map
      required: yes
      mapping:
        "id":
          type: int
          required: yes
        "endurance":
          type: int
        "actions":
          type: seq
          required: yes
          sequence:
            - type: int
```

actions assigned to a specific `Hummingbird` asset. The elements `actions` and `endurance` correspond, respectively, to the object property `hasAction` and the datatype property `hasEnduranceInSeconds` encoded in Listing 4.1.

Appendix E presents a complete schema definition for the YAML DSL.

5.2 Verification with Semantic Reasoning

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*.[1]

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 5.3 formally defines a valid ABox individual with identifier `H1`, which corresponds to the identifier of the Hummingbird asset specified in Mission A.

Asset `H1` is related via the object property `hasAction` to actions `TPSA1` and `TPSA2` (lines 3 and 4, respectively, in Listing 5.3). Mission A specifies that action `TPSA1` is a precondition to action `TPSA2` (line 10 in Listing 3.1). Because preconditions associate actions with other actions, a precondition would be inconsistent, with respect to the

[1] A great circle is the intersection of the Earth's surface with a plane passing through the center of the Earth. The concepts that underpin geodesic equations will be elaborated in Appendix A.

Listing 5.3: OWL code for the individual `H1`

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

definition of the object property `hasPrecondition` presented in Section 4.1, if it associated individuals that were not members of class `Action`. The OWL code in Listing 5.4 specifies two valid ABox individuals with identifiers `TPSA1` and `TPSA2`, where `TPSA2` is related to `TPSA1` via `hasPrecondition` (line 6). Figure 5.1 illustrates elements of semantic verification with respect to the actions `TPSA1` and `TPSA2`, and the object property `hasPrecondition`. As a valid ABox individual, action `TPSA2` inherits from the built-in OWL class `Thing`, which is highlighted with red text (an invalid individual would inherit from the built-in OWL class `Nothing`).

Listing 5.4: OWL code for the individuals `TPSA1` and `TPSA2`

```
1 Individual: TPSA1
2   Types: TraversePathSegmentAction
3
4 Individual: TPSA2
5   Types: TraversePathSegmentAction
6   Facts: hasPrecondition TPSA1
```

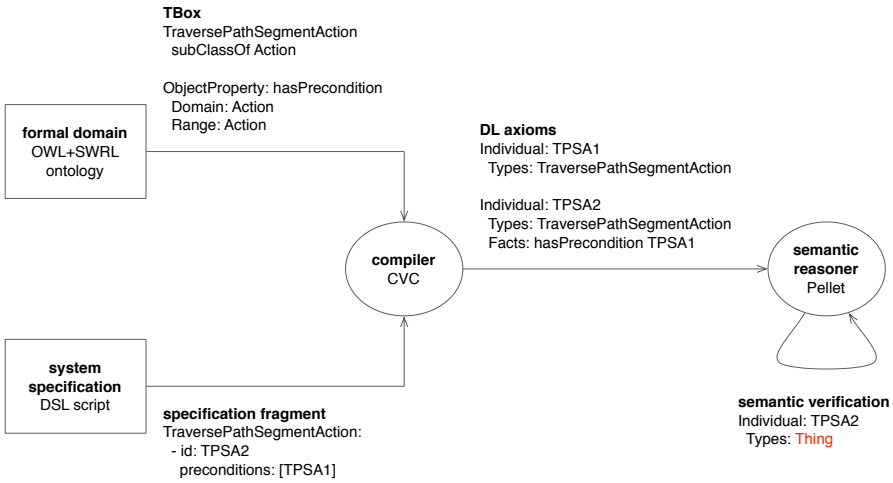

Figure 5.1: Elements of semantic verification with respect to the actions `TPSA1` and `TPSA2`, and the object property `hasPrecondition`.

With ABox consistency deduced, Pellet proceeds to generate inferences by, for example, reasoning about *realization*, which determines the direct types of each individual. Realization-based inferences were described in Section 4.1.2 with the classification of a specific kinetic action as a DTAA, and the resultant classification of the asset to which

that action was assigned as a valid asset. As an example of realization with respect to Mission A, let us assume a threat area action classification for action `TPSA4` (via the preprocessing phase described above). Given this assumption, Pellet would infer action `TPSA4` to be a DTAA because start point and endpoint of `TPSA4` are members of classes `Waypoint` (the start point of action `TPSA4` is the endpoint of action `TPSA3`, which is not a threat area action) and `ThreatAreaWaypoint`, respectively. Asset `H2` would consequently be inferred a valid asset (since `H2` executes at least one sensor action, namely `PSA5`, during the incursion).

Appendix B presents the generated ABox for Mission A.

5.3 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 5.5 formally defines rule `terminal_constrained_observer`, which encapsulates the atom `terminal`.

Listing 5.5: Prolog code for rule `terminal_constrained_observer`

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

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` (cross-cutting actions were introduced in Section 4.2). Figure 5.2 illustrates elements of composite semantic- and Prolog-based reasoning with respect to the actions `TPSA1` and `TPSA2`; the inferred object property `isPreconditionTo` presented in Section 4.2; and the rule `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.

Table 5.1 highlights the impact of kinetic action classifications with respect to synthesized DTMC and PCTL constructs. For example, the action label of the asset mod-

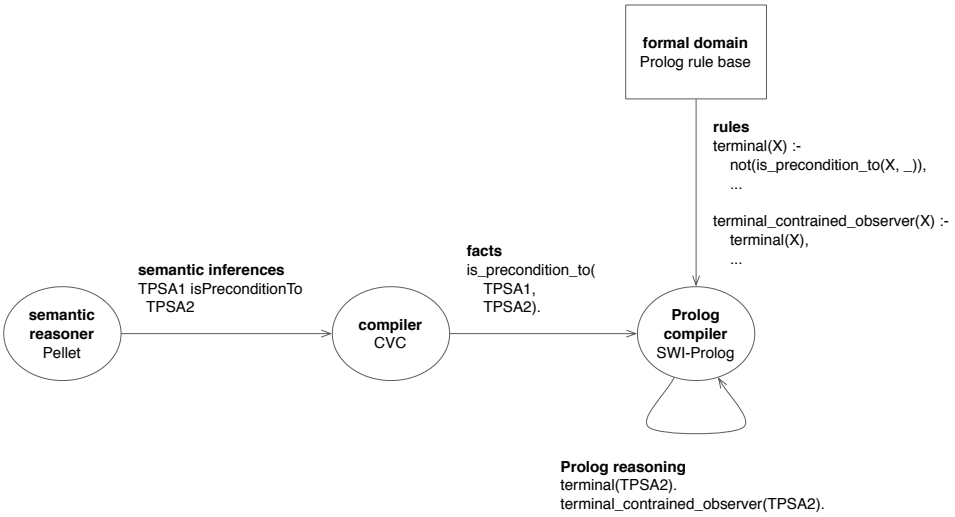

Figure 5.2: Elements of composite semantic- and Prolog-based reasoning with respect to the actions `TPSA1` and `TPSA2`; the inferred object property `isPreconditionTo` presented in Section 4.2; and the rule `terminal_constrained_observer`.

ule command associated with a `default` classification would assume the value of the identifier assigned to the classified action (as denoted by the dot notation); the action label of the command associated with a `terminal_constrained_observer` classification would assume the identifier of the asset to which the classified action is assigned; and, given the notation in Listing 4.26, the identifier of the property associated with a `terminal_constrained_observer` classification would assume the value of the identifier assigned to the classified action. Not applicable (n/a) and `null` values in Table 5.1 denote classifications that either do not affect the synthesis of a specific action label, or synthesize an empty action label, respectively. Thus Prolog inferences affect the synthesis of action labels, which in turn impact module synchronization (as described in Section 4.3) and, ultimately, the verification results returned by PRISM.

For several kinetic action classifications, action labels assume the identifier of a *primary asset* (as illustrated in Table 5.1), which may be different from the asset to which a classified action is assigned. The behavioral modeling process described in Section 4.3 was guided by two concerns. Our primary goal was to develop template models and properties that could support the verification of multiple and variable UAV mission plans. It was also our intention to accurately model domain processes and thereby achieve the greatest possible confidence in the resulting verification process. To mimic real-world asset behavior, module executions were loosely coupled and synchronizations limited to occur on an as-needed basis. But this approach is not appropriate for modeling cross-cutting actions, which must be verified with temporal precision to ensure kinetic action workflow continuity during asset operations (this issue will be elaborated in Chapter 6). The conflict between modeling and analytical accuracy occurs because enabled DTMC commands are executed by PRISM with equal probability [100]. Lack of native scheduling precision necessitates the synchronization of all modules representing concepts related to cross-cutting actions. The required synchronization is achieved with the designation,

	DTMC asset module		PCTL
kinetic action classification	action label	last action label	property id
default	action.id	n/a	n/a
default_singleton	action.id	null	action.id
default_terminal	action.id	null	action.id
subject_precondition	primary_asset.id	n/a	n/a
observer_precondition	primary_asset.id	n/a	n/a
constrained_subject	primary_asset.id	n/a	n/a
leading_subject	action.id	n/a	n/a
singleton_subject	primary_asset.id	primary_asset.id	n/a
terminal_subject	primary_asset.id	primary_asset.id	n/a
default_observer	action.id	n/a	n/a
terminal_observer	action.id	null	action.id
terminal_constrained_observer	action.asset.id	action.asset.id	action.id
observer_and_constrained_subject	primary_asset.id	n/a	n/a
observer_and_singleton_subject	primary_asset.id	primary_asset.id	n/a
observer_and_terminal_subject	primary_asset.id	primary_asset.id	n/a
subject_constraint	primary_asset.id	n/a	n/a
terminal_subject_constraint	primary_asset.id	primary_asset.id	action.id

Table 5.1: Kinetic action classifications, which are ultimately derived via Prolog inferences, and their impact on synthesized DTMC and PCTL constructs associated with classified actions.

via Prolog inferences, of a single primary asset whose identifier is assumed by the action labels associated with cross-cutting actions.

The Prolog code in Listing 5.6 formally defines rule `primary_asset`, which encapsulates inferred ontological knowledge encoded with the atoms `observer_asset` (line 2), `observed_asset` (line 4) and `observes` (lines 6 and 7). Rule `primary_asset` formalizes the concept of a primary asset as an *observer asset* (i.e., an asset observing other assets) that is either not itself observed by, or participates in a symmetric observer relationship with, another asset (a symmetric relationship is the opposite of the asymmetric relationship described in Section 4.1.2).

Listing 5.6: Prolog code for rule `primary_asset`

```
1 primary_asset(A) :-
2     observer_asset(A),
3     (
4         not(observed_asset(A));
5         (
6             observes(A, B),
7             observes(B, A)
8         )
9     ).
```

Lines 1–3 in Listing 5.7 formally define class `ObserverAsset`, which extends class `Asset`. Every member of class `ObserverAsset` must be associated via the object property `hasAction` with a member of class `Observer`. Lines 5 and 6 formally define the OWL class `Observer`, which is augmented by a synonymous SWRL rule (lines 8–13). Rule `Observer` formalizes the concept of an *observer action* as an action that observes cross-cutting actions via preconditions. Given the domain knowledge encoded in Listing 5.7, an observer asset can be described more precisely as an asset that observes other assets via an assigned action that has as precondition a cross-cutting action. Conversely, an *observed asset* is an asset observed by other assets via an assigned cross-cutting action. Class `ObservedAsset`, which formalizes the observed asset concept, is encoded in a similar manner to class `ObserverAsset`. Knowledge inferred by Pellet with respect to `ObserverAsset` and `ObservedAsset` is transformed by the CVC into knowledge encoded with the Prolog atoms `observer_asset` and `observed_asset`, respectively.

Listing 5.8 formally defines the object property `observes` (lines 1–5), which is augmented by a synonymous SWRL rule (lines 7–12). Rule `observes` uses the cross-cutting action relationship (lines 8–11) to underpin an asset-oriented subject-observer relationship. Knowledge inferred by Pellet with respect to the OWL concept `observes` is transformed by the CVC into knowledge encoded with the Prolog atom `observes`, which is used in the context of rule `primary_asset` to formalize a symmetric observer relationship. In conjunction with the observed and observer asset concepts, the *observes* concept supports SWI-Prolog inferences with respect to `primary_asset`, which in turn supports kinetic action classifications.

Rule `primary_asset` enables Prolog to discover the assets with assigned cross-cutting

Listing 5.7: OWL+SWRL code for class `ObserverAsset`

```
Class: ObserverAsset
  EquivalentTo: Asset
    and (hasAction some Observer)

Class: Observer
  SubClassOf: Action

Rule:
    hasAction(?a, ?x),
    hasAction(?b, ?y),
    hasPrecondition(?y, ?x),
    DifferentFrom (?a, ?b)
  -> Observer(?y)
```

Listing 5.8: OWL+SWRL code for the object property `observes`

```
ObjectProperty: observes
  Characteristics: Asymmetric,
    Irreflexive
  Domain: Asset
  Range: Asset

Rule:
    hasAction(?a, ?x),
    hasAction(?b, ?y),
    hasPrecondition(?y, ?x),
    DifferentFrom (?a, ?b)
  -> observes(?a, ?b)
```

actions that have temporal precedence in a workflow of related actions. Such a workflow may comprise one or, given the symmetric observer relationship encapsulated in `primary_asset`, multiple primary assets. The action label identifier resulting from a single primary asset guarantees the synchronization of all modules representing concepts related to cross-cutting actions. Synchronization is also guaranteed with multiple primary assets when a single identifier is chosen randomly from the candidate set. We note that a single mission plan comprising multiple independent action workflows could be verified by PRISM without the need for synchronization between the actions that constitute those workflows.

The primary asset mechanism may be perceived as an obfuscated method for achieving what is in essence a random result, at least with respect to a subset of the mission space. But primary assets support an inference-based selection process that is conceptually compatible with the methods that derive the remaining action- and asset-based identifiers in Table 5.1. And while this mechanism fails to eliminate randomness during the identifier selection process, it performs better than random by eliminating and minimizing randomness for mission plans comprising one or more primary assets, respectively. It is of course entirely possible for the primary asset concept, and the reasoning that it affords, to be extended or repurposed in subsequent versions of our prototype. In

any case, these concerns apply neither to our method nor our prototype as a whole, but rather to a small subset of the domain model that constitutes our prototype.

As a prelude to the following section, which presents synthesized PRISM artifacts, Table 5.2 lists kinetic action classifications for the path segment traversal actions specified in Mission A (top), and the impact of each classification on the synthesized DTMC and PCTL constructs associated with the classified action (bottom).

	TPSA1	TPSA2	TPSA3	TPSA4
default	false	false	false	false
default_singleton	false	false	false	false
default_terminal	false	false	false	false
subject_precondition	false	false	false	false
observer_precondition	true	false	false	false
constrained_subject	false	false	true	false
leading_subject	false	false	false	false
singleton_subject	false	false	false	false
terminal_subject	false	false	false	false
default_observer	false	false	false	false
terminal_observer	false	false	false	false
terminal_constrained_observer	false	true	false	false
observer_and_constrained_subject	false	false	false	false
observer_and_singleton_subject	false	false	false	false
observer_and_terminal_subject	false	false	false	false
subject_constraint	false	false	false	false
terminal_subject_constraint	false	false	false	true
asset module action label	H1	H1	H1	H1
asset module last action label	n/a	H1	n/a	H1
property id	n/a	TPSA2	n/a	TPSA4

Table 5.2: Kinetic action classifications for the path segment traversal actions specified in Mission A (top), and the impact of each classification on the synthesized DTMC and PCTL constructs associated with the classified action (bottom).

5.4 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 5.9 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.

Listing 5.9: PRISM asset module code

```
module Hummingbird2
  e2 : [0..120] init 120;
  [H1] e2>0 & d3>0 -> (e2'=e2-1);
  [H1] e2>0 & d4>0 -> (e2'=e2-1);
  [H1] e2=0 | d4=0 -> true;
endmodule
```

As a consequence of the realization-based inferences described in Section 5.2, the CVC synthesizes an asset survivability module to calculate the probability of survival for `H2` (as described in Section 4.3.1). The PRISM code in Listing 5.10 specifies the synthesized survivability module corresponding to asset `H2`, where variable `a2d` (lines 6–8) represents the asset's destruction.

Listing 5.10: PRISM asset survivability code

```
1 const int start4 = 60;
2 const int finish4 = 0;
3 formula actn4_tai = d4>finish4 & d4<=start4;
4
5 module Hummingbird2_Survivability
6   a2d : bool init false;
7   [H1] !a2d &  actn4_tai -> 0.99:(a2d'=false) + 0.01:(a2d'=true);
8   [H1]  a2d | !actn4_tai -> true;
9 endmodule
```

The CVC also synthesizes PRISM code to calculate a RAF value for the threat area incursion prosecuted by asset `H2` (as described in Section 4.3.2). The PRISM code in Listing 5.11 specifies the synthesized sensor action counter module corresponding to asset `H2`. Variables `start4`, `finish4` and `actn4_tai` in Listing 5.11 are declared in Listing 5.10.

Listing 5.11: PRISM risk acceptability code

```
1  formula duration4 = start4 - finish4;
2
3  formula tkad2 = duration4;
4
5  module SensorActionCounter2
6    sad2 : [0..tkad2] init 0;
7    [H1]  actn4_tai &  (r5) & sad2<tkad2 -> (sad2'=sad2+1);
8    [H1] !actn4_tai | !(r5) -> true;
9  endmodule
10
11 formula raf2 = sad2 / tkad2;
```

At the conclusion of the synthesis process, the DTMC artifact that models Mission A is verified against a set of synthesized PCTL properties. The PRISM code in Listing 5.12 specifies a synthesized property for Mission A. 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.

Listing 5.12: PRISM mission property code

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

Given the property in Listing 5.12, 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.

5.5 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 (version 1.9.3). And the DSL described in Section 5.1 was implemented in YAML, which is particularly compatible with Ruby. Because of its powerful reflective and meta-programming capabilities, Ruby affords our prototype flexibility to substitute the current YAML DSL with a more expressive mission specification language, if so required [101, 102].

In developing the prototype, we leveraged open source software including Pellet 2.2.0, PRISM 4.1 and the Protégé ontology editor (version 4.1.0). While integration, testing and extension of implemented components is currently ongoing, completed work was sufficient to support an evaluation, which is presented in Chapter 6.

5.6 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 [25]. 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 en-

vironment, 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 [26]. 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 [27].

Oghabi et al. use OWL-S, an OWL ontology that supersedes DAML-S, to describe Web service behavior [28]. 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 [29]. Liu et al. extend OWL-S with multiple annotation layers for specifying Web service flow properties including temporal constraints [30]. 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) [31]. 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 thesis is (to our knowledge) unique because it addresses semantic model checking limitations, and applies the resulting method to a novel domain.

5.7 Summary

Chapter 4 describes the technologies and modeling methods that underpin cascading verification; in this chapter, we describe the actual verification process, which involves several stages of reasoning and analysis. The process begins when model builders use a high-level DSL to encode system specifications. A compiler uses automated reasoning to verify the consistency between each specification and formalized domain knowledge encoded in OWL+SWRL and Prolog. If consistency is deduced, then explicit and inferred

domain knowledge is used by the compiler to synthesize a system model and behavioral properties from template code. PRISM subsequently verifies the model against the properties.

At its core, this chapter exposes the composite inference mechanism that underpins our method and prototype. Composite reasoning encompasses Pellet- and Prolog-based inference services; the former include consistency checking and realization with respect to an OWL+SWRL ontology, while the latter involve query resolution with respect to a knowledge base comprising rules and facts. Prolog-based inferences enable our prototype to classify kinetic actions, and thereby calibrate the synthesis of PRISM modules related to those actions. Our implementation of composite reasoning is illustrated with several examples in the context of Mission A, which is assumed to involve the prosecution of a threat area incursion. The discussion in this chapter thus utilizes the tactical case study introduced in Section 4.1.2.

By tracing cascading verification from system specifications to probabilistic model checking, we describe a prototype greater than the sum of its parts, i.e., a prototype that integrates technologies presented in Chapter 4 to improve the correctness of UAV mission plans. The following chapter quantifies this improvement.

Chapter 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 thesis 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.

The remainder of this chapter is structured as follows. Section 6.1 describes the methodologies and metrics that underpin the evaluation of our method and prototype. Limitations and threats to the validity of the evaluation are presented in Section 6.2. This chapter is summarized in Section 6.3.

6.1 Evaluation Methods and Metrics

This evaluation focuses on a single project, and assesses the effects of change prior to large-scale implementation and deployment. Case studies therefore constitute an appropriate evaluation method [103]. Importantly, case studies 1) avoid scalability issues associated with the evaluation of software-engineering tools and methods, and 2) support high-level assessments, which are desirable for wide-ranging process changes. The evaluation was not affected by ancillary budgetary, scheduling and staffing issues.

6.1.1 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 [32, 33]. We evaluated our approach by comparing the 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 provide preliminary evidence of non-trivial reduction in the effort required to produce mission models and properties.

Table 6.1 lists metric values for a subset of the 58 mission plans. This subset represents the full range of observed PRISM-to-YAML ratios. (Appendix F contains verification artifacts—including mission specifications encoded in YAML, synthesized DTMC and PCTL artifacts, and PRISM results—for the mission plans presented in Table 6.1.)

	YAML		PRISM		PRISM-to-YAML ratio	
id	LOC	tokens	LOC	tokens	LOC	tokens
1a	9	31	22	104	244.4%	335.5%
1d	12	45	32	169	266.7%	375.6%
1g	15	59	36	218	240.0%	369.5%
2a	18	70	45	255	250.0%	364.3%
2b	18	72	52	277	288.9%	384.7%
2e	23	97	57	331	247.8%	341.2%
2g	25	105	77	457	308.0%	435.2%
2j	23	98	76	443	330.4%	452.0%
2m	26	112	91	558	350.0%	498.2%
2p	26	114	91	562	350.0%	493.0%
2r	25	109	84	468	336.0%	429.4%
2u	28	123	94	533	335.7%	433.3%
2v	28	123	99	583	353.6%	474.0%
2w	31	137	111	666	358.1%	486.1%
2x	31	137	111	666	358.1%	486.1%
3a	12	42	32	158	266.7%	376.2%
3e	15	56	45	252	300.0%	450.0%
3h	16	61	42	227	262.5%	372.1%
3k	18	70	60	357	333.3%	510.0%
3o	18	72	60	373	333.3%	518.1%
4a	21	94	39	267	185.7%	284.0%
4b	21	94	73	513	347.6%	545.7%
4d	25	110	100	707	400.0%	642.7%
4f	28	124	111	788	396.4%	635.5%
5a	23	92	58	329	252.2%	357.6%
5b	23	92	104	744	452.2%	808.7%

Table 6.1: Metric values for representative mission plans.

We observe that tactical missions (4b, 4d and 4f in Table 6.1) and traffic surveillance missions (5a and 5b) generate more LOC and tokens than *standalone mission plans*,

which are mission plans underpinned exclusively by CEMO and thereby not associated with the more specialized subdomains encoded in Tactical- and Traffic-CEMO. 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.) Mission 5b generates PRISM code that is 4.522 and 8.087 times greater than the size of YAML input with regard to LOC and tokens, 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 and traffic surveillance 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.

6.1.2 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). 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.

In the context of the PRISM language and the PRISM models created for this project:

- *Inconsistent variable errors* occur when PRISM variables are inconsistent in value with corresponding variables in mission specifications; for example, the duration of an action specified in YAML should be consistent in value with the local variable of the PRISM module corresponding to that action.[1]
- *Command action errors* occur when command actions, which affect module synchronization, are incorrect across two or more modules.
- *Command probability errors* occur when commands are annotated with probabilities that fail to accurately reflect the system being modeled; for example, the probabilities encapsulated in an asset survivability module should accurately reflect the vulnerability of the asset corresponding to that module.[2]
- *Command update errors* occur when command updates, which affect module behavior, are incomplete or incorrect. For example, an incomplete update could fail

[1] Action duration in the context of behavioral modeling was described in Section 4.3.
[2] Asset survivability was described in Section 4.1.2.

to couple the modules corresponding to an asset and the action assigned to that asset; an incorrect update could couple modules corresponding to an asset and an action assigned to a different asset.

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. This error type affects mission correctness by ambiguating mission constructs for humans and, potentially, computers. Construct ambiguity could, for example, cause humans to confuse action subtypes and thereby develop mission plans comprising one or more sensor actions and zero kinetic actions; without appropriate verification, automated processes could subsequently attempt to deploy erroneous missions that violate asset constraints related to the execution of kinetic actions (as described in Section 4.1). The OWL statements in Listing 6.1 formally define disjoint classes that can cause domain-specific errors.

Listing 6.1: Disjoint class statements specified in OWL

```
1 Action DisjointWith Asset
2 ARDrone DisjointWith Hummingbird
3 HoverAction DisjointWith TraversePathSegmentAction
4 HoverAction DisjointWith LidarAction
5 HoverAction DisjointWith PhotoSurveillanceAction
6 TraversePathSegmentAction DisjointWith LidarAction
7 TraversePathSegmentAction DisjointWith PhotoSurveillanceAction
8 LidarAction DisjointWith PhotoSurveillanceAction
```

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. The OWL statements in Listing 6.2 formally define existential restrictions that can cause domain-specific errors. Because of OWA, which was described in Section 2.1, OWL-based reasoning cannot conclude the absence of a mandatory relationship from the absence of knowledge about that relationship. Consequently, existential restriction errors are detected exclusively by the SWI-Prolog compiler. The Prolog rules in Listing 6.3 encompass atoms that correspond to the existential restrictions in Listing 6.2.

Data property value errors occur when data property values declared in system specifications fall outside the ranges of corresponding data properties encoded in CEMO; for example, the endurance of a Hummingbird is 1200 time-steps. The OWL statements in Listing 6.4 formally define data properties that can cause domain-specific errors.

Listing 6.2: Existential restriction statements specified in OWL

```
Asset hasAction some KineticAction
Asset hasEnduranceInSeconds some int
HoverAction hasWaypoint some Waypoint
KineticAction hasDurationInSeconds some int
LidarAction hasIntervalInSeconds some int
LidarAction isConcurrentWith some HoverAction
Mission hasAsset some Asset
PhotoSurveillanceAction hasDurationInSeconds some int
TraversePathSegmentAction hasEndpoint some Waypoint
TraversePathSegmentAction hasStartPoint some Waypoint
```

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`. The OWL code in Listing 6.5 formally defines data properties that can cause domain-specific errors.

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; 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). The OWL code in Listing 6.6 formally defines object properties that can cause domain-specific errors.

Threatened asset errors occur when mission plans comprise a threatened asset that is not also a valid asset.[3] With the exception of this UAV-related error class, domain-specific errors are caused by inconsistencies between a mission specification and the underlying domain model. Given adequate documentation of the YAML DSL, these inconsistencies could, in theory, be eliminated by mission developers during the design and implementation of mission plans (in practice, software bugs elude extinction). But threatened and valid assets can only be discovered with geodetic calculations and subsequent formal reasoning, as described in Section 3.2 and Section 4.1.2, respectively. The OWL code in Listing 6.7 formally defines class `ThreatenedAsset`.

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.

[3] Threatened and valid assets were described in Section 4.1.2.

Listing 6.3: Existential restrictions encompassed in Prolog rules

```
invalid_asset(A) :-
    zero_action_asset(A);
    not(has_endurance_in_seconds(A, _)).

invalid_hover_action(H) :-
    invalid_kinetic_action(H);
    not(has_waypoint(H, _)).

invalid_kinetic_action(K) :-
    not(has_duration_in_seconds(K, _)).

invalid_lidar_action(L) :-
    not(has_interval_in_seconds(L, _));
    not(is_concurrent_with(L, _)).

invalid_mission(M) :-
    not(has_asset(M, _)).

invalid_photo_surveillance_action(P) :-
    not(has_duration_in_seconds(P, _)).

invalid_traverse_path_segmentAction(T) :-
    invalid_kinetic_action(T);
    not(has_endpoint(T, _));
    not(has_start_point(T, _)).
```

Listing 6.4: Data property statements specified in OWL

```
ARDrone hasEnduranceInSeconds some int[<= 70]
Hummingbird hasEnduranceInSeconds some int[<= 120]
```

6.1.3 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.[4]

Nine mission plans failed because kinetic or sensor action workflow durations exceeded the endurances of the assets to which those workflows were assigned. Figure 6.1 illustrates this type of error with Mission 3g, one of the 58 mission plans supporting our evaluation. Mission 3g comprises one asset with identifier `ARD1`, two hover actions with identifiers `HA1` and `HA2`, and one photo surveillance action with identifier `PSA3`. The mission fails because the duration of `PSA3` exceeds the endurance of `ARD1`, to which `PSA3` is assigned.

Finally, eight mission plans contained action workflow errors that resulted in deadlock. Figure 6.2 illustrates this type of error with Mission 2d, which also belongs to

[4]The RAF value was described in Section 5.2.

Listing 6.5: Data properties specified in OWL

```
DataProperty: hasDurationInSeconds
  Characteristics: Functional
  Domain: Action
  Range: int

DataProperty: hasEnduranceInSeconds
  Characteristics: Functional
  Domain: Asset
  Range: int

DataProperty: hasIntervalInSeconds
  Characteristics: Functional
  Domain: LidarAction
  Range: int
```

the set of missions supporting our evaluation. Mission 2d comprises two assets with identifiers `ARD1` and `H1`, and three hover actions with identifiers `HA1`–`HA3`. The mission becomes deadlocked, and consequently fails, when the action workflow comprising `HA1` and `HA2` is disrupted. This disruption occurs because:

- `HA1` and `HA2` are kinetic actions assigned to asset `ARD1`;
- `HA1` and the cross-cutting action `HA3`, which is assigned to asset `H1`, are preconditions to `HA2`;
- the duration of `HA3` is greater than the duration of `HA1`.

The propensity for cross-cutting actions to produce deadlock and thereby compromise mission correctness is associated with the degree of coupling between kinetic actions assigned to the same asset. The type of deadlock observed in Mission 2d is, to some extent, a consequence of tight coupling between the actions `HA1` and `HA2`, which are required by our behavioral model to execute continuously during the operation of `ARD1`. By eliminating the requirement for continuity, a looser coupling could permit `HA2` to begin its execution after the end of `HA3`. But missions comprising loosely coupled kinetic actions would still have to account for side effects resulting from potential workflow disruptions. For example, a loose coupling between `HA1` and `HA2` would enable the execution of `HA3` to impose a hiatus on the workflow assigned to `ARD1`, with implications for the operation of that asset. We have chosen to avoid these implications, whose resolution would not have been straightforward, by establishing the deadlock in Mission 2d as a mission specification error.

The errors presented in this section 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 5.4).

Listing 6.6: Object properties specified in OWL

```
ObjectProperty: hasAction
  Characteristics: Asymmetric, Irreflexive, InverseFunctional
  Domain: Asset
  Range: Action
  InverseOf: isActionOf

ObjectProperty: hasAsset
  Characteristics: Asymmetric, Irreflexive, InverseFunctional
  Domain: Mission
  Range: Asset

ObjectProperty: hasPrecondition
  Characteristics: Transitive
  Domain: Action
  Range: Action
  InverseOf: isPreconditionTo

ObjectProperty: isConcurrentWith
  Characteristics: Asymmetric, Irreflexive
  Domain: LidarAction
  Range: HoverAction
```

Listing 6.7: OWL code for class `ThreatenedAsset`

```
Class: ThreatenedAsset
  EquivalentTo: Asset
    and (hasAction some DirectThreatAreaHoverAction)
  DisjointWith: DirectThreatAreaTPSA
```

6.1.4 Proof of Correctness

Given the broad range of disparate techniques described throughout this thesis, a formal proof of correctness for cascading verification may not be feasible. But correctness can be evaluated against simulations carried out during the behavioral modeling process. To this end, simulations were carried out during the development of the 58 mission plans. Simulated model executions are facilitated by the PRISM *simulator*, a tool that generates sample paths through PRISM models. Partial simulation results for Mission A are plotted by the graph in Figure 6.3, where x-axis and y-axis represent variable values and time during specific executions, respectively; and plotted values represent endurance for asset `H1` (variable `e1`) and duration for the actions assigned to that asset (variables `d1` and `d2`).

6.2 Threats to Validity

This evaluation has thus far been concerned with the abstraction and effectiveness afforded by our method and prototype. Research can be evaluated further by considering threats to its validity. Threat types applicable to our research include conclusion validity,

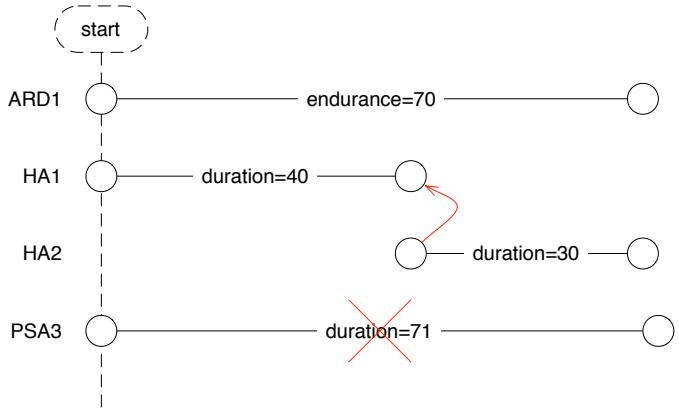

Figure 6.1: An illustration of Mission 3g, with red arrows representing preconditions and horizontal lines representing the passage of time. The *x mark* denotes a point of failure for the mission.

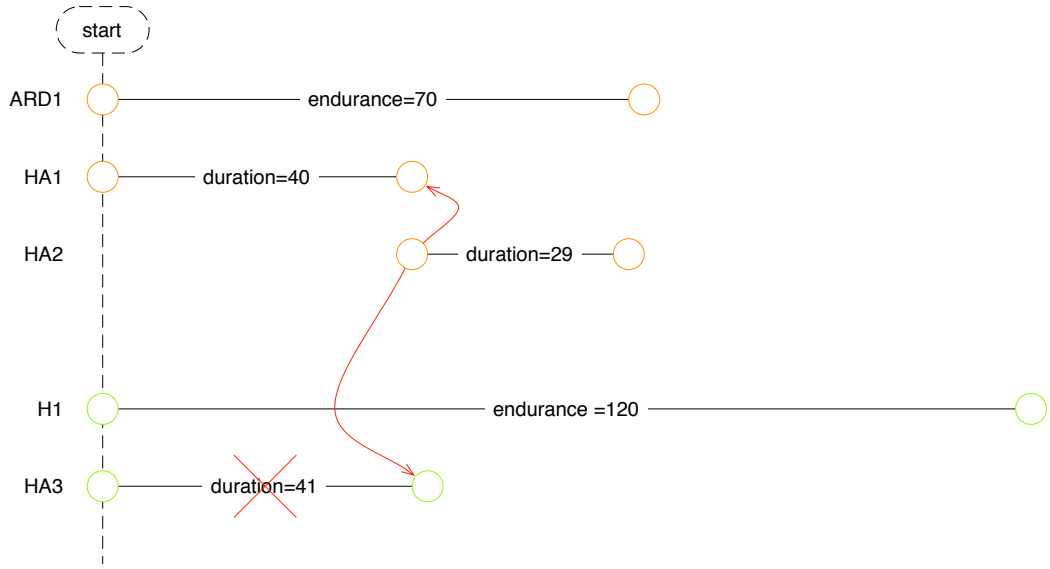

Figure 6.2: An illustration of Mission 2d, with red arrows representing preconditions and horizontal lines representing the passage of time. The *x mark* denotes a point of failure for the mission. Color coded circles delineate the operation and execution of assets and actions, respectively, and group actions and the assets to which those actions are assigned.

internal validity, construct validity and external validity [104, 105].

Conclusion validity addresses the issue of correlation between the functionality afforded by our prototype and the correctness of UAV mission specifications, whereby correctness is determined with respect to specific properties of interest. We believe this correlation to be established by the results presented in Section 6.1. To recap, in conjunction with the verification provided by PRISM, reduced LOC, tokens and potential (domain-specific and behavioral modeling) errors demonstrate that our method and prototype have a significant and positive effect on the correctness of UAV mission specifications.

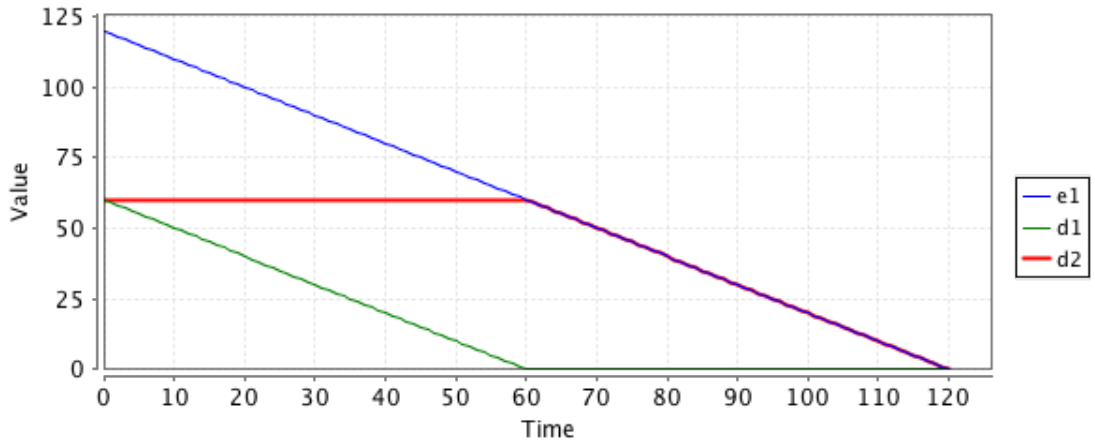

Figure 6.3: Partial simulation results for Mission A. The x-axis and y-axis represent variable values and time during specific executions, respectively; and plotted values depict endurance for asset `H1` (variable `e1`) and duration for the actions assigned to that asset (variables `d1` and `d2`).

6.2.1 Internal Validity

Having established correlation, we proceed to establish internal validity, which addresses the issue of causation between the functionality afforded by our prototype and the correctness of UAV mission specifications. For correlated variables x and y, it can be claimed that x causes y if x precedes y in the absence of *confounding factors* [106]. We will not elaborate on the issue of precedence except to mention that our prototype's functionality precedes verification results returned by PRISM. The problem of confounding is somewhat more challenging.

Confounding can occur when unknown or extraneous factors affect the relationship between correlated variables. With well defined input and output interfaces, our prototype is not likely to be influenced by unknown factors. But the 58 mission plans that act as input to the prototype, and thereby generate the established correlation, could constitute an extraneous factor if they fail to 1) be grounded in the real-world, or 2) sample a sufficiently large subset of the UAV mission state space. The former concern is mitigated by the real-world mission scenarios underpinning our mission plans. The latter concern is related to the issue of variability.

We do not attempt to achieve variability with, for example, randomized mission parameters. Nevertheless, mission plans vary with respect to LOC and token values, as indicated in Table 6.1. Mission plans also vary along several parameters, which are listed in Table 6.2 for a subset of the 58 mission plans. These parameters include numbers of assets; kinetic, sensor and cross-cutting actions; and preconditions and concurrencies. Concerns regarding the potential for mission plans to act as a confounding factor are therefore mitigated by the observed variability in both metric values and mission parameters.

The size of DTMC and PCTL templates could constitute a second extraneous factor if PRISM-to-YAML ratios are affected by poor template design rather than the functionality afforded by our prototype. But this is not the case: our templates are not

		actions			*dependencies*	
id	assets	kinetic	sensor	cross-cutting	preconditions	concurrencies
1a	1	1	0	0	0	0
1d	1	2	0	0	1	0
1g	1	3	0	0	2	0
2a	2	3	0	0	1	0
2b	2	3	0	1	2	0
2e	3	4	0	2	3	0
2g	2	5	0	2	5	0
2j	2	5	0	1	4	0
2m	2	6	0	1	5	0
2p	2	6	0	2	6	0
2r	3	5	0	2	4	0
2u	3	6	0	2	5	0
2v	3	6	0	2	5	0
2w	3	7	0	2	6	0
2x	3	7	0	2	6	0
3a	1	1	1	0	0	0
3e	1	2	1	0	1	0
3h	1	2	1	0	2	0
3k	1	2	2	0	2	0
3o	1	2	2	0	3	0
4a	1	4	0	0	3	0
4b	1	4	0	0	3	0
4d	1	4	1	0	4	0
4f	1	4	2	0	5	0
5a	1	3	2	0	2	2
5b	1	3	2	0	2	2

Table 6.2: Parameters—including number of assets, actions and action dependencies—for the representative mission plans listed in Table 6.1.

designed to create code bloat, but rather to generate PRISM code that is intelligible to a human model builder. We also note that metric values in Table 6.1 do not account for programmer-readable comments.

6.2.2 Construct Validity

Construct validity addresses the metrics, including LOC and token count, used to quantify model and property specification complexity. These are widely applicable, language independent software metrics that ignore logic structure and control flow. The abil-

ity to bypass control flow is useful when evaluating complexity in the context of the PRISM language, a somewhat unconventional formalism that lacks appropriate control structures.

We do acknowledge that a LOC- and token-based analysis cannot in and of itself account for the intricate syntax of the PRISM language. To address this limitation, we propose a multidimensional complexity model [107], which encompasses domain-specific and behavioral modeling errors prevented by our prototype. The (potentially bidirectional) correlation between error rates and software complexity suggests that software errors can be used as descriptors of complexity [108, 109]. By considering complexity from multiple dimensions, our metric- and error-based analysis provides a more complete understanding of PRISM artifact complexity, which in turn increases confidence in the resulting evaluation.

6.2.3 External Validity

Finally, we consider external validity, which addresses the issue of transferability/portability. In conjunction with the complexity of the UAV domain, and the real-world DARPA and DRDC mission scenarios underpinning this evaluation, the results presented in Section 6.1 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 were presented in Section 5.6). The successful application of these methods to the Web service domain further supports the portability of cascading verification.

6.3 Summary

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.

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.

Chapter 7

Conclusions and Future Work

This thesis 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.

The remainder of this chapter is structured as follows. Section 7.1 reiterates our contributions to the state of the art in semantic model checking. Section 7.2 discusses directions for future work. Two specific areas of research, involving network centric and annotation-guided model checking, are described in Section 7.2.1 and Section 7.2.2, respectively.

7.1 Contributions

With cascading verification, we claim several contributions to semantic model checking, a method that leverages semantic reasoning over domain knowledge to augment the model checking process. Unlike related work, our method synthesizes both system models *and* behavioral properties for probabilistic model checking. Cascading verification is underpinned by a composite DL- and LP-based inference mechanism that overcomes expressive and reasoning limitations in the ontology language OWL. By using our method to verify UAV missions, we highlight the potential portability of cascading verification and, ultimately, semantic model checking, which has thus far been applied exclusively to the Web services domain.

We illustrate cascading verification with a prototype system that verifies the correctness of 58 UAV mission plans; the development of those plans is structured with DSM. 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. For traffic-surveillance missions, the prototype realizes even bigger reductions with synthesized PRISM code that is 4.522 and 8.087 times greater than the size of YAML input with regard to LOC and tokens, respectively. These results provide preliminary evidence of non-trivial reduction in the effort required to produce mission models and properties.

LOC- and token-based metrics are used to evaluate the abstraction, i.e., the reduction in modeling complexity, afforded by our prototype. In addition to enhanced abstraction, the prototype augments PRISM's verification capabilities and thereby enhances the effectiveness of probabilistic model checking. We evaluate effectiveness by presenting errors that can only be effectively eliminated with the automated synthesis of PRISM artifacts. We also evaluate the utility of the DTMC and PCTL artifacts synthesized by our prototype.

7.2 Future Work

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. The effects of this *pipeline* architecture were particularly pronounced in Section 5.3, where semantic reasoning underpinned Prolog-based classifications, which subsequently impacted the synthesis of DTMC and PCTL artifacts. While conceptually and practically appealing, an 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.

We also 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 thesis. And we intend to confirm the portability of cascading verification by applying our method to other significant application domains. We expect a more robust evaluation process to facilitate the abstraction and formal specification of the connections that link different technologies in the context of our method. Once formalized, these connections will likely support the consistency management process described in the preceding paragraph, and the development of a domain-agnostic compiler. Such a compiler would receive as input the artifacts and connections that constitute a domain-specific knowledge base and thereby eliminate the current de facto requirement for bespoke implementations of the CVC.

Our work has yet to address the problem of tracing PRISM results back to the un-

derlying system specifications. Without traceability, the analysis provided by PRISM may be incomprehensible to model builders [110]. In the context of the YAML DSL, traceability would serve to disambiguate the relationship between syntactic rules and operational semantics, which are defined with OWL and the PRISM language, respectively. Once formalized, elements of this relationship will likely parallel the connections described in the preceding paragraph.

Future work discussed thus far is closely related to the research, development, evaluation and outcomes presented in this thesis. The following sections present research directions that are more expansive in scope.

7.2.1 Network-Centric Model Checking

Network-Centric Operations (NCO) is a doctrine that leverages information technology to improve the effectiveness and efficiency of military operations [111]. NCO is underpinned by contemporary socio-technological advancements, and enabled by "a high-performance information grid, access to all appropriate information sources, weapons reach and maneuver with precision and speed of response, value-adding C2 processes—to include high-speed automated assignment of resources to need—and integrated sensor grids closely coupled in time to shooters and C2 processes." [18] When combined, these elements support *speed of command*, the process by which superior information is turned into competitive advantage. Speed of command can be substantially enhanced when command-and-control processes are automated. Enhanced speed of command accelerates the *observe, orient, decide and act* (OODA) loop, which denies the enemy operational pause. Regaining this time amplifies the effects associated with speed of command, resulting in an accelerated rate of change that leads to enemy lock-out.

By automating the organization and utilization of complex operational knowledge, cascading verification could support the analysis and deployment of mission plans comprising asset configurations derived from real-time operational data including asset location, fuel and weapon statuses. Near real-time coupling of mission verification and deployment has the potential to yield a near real-time OODA loop, which will inevitably be susceptible to network and processing speed latencies. Addressing the impact of latency on mission correctness in the context of NCO constitutes an interesting research direction.

7.2.2 Annotation-Guided Model Checking

With our prototype implementation of cascading verification, model builders encode mission specifications in a YAML DSL. Notwithstanding the inherent advantages of YAML, the introduction of a novel declarative formalism can be associated with potential disadvantages. Declarative programming that is based on first-order or higher-order logics does not cope well with temporal systems [112]. This limitation, which derives from the static model theory that defines standard logics, impacts the level of expressivity afforded to mission developers. Furthermore, the novelty of our formalism increases the

complexity for potential adopters, and distances the project from real-world grounding, thereby potentially compromising the utility and evaluation of our prototype.

URBI and the Robot Operating System (ROS) are sophisticated, cross-platform and open-source frameworks that support robotic software development. Both frameworks are interoperable with established programming languages, including C++ and Java, that mitigate the aforementioned limitations. With Java-encoded missions, Java annotations could be used by model builders to embed domain knowledge in executable code, and thereby guide the automated synthesis of verification artifacts. Annotation-based verification frameworks and other related work indicate this to be a promising direction for future work [113, 114, 115]. Development and evaluation of the proposed annotation framework would be informed by, and thereby benefit from, our experience with the existing YAML DSL.

References

[1] "UAV Planner." [online] Available at: http://www.orbitlogic.com/products/uav.php.

[2] "Research Projects: MissionLab." [online] Available at: http://www.cc.gatech.edu/ai/robot-lab/research/MissionLab/.

[3] D. J. Nowak, I. Price, and G. B. Lamont, "Self Organized UAV Swarm Planning Optimization for Search and Destroy Using SWARMFARE Simulation," in *Proceedings of the 39th Conference on Winter Simulation: 40 Years! The Best Is Yet to Come*, WSC '07, pp. 1315–1323, IEEE Press, 2007.

[4] J. Perron, J. Hogan, B. Moulin, J. Berger, and M. Bélanger, "A Hybrid Approach Based on Multi-Agent Geosimulation and Reinforcement Learning to Solve a UAV Patrolling Problem," in *Proceedings of the 40th Conference on Winter Simulation*, WSC '08, pp. 1259–1267, Winter Simulation Conference, 2008.

[5] D. J. Nowak, G. B. Lamont, and G. L. Peterson, "Emergent Architecture in Self Organized Swarm Systems for Military Applications," in *Proceedings of the 2008 GECCO Conference Companion on Genetic and Evolutionary Computation*, GECCO '08, pp. 1913–1920, ACM, 2008.

[6] E. Zitzler, M. Laumanns, and S. Bleuler, "A Tutorial on Evolutionary Multiobjective Optimization," in *Metaheuristics for Multiobjective Optimisation* (X. Gandibleux, M. Sevaux, K. Sörensen, and V. T'kindt, eds.), vol. 535 of *Lecture Notes in Economics and Mathematical Systems*, pp. 3–37, Springer Berlin Heidelberg, 2004.

[7] A. Agogino, C. HolmesParker, and K. Tumer, "Evolving Large Scale UAV Communication System," in *Proceedings of the Fourteenth International Conference on Genetic and Evolutionary Computation Conference*, GECCO '12, pp. 1023–1030, ACM, 2012.

[8] D. W. Stouch, E. Zeidman, M. Richards, K. D. McGraw, and W. Callahan, "Coevolving Collection Plans for UAS Constellations," in *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11, pp. 1691–1698, ACM, 2011.

[9] B. Rosenberg, M. Richards, J. T. Langton, S. Tenenbaum, and D. W. Stouch, "Applications of Multi-objective Evolutionary Algorithms to Air Operations Mission Planning," in *Proceedings of the 2008 GECCO Conference Companion on Genetic and Evolutionary Computation*, GECCO '08, pp. 1879–1886, ACM, 2008.

[10] A. J. Pohl and G. B. Lamont, "Multi-Objective UAV Mission Planning Using Evolutionary Computation," in *Proceedings of the 2008 Winter Simulation Conference*, WSC '08, pp. 1268–1279, Winter Simulation Conference, 2008.

[11] V. K. Shetty, M. Sudit, and R. Nagi, "Priority-Based Assignment and Routing of a Fleet of Unmanned Combat Aerial Vehicles," *Computers and Operations Research*, vol. 35, pp. 1813–1828, June 2008.

[12] D. K. Ahner, A. H. Buss, and J. Ruck, "Assignment Scheduling Capability for Unmanned Aerial Vehicles: A Discrete Event Simulation With Optimization in the Loop Approach to Solving a Scheduling Problem," in *Proceedings of the 38th Conference on Winter Simulation*, WSC '06, pp. 1349–1356, Winter Simulation Conference, 2006.

[13] Y. Alver, M. Ozdogan, and E. Yucesan, "Assessing the Robustness of UAV Assignments," in *Proceedings of the 2012 Winter Simulation Conference*, WSC '12, pp. 1–11, Winter Simulation Conference, 2012.

[14] F. Kamrani and R. Ayani, "Simulation-Aided Path Planning of UAV," in *Proceedings of the 39th Conference on Winter Simulation: 40 Years! The Best Is Yet to Come*, WSC '07, pp. 1306–1314, IEEE Press, 2007.

[15] J. J. Corner and G. B. Lamont, "Parallel Simulation of UAV Swarm Scenarios," in *Proceedings of the 36th Conference on Winter Simulation*, WSC '04, pp. 355–363, Winter Simulation Conference, 2004.

[16] Z. Lian and A. Deshmukh, "Performance Prediction of an Unmanned Airborne Vehicle Multi-Agent System," *European Journal of Operational Research*, vol. 172, no. 2, pp. 680–695, 2006.

[17] S. Hamilton, C. T. Schmoyer, and J. A. "Drew" Hamilton, Jr., "Validating a Network Simulation Testbed for Army UAVs," in *Proceedings of the 39th Conference on Winter Simulation: 40 Years! The Best Is Yet to Come*, WSC '07, pp. 1300–1305, IEEE Press, 2007.

[18] A. K. Cebrowski and J. J. Garstka, "Network-Centric Warfare: Its Origin and Future," *U.S. Naval Institute Proceedings*, vol. 124, no. 1, pp. 28–35, 1998.

[19] G. C. Chasparis and J. S. Shamma, "Linear-Programming-Based Multi-Vehicle Path Planning With Adversaries," in *Proceedings of the 2005 American Control Conference*, ACC '05, pp. 1072–1077, June 2005.

[20] M.-W. Jang and G. Agha, "Scalable Agent Distribution Mechanisms for Large-Scale UAV Simulations," in *International Conference on Integration of Knowledge Intensive Multi-Agent Systems*, KIMAS '05, pp. 85–90, April 2005.

[21] J. Russo, M. Amduka, K. Pedersen, R. Lethin, B. Gelfand, J. Springer, R. Manohar, and R. Melhem, "Enabling Cognitive Architectures for UAV Mission Planning," in *Proceedings of the High Performance Embedded Computing Workshop*, HPEC '06, September 2006.

[22] F. Zervoudakis, D. S. Rosenblum, S. Elbaum, and A. Finkelstein, "Cascading Verification: An Integrated Method for Domain-Specific Model Checking," in *Proceedings of the 2013 9th Joint Meeting on the Foundations of Software Engineering*, ESEC/FSE '13, pp. 400–410, ACM, 2013.

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

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

[33] 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.

[34] 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.

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

[36] M. Mernik, J. Heering, and A. M. Sloane, "When and How to Develop Domain-Specific Languages," *ACM Computing Surveys*, vol. 37, pp. 316–344, December 2005.

[37] J. E. Rivera, F. Durán, and A. Vallecillo, "Formal Specification and Analysis of Domain Specific Models Using Maude," *Simulation*, vol. 85, pp. 778–792, November 2009.

[38] M. A. Musen, "Ontology-Oriented Design and Programming," in *Knowledge Engineering and Agent Technology*, pp. 3–16, IOS Press, 2004.

[39] T. Walter, F. S. Parreiras, and S. Staab, "*OntoDSL*: An Ontology-Based Framework for Domain-Specific Languages," in *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, MODELS '09, pp. 408–422, Springer-Verlag, 2009.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

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

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] 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.

[62] United States Department of Defense, "Unmanned Aircraft Systems Roadmap 2005–2030," 2005.

[63] M. Arjomandi, *Classification of Unmanned Aerial Vehicles*. The University of Adelaide.

[64] D. Crocker, *Dictionary of Aviation*. A&C Black Publishers Ltd, second ed., March 2007.

[65] AscTec, "Manufacturer and Innovator of Aerial Imaging and Research UAVs—Ascending Technologies." [online] Available at: http://www.asctec.de/.

[66] Draganfly, "Draganfly.com Industrial Aerial Video Systems & UAVs." [online] Available at: http://www.draganfly.com/.

[67] Parrot, "Parrot USA." [online] Available at: http://www.parrot.com/.

[68] C. E. Nehme, M. Cummings, and J. W. Crandall, *A UAV Mission Hierarchy*. Massachusetts Institute of Technology, 2006.

[69] M. Hou and R. D. Kobierski, "Intelligent Adaptive Interfaces," tech. rep., Defence Research and Development Canada, December 2006.

[70] M. Hou, H. Zhu, M. Zhou, and G. G. Arrabito, "Optimizing Operator—Agent Interaction in Intelligent Adaptive Interface Design: A Conceptual Framework," *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, vol. 41, pp. 161–178, March 2011.

[71] 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.

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

[73] J. Bao and V. Honavar, "Adapt OWL as a Modular Ontology Language," in *Proceedings of the OWLED*06 Workshop on OWL: Experiences and Directions*, CEUR-WS.org, 2006.

[74] S. Staab and R. Studer, *Handbook on Ontologies*. Springer, second ed., 2009.

[75] S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein, "OWL Web Ontology Language Reference." [online] Available at: http://www.w3.org/TR/owl-ref/, February 2004.

[76] 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.

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

[78] Joint Technical Coordinating Group on Aircraft Survivability (JTCG/AS), "JTCG/AS Aerospace Systems Survivability Handbook Series," Volume 1. Handbook Overview, United States Department of Defense, May 2001.

[79] M. J. Cassidy, S. B. Anani, and J. M. Haigwood, "Study of Freeway Traffic Near an Off-Ramp," *Transportation Research Part A: Policy and Practice*, vol. 36, no. 6, pp. 563–572, 2002.

[80] M. U. Piracha, D. Nguyen, D. Mandridis, T. Yilmaz, I. Ozdur, S. Ozharar, and P. J. Delfyett, "Range Resolved LIDAR for Long Distance Ranging With Sub-Millimeter Resolution," *Optics Express*, vol. 18, pp. 7184–7189, March 2010.

[81] M. Şensoy, W. W. Vasconcelos, T. J. Norman, and K. Sycara, "Reasoning Support for Flexible Task Resourcing," *Expert Systems With Applications*, vol. 39, pp. 1998–2010, February 2012.

[82] G. de Mel, M. Şensoy, W. Vasconcelos, and T. J. Norman, "A Hybrid Reasoning Mechanism for Effective Sensor Selection for Tasks," *Engineering Applications of Artificial Intelligence*, vol. 26, pp. 873–887, February 2013.

[83] L. Obrst, S. Stoutenburg, D. McCandless, D. Nichols, P. Franklin, M. Prausa, and R. Sward, "Ontologies for Rapid Integration of Heterogeneous Data for Command, Control, & Intelligence," in *Proceedings of the 2010 Conference on Ontologies and Semantic Technologies for Intelligence*, pp. 71–89, IOS Press, 2010.

[84] Z. Zombori, "Efficient Two-Phase Data Reasoning for Description Logics," in *Artificial Intelligence in Theory and Practice II*, vol. 276 of *IFIP '08*, pp. 393–402, Springer US, 2008.

[85] G. Lukácsy, P. Szeredi, and B. Kádár, "Prolog Based Description Logic Reasoning," in *Proceedings of the 24th International Conference on Logic Programming*, ICLP '08, pp. 455–469, Springer-Verlag, 2008.

[86] G. Lukácsy and P. S. and, "Scalable Web Reasoning Using Logic Programming Techniques," in *Proceedings of the 3rd International Conference on Web Reasoning and Rule Systems*, RR '09, pp. 102–117, Springer-Verlag, 2009.

[87] A. Preece, M. Gomez, G. de Mel, W. Vasconcelos, D. Sleeman, S. Colley, G. Pearson, T. Pham, and T. L. Porta, "Matching Sensors to Missions Using a Knowledge-Based Approach," in *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, 2008.

[88] M. Ferraro, J. Scanlan, H. Fangohr, and B. Schumann, "A Generic Unifying Ontology for Civil Unmanned Aerial Vehicle Missions," in *12th AIAA Aviation Technology, Integration, and Operations (ATIO) Conference and 14th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*, 2012.

[89] A. Valente, D. Holmes, and F. C. Alvidrez, "Using Ontologies to Build Web Service-Based Architecture for Airspace Systems," in *Proceedings of the 8th International Protégé Conference*, 2005.

[90] C. Schlenoff and E. Messina, "A Robot Ontology for Urban Search and Rescue," in *Proceedings of the 2005 ACM Workshop on Research in Knowledge Representation for Autonomous Systems*, KRAS '05, pp. 27–34, ACM, 2005.

[91] F. Amigoni and M. A. Neri, "An Application of Ontology Technologies to Robotic Agents," in *Proceedings of the 2005 IEEE/WIC/ACM International Conference on Intelligent Agent Technology*, pp. 751–754, IEEE Computer Society, 2005.

[92] A. Chella, M. Cossentino, R. Pirrone, and A. Ruisi, "Modeling Ontologies for Robotic Environments," in *Proceedings of the 14th international Conference on Software Engineering and Knowledge Engineering*, SEKE '02, pp. 77–80, ACM, 2002.

[93] C. Schlenoff, S. Balakirsky, M. Uschold, R. Provine, and S. Smith, "Using Ontologies to Aid Navigation Planning in Autonomous Vehicles," *Knowledge Engineering Review*, vol. 18, pp. 243–255, September 2003.

[94] Y. Gavshin and J. Shumik, "Runtime Generation of Robot Control Code From Ontology File," in *Proceedings of the Second International Conference on Adaptive and Intelligent Systems*, ICAIS '11, pp. 157–167, Springer-Verlag, 2011.

[95] C. A. Bohn, "Heuristics for Designing the Control of a UAV Fleet With Model Checking," in *Cooperative Systems* (D. Grundel, R. Murphey, P. Pardalos, and O. Prokopyev, eds.), vol. 588 of *Lecture Notes in Economics and Mathematical Systems*, pp. 21–36, Springer Berlin Heidelberg, 2007.

[96] M. Webster, M. Fisher, N. Cameron, and M. Jump, "Formal Methods for the Certification of Autonomous Unmanned Aircraft Systems," in *Proceedings of the 30th International Conference on Computer Safety, Reliability, and Security*, SAFECOMP '11, pp. 228–242, Springer-Verlag, 2011.

[97] S. Jeyaraman, A. Tsourdos, R. Żbikowski, and B. A. White, "Formal Techniques for the Modelling and Validation of a Co-Operating UAV Team That Uses Dubins Set for Path Planning," in *Proceedings of the 2005 American Control Conference*, vol. 7, pp. 4690–4695, 2005.

[98] G. Sirigineedi, A. Tsourdos, B. A. White, and R. Żbikowski, "Kripke Modelling and Verification of Temporal Specifications of a Multiple UAV System," *Annals of Mathematics and Artificial Intelligence*, vol. 63, pp. 31–52, September 2011.

[99] M. Kuwata, "Kwalify User's Guide (for Ruby)." [online] Available at: http://www.kuwata-lab.com/kwalify/ruby/users-guide.html, 2011.

[100] "PRISM Manual | The PRISM Language / Parallel Composition." [online] Available at: http://www.prismmodelchecker.org/manual/ThePRISMLanguage/.

[101] D. Flanagan and Y. Matsumoto, *The Ruby Programming Language*. O'Reilly Media, first ed., 2008.

[102] P. Perrotta, *Metaprogramming Ruby: Program Like the Ruby Pros*. The Pragmatic Programmers, first ed., January 2010.

[103] B. Kitchenham, L. Pickard, and S. L. Pfleeger, "Case Studies for Method and Tool Evaluation," *IEEE Software*, vol. 12, pp. 52–62, July 1995.

[104] H. K. Wright, M. Kim, and D. E. Perry, "Validity Concerns in Software Engineering Research," in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, pp. 411–414, ACM, 2010.

[105] R. Feldt and A. Magazinius, "Validity Threats in Empirical Software Engineering Research—An Initial Survey," in *Proceedings of the 22nd International Conference on Software Engineering & Knowledge Engineering*, SEKE '10, pp. 374–379, Knowledge Systems Institute Graduate School, 2010.

[106] H. J. Seltman, *Experimental Design and Analysis*. Carnegie Mellon, 2013.

[107] C. Kaner and W. P. Bond, "Software Engineering Metrics: What Do They Measure and How Do We Know?," in *10th IEEE International Software Metrics Symposium*, Metrics '04, IEEE Computer Society, 2004.

[108] R. D. Banker, S. M. Datar, and D. Zweig, "Software Complexity and Maintainability," in *Proceedings of the Tenth International Conference on Information Systems*, ICIS '89, pp. 247–255, ACM, 1989.

[109] S. H. Kan, *Metrics and Models in Software Quality Engineering*. Addison-Wesley Professional, 2nd ed., 2002.

[110] B. Combemale, L. Gonnord, and V. Rusu, "A Generic Tool for Tracing Executions Back to a DSML's Operational Semantics," in *Proceedings of the 7th European Conference on Modelling Foundations and Applications*, ECMFA '11, pp. 35–51, Springer-Verlag, 2011.

[111] C. Wilson, "Network Centric Operations: Background and Oversight Issues for Congress," March 2007.

[112] J. W. Lloyd, "Practical Advantages of Declarative Programming," in *Proceedings of the 1994 Joint Conference on Declarative Programming*, GULP-PRODE '94, pp. 18–30, 1994.

[113] P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll, "Beyond Assertions: Advanced Specification and Verification With JML and ESC/Java2," in *Proceedings of the 4th International Conference on Formal Methods for Components and Objects*, FMCO '05, pp. 342–363, Springer-Verlag, 2006.

[114] A. Holmgren, "Using Annotations to Add Validity Constraints to JavaBeans Properties." [online] Available at: http://192.9.162.55/developer/technicalArticles/J2SE/constraints/annotations.html, March 2005.

[115] G. Ferreira, E. Loureiro, and E. Oliveira, "A Java Code Annotation Approach for Model Checking Software Systems," in *Proceedings of the 2007 ACM Symposium on Applied Computing*, SAC '07, pp. 1536–1537, ACM, 2007.

[116] C. Veness, "Calculate Distance, Bearing and More Between Latitude/Longitude Points." [online] Available at: http://www.movable-type.co.uk/scripts/latlong.html. Movable Type Ltd.

Appendix A

Threat Area Calculations

Threat area incursions have been considered throughout this thesis. The CVC uses hard-coded geodesic equations to establish the occurrence, and calculate the duration, of threat area incursions committed by UAVs (as described in Chapter 3). The equations employed by the CVC are underpinned by a spherical Earth with a two-dimensional surface. This rudimentary *mission environment* enables our prototype to avoid complexities associated with three-dimensional topographies. (Because the shape of the Earth is approximately ellipsoidal, geodetic calculations that assume a spherical geometry result in errors ranging between 0.3% and 0.55%.)

In this context, a direct flightpath between two waypoints can be delineated by the minor arc of a great circle. A waypoint is a zero-dimensional spherical point designated by latitude and longitude; a great circle is the intersection of a sphere and a plane passing through the center of that sphere. The minor arc of a great circle, which we will refer to as a *great circle arc*, is the shortest path between two points on the surface of a sphere.

Since great circle arcs can also be used to delineate threat area boundaries, an asset enters and exits a threat area at the intersection of two great circle arcs. We therefore use great circle intersections to establish the occurrence of threat area incursions.

A.1 Establishing Threat Area Incursions

Following from the definition of a great circle, a great circle arc lies on a plane that passes through the center of a sphere. Two great circles intersect when their respective planes intersect (the only other possibility being that the planes overlap), thereby forming a straight line that crosses the surface of the underlying sphere at two points. Consequently, the intersection of two great circle arcs must occur at one of the spherical points resulting from the intersection of the great circles underpinning those arcs.

We consider great circle arcs a and b defined by points a_1, a_2 and b_1, b_2, respectively. The latitude and longitude coordinates that designate these points are converted to Cartesian coordinates using Equation A.1, where R is the Earth's mean radius (6371 kilometers).

Equation A.2 calculates vectors v_a and v_b from points a_1, a_2 and b_1, b_2, respectively, where each vector defines a plane containing the points used to calculate that vector.

$$p = \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} R \cdot \cos lat \cdot \cos lon \\ R \cdot \cos lat \cdot \sin lon \\ R \cdot \sin lat \end{bmatrix} \tag{A.1}$$

Equation A.3 calculates unit vectors u_a and u_b from v_a and v_b, respectively. If unit vectors u_a and u_b are not identical (as determined by Equation A.4) vectors v_a and v_b define planes that do not overlap.

$$v = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} = \begin{bmatrix} y_1 \cdot z_2 - y_2 \cdot z_1 \\ x_2 \cdot z_1 - x_1 \cdot z_2 \\ x_1 \cdot y_2 - x_2 \cdot y_1 \end{bmatrix} \tag{A.2}$$

$$u = \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix} = \begin{bmatrix} v_x / \sqrt{{v_x}^2 + {v_y}^2 + {v_z}^2} \\ v_y / \sqrt{{v_x}^2 + {v_y}^2 + {v_z}^2} \\ v_z / \sqrt{{v_x}^2 + {v_y}^2 + {v_z}^2} \end{bmatrix} \tag{A.3}$$

$$u_1 \cap u_2 = \begin{cases} |u_{1x} - u_{2x}| < \varepsilon \\ |u_{1y} - u_{2y}| < \varepsilon \\ |u_{1z} - u_{2z}| < \varepsilon \end{cases} \tag{A.4}$$

Two non-overlapping planes intersect in a straight line defined by direction vector d. Equation A.5 calculates d from vectors u_a and u_b. Equation A.6 calculates the unit vector of d, which defines the coordinates of spherical point p_1. Equation A.7 calculates the inverse of the unit vector of d, which defines the coordinates of spherical point p_2. The latitude and longitude coordinates of p_1 and p_2 are calculated from their Cartesian coordinates using Equation A.8 and Equation A.9, respectively. Since all values passed to trigonometric functions are expressed in radians, output from Equation A.8 and Equation A.9 must be converted from radians to degrees before proceeding to the next set of calculations.

Having identified points p_1 and p_2, we check if either point is located on both arcs a and b. For arc a (defined by points a_1 and a_2) and point p_1, we use the Haversine formula, which is described in Section A.2.1, to calculate the distance from a_1 to a_2, $d_{(a_1,a_2)}$; and the distances from p_1 to a_1 and a_2, $d_{(p_1,a_1)}$ and $d_{(p_1,a_2)}$, respectively. Consequently, $p_1 \in a \Leftrightarrow d_{(a_1,a_2)} = d_{(p_1,a_1)} + d_{(p_1,a_2)}$. Similarly, we check p_1 with respect to arc b. If p_1 is not an intersection, we check p_2. If p_1 and p_2 are not intersection points, then arcs a and b do not intersect.

A.2 Calculating Threat Area Durations

Having established the occurrence of a threat area incursion, we proceed to calculate its duration. The duration of travel between two points is a function of distance and speed.

$$d = \begin{bmatrix} d_x \\ d_y \\ d_z \end{bmatrix} = \begin{bmatrix} u_{1y} \cdot u_{2z} - u_{2y} \cdot u_{1z} \\ u_{2x} \cdot u_{1z} - u_{1x} \cdot u_{2z} \\ u_{1x} \cdot u_{2y} - u_{2x} \cdot u_{1y} \end{bmatrix} \tag{A.5}$$

$$p_1 = \begin{bmatrix} p_{1x} \\ p_{1y} \\ p_{1z} \end{bmatrix} = \begin{bmatrix} d_x / \sqrt{{d_x}^2 + {d_y}^2 + {d_z}^2} \\ d_y / \sqrt{{d_x}^2 + {d_y}^2 + {d_z}^2} \\ d_z / \sqrt{{d_x}^2 + {d_y}^2 + {d_z}^2} \end{bmatrix} \tag{A.6}$$

$$p_2 = \begin{bmatrix} p_{2x} \\ p_{2y} \\ p_{2z} \end{bmatrix} = \begin{bmatrix} -d_x / \sqrt{{d_x}^2 + {d_y}^2 + {d_z}^2} \\ -d_y / \sqrt{{d_x}^2 + {d_y}^2 + {d_z}^2} \\ -d_z / \sqrt{{d_x}^2 + {d_y}^2 + {d_z}^2} \end{bmatrix} \tag{A.7}$$

$$lat = \arcsin\left(\frac{z}{R}\right) \tag{A.8}$$

$$lon = \arctan(y, x) \tag{A.9}$$

A.2.1 Distance

A great circle distance is the shortest distance between two points along a path on the surface of a sphere. The great circle distance d between points p_1 and p_2 with coordinates lat_1, lon_1 and lat_2, lon_2, respectively, can be calculated using the Haversine formula in Equation A.10, where R is the Earth's mean radius [116]. (All values passed to trigonometric functions are assumed radians.)

$$\begin{aligned} a &= \sin^2 \frac{lat_2 - lat_1}{2} + \sin^2 \frac{lon_2 - lon_1}{2} \cdot \cos lat_1 \cdot \cos lat_2 \\ c &= 2 \cdot \arctan(\sqrt{a}, \sqrt{1-a}) \\ d &= R \cdot c \end{aligned} \tag{A.10}$$

A.2.2 Speed

The velocity of a UAV at any given moment during flight is a function of four parameters including the velocity of the UAV in still air; the UAV's direction of travel; and the velocity and direction of the wind. Consider a UAV with velocity vector v_a, which has magnitude $|v_a|$ and direction θ_a. The UAV flies in a wind with velocity vector v_w, which has magnitude $|v_w|$ and direction θ_w. Let i and j be $1ms^{-1}$ (meters per second) East and $1ms^{-1}$ North, respectively. The component form for each velocity vector can be calculated using Equation A.11 and Equation A.12; velocity can be calculated using Equation A.13; and the speed of the UAV in ms^{-1} can be calculated using Equation A.14.

$$v_a = |v_a| \cos\theta_a i + |v_a| \sin\theta_a j \tag{A.11}$$
$$v_w = |v_w| \cos\theta_w i + |v_w| \sin\theta_w j \tag{A.12}$$

$$v = v_a + v_w \tag{A.13}$$

$$s = |v| \approx \sqrt{{v_a}^2 + {v_w}^2} \tag{A.14}$$

A.2.3 Bearing

The direction of travel along a great circle arc is defined by initial and final bearings. The initial bearing θ from point p_1 to point p_2 is calculated using Equation A.15. Because the range of arctan is the interval $[-180°, 180°]$, the initial bearing is normalized to a compass bearing θ_{in} using Equation A.16. The range of θ_{in} is the interval $[0, 360°]$. The normalized final bearing θ_{fn} is calculated by reversing the initial bearing from p_2 to p_1 using Equation A.17.

$$\begin{aligned} y &= \sin(lon_2 - lon_1) \cdot \cos lat_2 \\ x &= \cos lat_1 \cdot \sin lat_2 - \sin lat_1 \cdot \cos lat_2 \cdot \cos(lon_2 - lon_1) \\ \theta &= \arctan(y, x) \end{aligned} \tag{A.15}$$

$$\theta_{in} = (\theta_{(p_1,p_2)} \cdot \frac{360}{2\pi} + 360) \bmod 360 \tag{A.16}$$

$$\theta_{fn} = (\theta_{(p_2,p_1)} \cdot \frac{360}{2\pi} + 180) \bmod 360 \tag{A.17}$$

Each bearing is applied to a specific route segment. The initial bearing defines the direction of travel from p_1 to midpoint p_m between p_1 and p_2. The final bearing defines the direction of travel from p_m to p_2. The latitude and longitude coordinates for p_m are calculated using Equation A.18 and Equation A.19, respectively.

$$\begin{aligned} Bx &= \cos lat_2 \cdot \cos(lon_2 - lon_1) \\ By &= \cos lat_2 \cdot \sin(lon_2 - lon_1) \\ lat_m &= \arctan(\sin lat_1 + \sin lat_2, \sqrt{(\cos lat_1 + Bx)^2 + By^2}) \end{aligned} \tag{A.18}$$
$$lon_m = lon_1 + \arctan(By, \cos lat_1 + Bx) \tag{A.19}$$

A.2.4 Duration

The duration of travel along a great circle arc defined by initial and final bearings requires calculations for two speeds and two distances. Equation A.20 generalizes this requirement

by calculating the duration of travel along a path delineated by n waypoints, where $d_{(p_i,p_{i+1})}$ is the distance in kilometers between sequential points p_i and p_{i+1}; and $s_{(p_i,p_{i+1})}$ is the speed of travel in ms^{-1} from p_i to p_{i+1}. Because Equation A.10 calculates distance in kilometers and Equation A.14 calculates speed in meters per second, Equation A.20 converts distances to meters in order to calculate duration in seconds.

$$\text{duration} := \sum_{i=1}^{n} \frac{d_{(p_i,p_{i+1})}}{s_{(p_i,p_{i+1})}} \cdot 1000 \tag{A.20}$$

Appendix B

Ontology

Listing B.1: OWL code for CEMO.

```
Ontology: complex_missions

Datatype: xsd:integer

Class: owl:Nothing

Class: Action

Class: ARDrone
  SubClassOf: NamedAsset,
    hasCostValue some xsd:integer[<= 500],
    hasEnduranceInSeconds some xsd:integer[<= 70],
    hasSpeedInKilometersPerHour some xsd:integer[<= 18]
  DisjointWith: Hummingbird

Class: Area
  SubClassOf: hasWaypoint some Waypoint

Class: Asset
  SubClassOf: hasAction some KineticAction,
    hasCostValue some xsd:integer,
    hasEnduranceInSeconds some xsd:integer,
    hasSpeedInKilometersPerHour some xsd:integer

Class: HoverAction
  SubClassOf: KineticAction,
    hasWaypoint some Waypoint
  DisjointWith: TraversePathSegmentAction

Class: Hummingbird
  SubClassOf: NamedAsset,
    hasCostValue some xsd:integer[>= 5000],
    hasEnduranceInSeconds some xsd:integer[<= 120],
    hasSpeedInKilometersPerHour some xsd:integer[<= 50]
  DisjointWith: ARDrone

Class: KineticAction
  SubClassOf: Action,
    hasDurationInSeconds some xsd:integer,
    hasPrecondition only Action
  DisjointWith: SensorAction

Class: Mission
  SubClassOf: hasAsset some Asset

Class: NamedAsset
  SubClassOf: Asset

Class: PhotoSurveillanceAction
  SubClassOf: SensorAction,
    hasDurationInSeconds some xsd:integer,
    hasPrecondition only Action

Class: SensorAction
```

```
  SubClassOf: Action
  DisjointWith: KineticAction

Class: TraversePathSegmentAction
  SubClassOf: KineticAction,
    hasStartPoint some Waypoint,
    hasEndpoint some Waypoint
  DisjointWith: HoverAction

Class: Waypoint

DisjointClasses: Action, Area, Asset, Mission, Waypoint

DataProperty: hasCostValue
  Characteristics: Functional
  Domain: Asset
  Range: xsd:integer

DataProperty: hasDurationInSeconds
  Characteristics: Functional
  Domain: Action
  Range: xsd:integer

DataProperty: hasEnduranceInSeconds
  Characteristics: Functional
  Domain: Asset
  Range: xsd:integer

DataProperty: hasSpeedInKilometersPerHour
  Characteristics: Functional
  Domain: Asset
  Range: xsd:integer

ObjectProperty: hasAction
  Characteristics: Asymmetric,
    Irreflexive,
    InverseFunctional
  Domain: Asset
  Range: Action
  InverseOf: isActionOf

ObjectProperty: hasAsset
  Characteristics: Asymmetric,
    Irreflexive,
    InverseFunctional
  Domain: Mission
  Range: Asset

ObjectProperty: hasEndpoint
  SubPropertyOf: hasWaypoint
  Characteristics: Asymmetric,
    Irreflexive,
    Functional
  Domain: TraversePathSegmentAction
  Range: Waypoint

ObjectProperty: hasPrecondition
  Characteristics: Transitive
  Domain: Action
  Range: Action
  InverseOf: isPreconditionTo

ObjectProperty: hasSibling
  Characteristics: Asymmetric,
    Irreflexive
  Domain: SensorAction
  Range: KineticAction

ObjectProperty: hasStartPoint
  SubPropertyOf: hasWaypoint
  Characteristics: Asymmetric,
    Irreflexive,
    Functional
```

```
  Domain: TraversePathSegmentAction
  Range: Waypoint

ObjectProperty: hasWaypoint
  Characteristics: Asymmetric,
    Irreflexive
  Range: Waypoint
  InverseOf: isWaypointOf

ObjectProperty: isActionOf
  Characteristics: Asymmetric,
    Irreflexive,
    Functional
  Domain: Action
  Range: Asset
  InverseOf: hasAction

ObjectProperty: isPreconditionTo
  Characteristics: Transitive
  Domain: Action
  Range: Action
  InverseOf: hasPrecondition

ObjectProperty: isWaypointOf
  Characteristics: Asymmetric,
    Irreflexive
  Domain: Waypoint
  InverseOf: hasWaypoint

Rule:
    KineticAction(?x),
    KineticAction(?y),
    SensorAction(?z),
    hasAction(?a, ?x),
    hasAction(?a, ?y),
    hasAction(?a, ?z),
    hasPrecondition(?x, ?y),
    hasPrecondition(?z, ?y)
  -> hasSibling(?z, ?x)

Rule:
    KineticAction(?x),
    KineticAction(?y),
    SensorAction(?z),
    hasAction(?a, ?x),
    hasAction(?a, ?y),
    hasAction(?a, ?z),
    hasPrecondition(?x, ?y),
    hasSibling(?z, ?y)
  -> hasSibling(?z, ?x)

Rule:
    SensorAction(?y),
    hasAction(?a, ?x),
    hasAction(?b, ?y),
    hasPrecondition(?x, ?y),
    DifferentFrom (?a, ?b)
  -> owl:Nothing(?y)
```

Listing B.2: OWL code for Tactical-CEMO.

```
Ontology: complex_tactical_missions

Import: complex_missions
  Class: Action
  Class: Area
  Class: Asset
  Class: HoverAction
  Class: KineticAction
  Class: Mission
  Class: SensorAction
  Class: TraversePathSegmentAction
```

```
  Class: Waypoint
  DataProperty: hasCostValue
  DataProperty: hasSpeedInKilometersPerHour
  ObjectProperty: hasAction
  ObjectProperty: hasEndpoint
  ObjectProperty: hasSibling
  ObjectProperty: hasWaypoint
  ObjectProperty: isWaypointOf

Datatype: xsd:double
Datatype: xsd:integer

Class: DirectThreatAreaAction
  EquivalentTo: ThreatAreaAction
    and (DirectThreatAreaHoverAction or DirectThreatAreaTPSA)

Class: DirectThreatAreaHoverAction
  EquivalentTo: ThreatAreaAction
    and (HoverAction
      and (hasWaypoint some ThreatAreaWaypoint))
  DisjointWith: DirectThreatAreaTPSA

Class: DirectThreatAreaTPSA
  EquivalentTo: ThreatAreaAction
    and (TraversePathSegmentAction
      and (hasEndpoint some ThreatAreaWaypoint))
  DisjointWith: DirectThreatAreaHoverAction

Class: DomainConcept

Class: HighVulnerabilityAsset
  EquivalentTo: Asset
    and (hasCostValue some xsd:integer[<= 1000])
    and (hasSpeedInKilometersPerHour some xsd:integer[<= 20])
  SubClassOf: hasRiskAcceptabilityFactor value HighRiskAcceptabilityFactor,
    hasVulnerability value HighVulnerability
  DisjointWith: LowVulnerabilityAsset

Class: LowVulnerabilityAsset
  EquivalentTo: Asset
    and (hasCostValue some xsd:integer[>= 3000])
    and (hasSpeedInKilometersPerHour some xsd:integer[<= 60])
  SubClassOf: hasRiskAcceptabilityFactor value LowRiskAcceptabilityFactor
    hasVulnerability value LowVulnerability,
  DisjointWith: HighVulnerabilityAsset

Class: RiskAcceptabilityFactor
  EquivalentTo: DomainConcept
    and ({HighRiskAcceptabilityFactor, LowRiskAcceptabilityFactor})
  DisjointWith: Vulnerability

Class: ThreatArea
  EquivalentTo: Area
    and (hasWaypoint some ThreatAreaWaypoint)

Class: ThreatAreaAction
  EquivalentTo: KineticAction
    and (hasWaypoint some ThreatAreaWaypoint)

Class: ThreatAreaWaypoint
  EquivalentTo: Waypoint
    and (isWaypointOf some ThreatArea)

Class: ThreatenedAsset
  EquivalentTo: Asset
    and (hasAction some DirectThreatAreaAction)

Class: ValidAsset
  EquivalentTo: ThreatenedAsset
    and (hasAction some (SensorAction
      and (hasSibling some DirectThreatAreaAction)))

Class: Vulnerability
```

```
  EquivalentTo: DomainConcept
    and ({HighVulnerability, LowVulnerability})
  DisjointWith: RiskAcceptabilityFactor

DisjointClasses: Action, Area, Asset, DomainConcept, Mission, Waypoint

DataProperty: hasDoubleValue
  Characteristics: Functional
  Domain: DomainConcept
  Range: xsd:double

Individual: HighRiskAcceptabilityFactor
  Types: RiskAcceptabilityFactor
  Facts: hasDoubleValue 0.8

Individual: HighVulnerability
  Types: Vulnerability
  Facts: hasDoubleValue 0.1

Individual: LowRiskAcceptabilityFactor
  Types: RiskAcceptabilityFactor
  Facts: hasDoubleValue 0.6

Individual: LowVulnerability
  Types: Vulnerability
  Facts: hasDoubleValue 0.01

ObjectProperty: hasRiskAcceptabilityFactor
  Characteristics: Asymmetric,
    Irreflexive,
    Functional
  Domain: Asset
  Range: RiskAcceptabilityFactor

ObjectProperty: hasVulnerability
  Characteristics: Asymmetric,
    Irreflexive,
    Functional
  Domain: Asset
  Range: Vulnerability

ObjectProperty: invades
  Characteristics: Asymmetric,
    Irreflexive,
    Functional
  Domain: DirectThreatAreaTPSA
  Range: ThreatArea

Rule:
    ThreatArea(?a),
    hasEndpoint(?x, ?w),
    isWaypointOf(?w, ?a)
  -> invades(?x, ?a)
```

Listing B.3: OWL code for Traffic-CEMO.

```
Ontology: complex_traffic_surveillance_missions

Import: complex_missions
  Class: owl:Nothing
  Class: Action
  Class: Area
  Class: Asset
  Class: HoverAction
  Class: Mission
  Class: SensorAction
  Class: Waypoint
  ObjectProperty: hasAction
  ObjectProperty: hasWaypoint

Datatype: xsd:double
Datatype: xsd:integer
```

```
Class: DomainConcept

Class: FreewaySection
  SubClassOf: Road,
    approachesMinimumSpeed some xsd:double,
    exceedsMinimumSpeed some xsd:double,
    exceedsNominalSpeed some xsd:double,
    hasLaneClassification some LaneClassification,
    hasRampFrequency some RampFrequency

Class: HighRampFrequency
  SubClassOf: RampFrequency
  DisjointWith: LowRampFrequency

Class: HighSpeedFreewaySection
  EquivalentTo: FreewaySection
    and (hasLaneClassification some ThreeLaneClassification)
    and (hasRampFrequency some LowRampFrequency)
  SubClassOf: approachesMinimumSpeed some xsd:double[>= 0.2],
    exceedsMinimumSpeed some xsd:double[>= 0.9],
    exceedsNominalSpeed some xsd:double[>= 0.5]
  DisjointWith: LowSpeedFreewaySection

Class: HighSpeedLidarAction
  EquivalentTo: LidarAction
    and (monitors some HighSpeedFreewaySection)
  SubClassOf: hasIntervalCalibrationFactor value HighIntervalCalibrationFactor
  DisjointWith: LowSpeedLidarAction

Class: IntervalCalibrationFactor
  EquivalentTo: DomainConcept
    and ({HighIntervalCalibrationFactor, LowIntervalCalibrationFactor})

Class: LaneClassification
  EquivalentTo: ThreeLaneClassification
    or TwoLaneClassification
  SubClassOf: ValuePartition
  DisjointWith: RampFrequency

Class: LidarAction
  SubClassOf: SensorAction,
    hasIntervalCalibrationFactor some IntervalCalibrationFactor,
    hasIntervalInSeconds some xsd:integer,
    isConcurrentWith some HoverAction
  DisjointWith: PhotoSurveillanceAction

Class: LowRampFrequency
  SubClassOf: RampFrequency
  DisjointWith: HighRampFrequency

Class: LowSpeedFreewaySection
  EquivalentTo: FreewaySection
    and (hasLaneClassification some TwoLaneClassification)
    and (hasRampFrequency some HighRampFrequency)
  SubClassOf: approachesMinimumSpeed some xsd:double[>= 0.3],
    exceedsMinimumSpeed some xsd:double[>= 0.7],
    exceedsNominalSpeed some xsd:double[>= 0.4]
  DisjointWith: HighSpeedFreewaySection

Class: LowSpeedLidarAction
  EquivalentTo: LidarAction
    and (monitors some LowSpeedFreewaySection)
  SubClassOf: hasIntervalCalibrationFactor value LowIntervalCalibrationFactor
  DisjointWith: HighSpeedLidarAction

Class: PhotoSurveillanceAction
  DisjointWith: LidarAction

Class: RampFrequency
  EquivalentTo: HighRampFrequency
    or LowRampFrequency
  SubClassOf: ValuePartition
```

```
  DisjointWith: LaneClassification

Class: Road
  SubClassOf: hasWaypoint some Waypoint

Class: ThreeLaneClassification
  SubClassOf: LaneClassification
  DisjointWith: TwoLaneClassification

Class: TwoLaneClassification
  SubClassOf: LaneClassification
  DisjointWith: ThreeLaneClassification

Class: ValuePartition

DisjointClasses: Action, Area, Asset, DomainConcept, Mission, Road,
    ValuePartition, Waypoint

DataProperty: approachesMinimumSpeed
  Characteristics: Functional
  Domain: FreewaySection
  Range: xsd:double

DataProperty: exceedsMinimumSpeed
  Characteristics: Functional
  Domain: FreewaySection
  Range: xsd:double

DataProperty: exceedsNominalSpeed
  Characteristics: Functional
  Domain: FreewaySection
  Range: xsd:double

DataProperty: hasDoubleValue
  Characteristics: Functional
  Domain: DomainConcept
  Range: xsd:double

DataProperty: hasIntegerValue
  Characteristics: Functional
  Domain: DomainConcept
  Range: xsd:integer

DataProperty: hasIntervalInSeconds
  Characteristics: Functional
  Domain: LidarAction
  Range: xsd:integer

Individual: HighIntervalCalibrationFactor
  Types: IntervalCalibrationFactor
  Facts: hasIntegerValue 3

Individual: LowIntervalCalibrationFactor
  Types: IntervalCalibrationFactor
  Facts: hasIntegerValue 2

ObjectProperty: hasIntervalCalibrationFactor
  Characteristics: Asymmetric,
    Irreflexive,
    Functional
  Domain: LidarAction
  Range: IntervalCalibrationFactor

ObjectProperty: hasLaneClassification
  Characteristics: Asymmetric,
    Irreflexive,
    Functional
  Domain: FreewaySection
  Range: LaneClassification

ObjectProperty: hasRampFrequency
  Characteristics: Asymmetric,
    Irreflexive,
```

```
    Functional
  Domain: FreewaySection
  Range: RampFrequency

ObjectProperty: isConcurrentWith
  Characteristics: Asymmetric,
    Irreflexive
  Domain: LidarAction
  Range: HoverAction

ObjectProperty: monitors
  Characteristics: Asymmetric,
    Irreflexive
  Domain: LidarAction
  Range: FreewaySection

Rule:
    hasAction(?a, ?x),
    isConcurrentWith(?y, ?x)
  -> hasAction(?a, ?y)

Rule:
    HoverAction(?h),
    FreewaySection(?f),
    LidarAction(?l),
    hasWaypoint(?f, ?w),
    hasWaypoint(?h, ?w),
    isConcurrentWith(?l, ?h)
  -> monitors(?l, ?f)

Rule:
    hasAction(?a, ?x),
    hasAction(?b, ?y),
    isConcurrentWith(?x, ?y),
    DifferentFrom (?a, ?b)
  -> owl:Nothing(?x)
```

Listing B.4: OWL code for Mission A.

```
Ontology: Mission_A

Import: complex_tactical_missions
  Class: Hummingbird
  Class: PhotoSurveillanceAction
  Class: ThreatArea
  Class: TraversePathSegmentAction
  Class: Waypoint
  ObjectProperty: hasAction
  ObjectProperty: hasEndpoint
  ObjectProperty: hasPrecondition
  ObjectProperty: hasStartPoint
  ObjectProperty: hasWaypoint

Individual: Hummingbird1
  Types: Hummingbird
  Facts: hasAction TraversePathSegmentAction1,
    hasAction TraversePathSegmentAction2

Individual: Hummingbird2
  Types: Hummingbird
  Facts: hasAction TraversePathSegmentAction3,
    hasAction TraversePathSegmentAction4,
    hasAction PhotoSurveillanceAction5

Individual: PhotoSurveillanceAction5
  Types: PhotoSurveillanceAction
  Facts: hasPrecondition TraversePathSegmentAction3

Individual: ThreatArea1
  Types: ThreatArea
  Facts: hasWaypoint Waypoint6
```

```
Individual: TraversePathSegmentAction1
  Types: TraversePathSegmentAction
  Facts: hasStartPoint Waypoint1,
    hasEndpoint Waypoint2

Individual: TraversePathSegmentAction2
  Types: TraversePathSegmentAction
  Facts: hasPrecondition TraversePathSegmentAction1,
    hasPrecondition TraversePathSegmentAction3,
    hasStartPoint Waypoint2,
    hasEndpoint Waypoint3

Individual: TraversePathSegmentAction3
  Types: TraversePathSegmentAction
  Facts: hasStartPoint Waypoint4,
    hasEndpoint Waypoint5

Individual: TraversePathSegmentAction4
  Types: TraversePathSegmentAction
  Facts: hasPrecondition TraversePathSegmentAction3,
    hasStartPoint Waypoint5,
    hasEndpoint Waypoint6

Individual: Waypoint1
  Types: Waypoint

Individual: Waypoint2
  Types: Waypoint

Individual: Waypoint3
  Types: Waypoint

Individual: Waypoint4
  Types: Waypoint

Individual: Waypoint5
  Types: Waypoint

Individual: Waypoint6
  Types: Waypoint
```

Appendix C

Prolog Knowledge Base

Listing C.1: Prolog code for asset rules.

```
/* OWL defined class */
single_action_asset(A) :-
    bagof(A, X^(has_action(A, X)), C),
    length(C, 1).

/* OWL defined class */
zero_action_asset(A) :-
    not(has_action(A, _)).

/* OWL defined class */
observer_asset(A) :-
    has_action(A, X),
    observer(X).

/* OWL defined class */
observed_asset(A) :-
    has_action(A, X),
    subject(X).

/* SWRL rule */
observes(A, B) :-
    crosscutting_precondition(A, _, B, _).

primary_asset(A) :-
    observer_asset(A),
    (
        not(observed_asset(A));
        (
            observes(A, B),
            observes(B, A)
        )
    ).
```

Listing C.2: Prolog code for utility rules.

```
:- dynamic asset/1.
:- dynamic kinetic_action/1.
:- dynamic has_action/2.
:- dynamic has_precondition/2.

precondition_util(A, X, B, Y) :-
    has_action(A, X),
    has_action(B, Y),
    has_precondition(Y, X).

crosscutting_precondition(A, X, B, Y) :-
    precondition_util(A, X, B, Y),
    not(A = B).

sibling_precondition(A, X, B, Y) :-
    precondition_util(A, X, B, Y),
    A = B.
```

```
/* OWL property */
is_precondition_to(X, Y) :-
    has_action(A, X),
    has_action(A, Y),
    has_precondition(Y, X).

/* OWL defined class */
singleton(X) :-
    has_action(A, X),
    single_action_asset(A).

terminal(X) :-
    has_action(A, X),
    not(is_precondition_to(X, _)),
    not(single_action_asset(A)),
    not(zero_action_asset(A)).
```

Listing C.3: Prolog code for default rules.

```
default_util(X) :-
    not(constrained_observer(X)),
    not(observer(X)),
    not(observer_precondition(X)),
    not(subject(X)),
    not(subject_constraint(X)),
    not(subject_precondition(X)).

default(X) :-
    default_util(X),
    not(singleton(X)),
    not(terminal(X)).

default_singleton(X) :-
    default_util(X),
    singleton(X).

default_terminal(X) :-
    default_util(X),
    terminal(X).
```

Listing C.4: Prolog code for observer and subject rules.

```
/* OWL defined class */
observer_and_subject(X) :-
    observer(X),
    subject(X).

/* OWL defined class */
observer_and_constrained_subject(X) :-
    observer(X),
    constrained_subject(X).

/* OWL defined class */
observer_and_singleton_subject(X) :-
    observer(X),
    singleton_subject(X).

observer_and_terminal_subject(X) :-
    observer(X),
    terminal_subject(X).
```

Listing C.5: Prolog code for observer rules.

```
/* SWRL rule */
observer(Y) :-
    crosscutting_precondition(_, _, _, Y).

default_observer(X) :-
    observer(X),
    not(observer_precondition(X)).
```

```
/* SWRL rule */
constrained_observer(Y) :-
    constrained_subject(X),
    crosscutting_precondition(_, X, _, Y).

/* SWRL rule */
sibling_observer(Y) :-
    sibling_precondition(_, _, _, Y).

terminal_observer(X) :-
    observer(X),
    terminal(X).

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

/* SWRL rule */
observer_precondition(X) :-
    sibling_precondition(_, X, _, Y),
    (
        observer(Y);
        observer_precondition(Y)
    ).
```

Listing C.6: Prolog code for subject rules.

```
/* SWRL rule */
subject(X) :-
    crosscutting_precondition(_, X, _, _).

/* OWL defined class */
constrained_subject(X) :-
    subject(X),
    is_precondition_to(X, _).

leading_subject(X) :-
    crosscutting_precondition(_, X, _, Y),
    not(sibling_observer(Y)).

/* OWL defined class */
singleton_subject(X) :-
    subject(X),
    singleton(X).

terminal_subject(X) :-
    subject(X),
    terminal(X).

/* SWRL rule */
subject_constraint(Y) :-
    sibling_precondition(_, X, _, Y),
    (
        constrained_subject(X);
        subject_constraint(X)
    ).

/* SWRL rule */
subject_precondition(X) :-
    sibling_precondition(_, X, _, Y),
    (
        subject(Y);
        subject_precondition(Y)
    ).

terminal_subject_constraint(X) :-
    subject_constraint(X),
    terminal(X).
```

Listing C.7: Prolog code for existential quantification rules.

```
:- include(asset_rules).
:- include(utility_rules).

:- dynamic has_asset/2.
:- dynamic has_duration_in_seconds/2.
:- dynamic has_endpoint/2.
:- dynamic has_endurance_in_seconds/2.
:- dynamic has_interval_in_seconds/2.
:- dynamic has_start_point/2.
:- dynamic has_waypoint/2.
:- dynamic is_concurrent_with/2.
:- dynamic lidar_action/1.
:- dynamic mission/1.
:- dynamic photo_surveillance_action/1.

invalid_kinetic_action(K) :-
    not(has_duration_in_seconds(K, _)).

invalid_hover_action(H) :-
    invalid_kinetic_action(H);
    not(has_waypoint(H, _)).

invalid_traverse_path_segmentAction(T) :-
    invalid_kinetic_action(T);
    not(has_endpoint(T, _));
    not(has_start_point(T, _)).

invalid_photo_surveillance_action(P) :-
    not(has_duration_in_seconds(P, _)).

invalid_asset(A) :-
    zero_action_asset(A);
    not(has_endurance_in_seconds(A, _)).

invalid_mission(M) :-
    not(has_asset(M, _)).

invalid_lidar_action(L) :-
    not(has_interval_in_seconds(L, _));
    not(is_concurrent_with(L, _)).
```

Appendix D

PRISM Templates

Listing D.1: Ruby code for the DTMC asset module template.

```
require './template_util'

class Action; attr_accessor :asset end

class Asset
  alias :original_add :add_action

  def add_action(action)
    original_add(action)
    action.asset = self
  end

  def last
    @kinetic_actions.each { |action| return action if action.is_terminal? }
  end
end

class KineticAction < Action
  attr_reader :observers

  def is_terminal?
    @type.include?(:singleton) || @type.include?(:terminal)
  end
end

module AssetTemplate
  @primary_asset = lambda { |action|
    @asset ||=
      if action.asset.type == :primary_asset
        @asset = action.asset
      else
        action.asset.kinetic_actions.each { |kinetic_action|
          kinetic_action.observers.each { |observer|
            return @primary_asset[observer] if observer.is_kinetic_action?
          }
        }
      end
  }

  def self.generate(asset)
    commands = String.new
    asset.kinetic_actions.each { |kinetic_action|
      action = "[#{action_name␣kinetic_action}]"
      guard = "e#{asset.id}>0␣&␣d#{kinetic_action.id}>0"
      update = "(e#{asset.id}'=e#{asset.id}-1)"
      commands += "#{action}␣#{guard}␣->␣#{update};"
      commands += "\n#{"\s"*2}" unless kinetic_action == asset.kinetic_actions.
          last
    }
    puts <<-end.gsub(/ {6}/, '')
      module #{asset.class}#{asset.id}
        e#{asset.id} : [0..#{asset.class.endurance}] init #{asset.class.endurance
            };
        #{commands}
```

```
        [#{last_action_name asset.last}] e#{asset.id}=0 | d#{asset.last.id}=0 ->
            true;
      endmodule
    end
  end

  private
    def self.action_name(action)
      case action.type
        when :default,
             :default_singleton,
             :default_terminal,
             :leading_subject,
             :default_observer,
             :terminal_observer
          "actn#{action.id}"
        when :terminal_constrained_observer
          "asst#{action.asset.id}"
        else
          "asst#{@primary_asset[action].id}"
      end
    end

    def self.last_action_name(action)
      case action.type
        when :terminal_constrained_observer
          "asst#{action.asset.id}"
        when :singleton_subject,
             :terminal_subject,
             :observer_and_singleton_subject,
             :observer_and_terminal_subject,
             :terminal_subject_constraint
          "asst#{@primary_asset[action].id}"
      end
    end
end

execute { |operation|
  puts "asset␣modules␣for␣#{operation}..."
  ClassLoader.new(operation).assets.each { |asset|
    AssetTemplate.generate asset
  }
}
```

Listing D.2: Ruby code for the DTMC survivability template.

```
require './template_util'

class Asset; attr_accessor :raf, :vulnerability end
class KineticAction < Action; attr_accessor :incursion end

class SensorAction < Action
  attr_reader :id
  defensive_copy :siblings

  def siblings=(siblings)
    @siblings = siblings.collect { |item| item.to_s }
  end
end

class ThreatLoader < ClassLoader
  def initialize(operation)
    super(operation)
    load(:actions, :incursion, :siblings)
    load(:assets, :raf, :vulnerability)
  end
end

module SurvivabilityTemplate
  @command = lambda { |params, action|
    newline = "\n#{"\s"*2}"
    temp = "#{params[:action]}␣#{params[:guard1]}␣->␣#{params[:update]};#{newline
```

```
      }"
  temp += "#{params[:action]}␣#{params[:guard2]}␣->␣true;"
  temp += newline unless action == @asset.kinetic_actions.last
  temp
}

def self.generate(asset)
  unless asset.vulnerability == nil
    @asset = asset
    generate_formulas
    generate_kinetic_action_modules(&@command)
    generate_sensor_action_modules(&@command) unless asset.sensor_actions.empty
        ?
    generate_properties
  end
end

private
  def self.generate_formulas
    durations = String.new
    @asset.kinetic_actions.each { |action|
      unless action.incursion == nil
        action.incursion.each_pair { |key, value| puts "const␣int␣#{key}#{
            action.id}␣=␣#{value};\n" }
        start = "start#{action.id}"
        finish = "finish#{action.id}"
        duration = "duration#{action.id}"
        puts <<-end.gsub(/ {2}/, '')
          formula actn#{action.id}_tai = d#{action.id}>#{finish} & d#{action.id
              }<=#{start};
          formula #{duration} = #{start} - #{finish};
        end
        durations += durations.empty? ? duration : "␣+␣#{duration}"
      end
    }
    sad = @asset.sensor_actions.empty? ? 0 : "sad#{@asset.id}"
    puts <<-end.gsub(/ {2}/, '')
      formula tkad#{@asset.id} = #{durations};
      formula raf#{@asset.id} = #{sad} / tkad#{@asset.id};
    end
  end

  def self.generate_kinetic_action_modules
    commands = String.new
    @asset.kinetic_actions.each { |action|
      unless action.incursion == nil
        params = {
          action: "[actn#{action.id}]",
          guard1: "!a#{@asset.id}d␣&␣␣actn#{action.id}_tai",
          guard2: "␣a#{@asset.id}d␣|␣!actn#{action.id}_tai",
          update: "#{1␣-␣@asset.vulnerability}:(a#{@asset.id}d'=false)␣+␣" +
                  "#{@asset.vulnerability}:(a#{@asset.id}d'=true)"
        }
        commands += yield(params, action)
      end
    }
    puts <<-end.gsub(/ {8}/, '')
      module #{@asset.class}#{@asset.id}_Survivability
        a#{@asset.id}d : bool init false;
        #{commands}
      endmodule
    end
  end

  def self.generate_sensor_action_modules
    commands = String.new
    @asset.kinetic_actions.each { |action|
      siblings = String.new
      @asset.sensor_actions.each { |sensor_action|
        if sensor_action.siblings.include?(action.id)
          siblings += (siblings.empty? ? 'r' : '␣|␣r') + sensor_action.id
        end
      }
```

```
        unless siblings.empty?
          params = {
            action: "[actn#{action.id}]",
            guard1: "␣actn#{action.id}_tai␣&␣␣(#{siblings})␣&␣sad#{@asset.id}<
                tkad#{@asset.id}",
            guard2: "!actn#{action.id}_tai␣|␣!(#{siblings})",
            update: "(sad#{@asset.id}'=sad#{@asset.id}+1)"
          }
          commands += yield(params, action)
        end
      }
      puts <<-end.gsub(/ {8}/, '')
        module SensorActionCounter#{@asset.id}
          sad#{@asset.id} : [0..tkad#{@asset.id}] init 0;
          #{commands}
        endmodule
      end
    end

    def self.generate_properties
      puts "P=?␣[␣F␣!a#{@asset.id}d␣&␣raf#{@asset.id}>#{@asset.raf}␣]"
    end
end

execute { |operation|
  puts "survivability␣constructs␣for␣#{operation}..."
  ThreatLoader.new(operation).assets.each { |asset|
    SurvivabilityTemplate.generate(asset)
  }
}
```

Listing D.3: Ruby code for the PCTL property template.

```
require './template_util'

class ClassLoader < OpLoader; defensive_copy :actions end
class LidarAction; include ActionUtil end

module PropertyTemplate
  def self.generate(actions)
    properties = String.new
    actions.each { |action|
      property = get_property(action)
      properties += (properties.empty? ? 'd' : '␣&␣d') + "#{property}=0" unless
          property.nil?
    }
    puts "P=?␣[␣F␣#{properties}␣]"
  end

  private
    def self.get_property(action)
      if action.is_kinetic_action?
        case action.type
          when :default_singleton,
               :default_terminal,
               :terminal_observer,
               :terminal_constrained_observer,
               :terminal_subject_constraint
            action.id
        end
      end
    end
end

execute { |operation|
  puts "properties␣for␣#{operation}..."
  PropertyTemplate.generate(ClassLoader.new(operation).actions)
}
```

Listing D.4: Ruby template utility code.

```
require './util'

autoload :Asset, './asset'
autoload :ClassLoader, './class_loader'

module ActionUtil
  def self.respondent *method_names
    method_names.each { |name|
      define_method("is_#{name}_action?") { self.class.superclass.name == "#{name
          .capitalize}Action" }
    }
  end

  respondent :kinetic, :sensor
end

module TypeUtil
  attr_reader :type

  def type=(type)
    @type = type.to_sym
  end
end

class Action; include ActionUtil end

class Asset;
  include TypeUtil

  def self.reader *method_names
    method_names.each { |name|
      temp = "#{name}_actions"
      define_method(temp) {
        unless instance_variable_defined?("@#{temp}")
          instance_variable_set("@#{temp}", Array.new(@actions).delete_if { |
              action|
            !action.send("is_#{name}_action?")
          })
        end
        instance_variable_get("@#{temp}")
      }
    }
  end

  attr_reader :id
  reader :kinetic, :sensor
end

class KineticAction < Action
  include TypeUtil
  attr_reader :id
end
```

Listing D.5: Ruby code for module `Mission`.

```
require './util'

autoload :ARDrone, './asset'
autoload :TraversePathSegmentAction, './action'

module Mission
  def self.execute(assets)
    while !assets.empty?
      assets.each { |asset|
        asset.execute
        assets.delete(asset) if asset.completed?
      }
    end
  end
end

execute { |operation|
```

```
  puts "executing␣#{operation}..."
  Mission.execute(OpLoader.new(operation).assets)
}
```

Listing D.6: Ruby utility code.

```
autoload :YAML, 'yaml'

autoload :HoverAction, './hover_action'
autoload :KineticAction, './action'
autoload :LidarAction, './lidar_action'
autoload :Observer, './observer'
autoload :OpLoader, './op_loader'
autoload :Subject, './subject'

Dir['./core/*'].each { |file| require file }

def execute
  Dir['./operations/operation_*'].each { |file|
    yield(File.basename(file, '.yaml'))
  }
end
```

Listing D.7: Ruby code for class `Asset`.

```
class Asset
  class << self; attr_accessor :endurance end

  def initialize(id)
    @id = id.to_s
    @endurance = self.class.endurance
    @actions = Array.new
  end

  def add_action(action)
    @actions.push(action)
  end

  def execute
    @actions.each { |action|
      action.execute
      @actions.delete(action) if action.completed?
    }
    @endurance -= 1
  end

  def completed?
    @endurance == 0 || @actions.empty?
  end
end

class ARDrone < Asset; end
class Hummingbird < Asset; end
```

Listing D.8: Ruby code for class `ClassLoader`.

```
class ClassLoader < OpLoader
  def initialize(operation)
    super(operation)
    @class_file = load_file(operation, :classifications)
    load(:actions, :type)
    load(:assets, :type)
  end

  private
    def load(name, *attributes)
      singular_name = name.capitalize.singularize
      if @class_file.include?(singular_name)
        @class_file.fetch(singular_name).each_value { |collection|
          collection.symbolize_keys!.each { |item|
```

```
            attributes.each { |attribute|
              if item.include?(attribute)
                instance_variable_get("@#{name}").fetch(item.fetch(:id)).send("#{
                    attribute}=", item.fetch(attribute))
              end
            }
          }
        }
      end
    end
end
```

Listing D.9: Ruby code for class `HoverAction`.

```
class HoverAction < KineticAction
  def initialize(id, duration)
    super id, duration
    @concurrencies = Array.new
  end

  def add_concurrency(action)
    @concurrencies.push(action)
  end

  def execute
    if @subjects.empty?
      @concurrencies.each { |action| action.execute }
      execute_naively
    end
  end
end
```

Listing D.10: Ruby code for class `Action`.

```
class Action
  include Subject
  include Observer

  def initialize(id, duration)
    super()
    @id = id.to_s
    @duration = duration
  end

  def add_precondition(action)
    add_subject(action)
  end

  def execute
    execute_naively if @subjects.empty?
  end

  def completed?
    @duration == 0
  end

  private
    def execute_naively
      @duration -= 1
      notify if completed?
    end
end

class KineticAction < Action; end
class SensorAction < Action; end
class PhotoSurveillanceAction < SensorAction; end
class TraversePathSegmentAction < KineticAction; end
```

Listing D.11: Ruby code for class `LidarAction`.

```
class LidarAction
  include Subject

  def initialize(id, interval)
    super()
    @id = id
    @interval = interval
    @readings = 0
    @timestep = 0
  end

  def execute
    @readings += 1 if @timestep % @interval == 0
    @timestep += 1
  end
end
```

Listing D.12: Ruby code for module `Observer`.

```
module Observer
  def initialize
    super()
    @subjects = Array.new
  end

  protected
    def add_subject(obj)
      @subjects.push(obj)
      obj.attach(self)
    end

    def update(obj)
      @subjects.delete(obj)
      obj.detach(self)
    end
end
```

Listing D.13: Ruby code for class `OpLoader`.

```
class OpLoader
  def self.loader *method_names
    method_names.each { |name|
      define_method("load_#{name}") {
        instance_variable_get("@#{name}").each_pair { |observer_id, subject_ids|
          subject_ids.each { |subject_id|
            case name
              when :concurrencies then id_1, id_2 = subject_id, observer_id
              when :preconditions then id_1, id_2 = observer_id, subject_id
            end
            @actions.fetch(id_1).send("add_#{name.singularize}", @actions.fetch(
                id_2))
          }
        }
      }
    }
  end

  defensive_copy :assets
  loader :concurrencies, :preconditions

  def initialize(operation)
    @op_file = load_file operation
    collections = [:actions, :concurrencies, :preconditions, :assets]
    collections.each { |collection| instance_variable_set("@#{collection}", Hash.
        new) }
    collections.each { |collection| send("load_#{collection}") }
  end

  private
    def load_file(file, dir = :operations)
      YAML::load(File.open("#{dir}/#{file}.yaml")).symbolize_keys!
```

```
    end

    def load_actions
      @op_file.fetch(:Action).each_pair { |type, actions|
        actions.symbolize_keys!.each { |action|
          id = action.fetch(:id)
          @actions[id] = type.to_c.new(id, type == :LidarAction ? action.fetch(:
              interval) : action.fetch(:duration))
          @concurrencies[id] = action.fetch(:concurrencies) if action.include?(:
              concurrencies)
          @preconditions[id] = action.fetch(:preconditions) if action.include?(:
              preconditions)
        }
      }
    end

    def load_assets
      load_file(:asset_data).each_pair { |type, asset| type.to_c.endurance =
          asset.fetch(:endurance) }
      @op_file.fetch(:Asset).each_pair { |type, assets|
        assets.symbolize_keys!.each { |asset|
          id = asset.fetch(:id)
          @assets[id] = type.to_c.new(id)
          asset.fetch(:actions).each { |action| @assets[id].add_action(@actions.
              fetch(action)) }
        }
      }
    end
end
```

Listing D.14: Ruby code for module `Subject`.

```
module Subject
  def initialize
    @observers = Array.new
  end

  protected
    def attach(obj)
      @observers.push(obj)
    end

    def detach(obj)
      @observers.delete(obj)
    end

    def notify
      @observers.each { |observer| observer.update(self) }
    end
end
```

Listing D.15: Code that modifies the Ruby core class `Array`.

```
class Array
  def symbolize_keys!
    self.each { |item| item.symbolize_keys! if item.class == Hash }
    self
  end
end
```

Listing D.16: Code that modifies the Ruby core class `Hash`.

```
class Hash
  def symbolize_keys!
    keys.each { |key| self[(key.to_sym rescue key)] = delete(key) }
    values.each { |value| value.symbolize_keys! if value.class == Hash }
    self
  end
end
```

Listing D.17: Code that modifies the Ruby core class `Module`.

```
class Module
  def defensive_copy *method_names
    method_names.each { |name|
      define_method(name) {
        var = instance_variable_get("@#{name}")
        Array.new(var.class == Array ? var : var.values)
      }
    }
  end
end
```

Listing D.18: Code that modifies the Ruby core class `Symbol`.

```
require 'active_support/inflector'

class Symbol
  def include?(sym)
    to_s.include?(sym.to_s)
  end

  def singularize
    to_s.singularize.intern
  end

  def to_c
    Kernel.const_get(self)
  end
end
```

Appendix E

DSL Schema

Listing E.1: A schema definition, encoded in the Kwalify schema language, for the YAML DSL presented in this thesis.

```
type: map
required: yes
mapping:
  "Action":
    type: map
    required: yes
    mapping:
      "HoverAction":
        type: seq
        sequence:
          - type: map
            required: yes
            mapping:
              "id":
                type: int
                required: yes
              "duration":
                type: int
                required: yes
              "preconditions":
                type: seq
                sequence:
                  - type: int
              "coordinates":
                type: seq
                sequence:
                  - type: float
      "LidarAction":
        type: seq
        sequence:
          - type: map
            required: yes
            mapping:
              "id":
                type: int
                required: yes
              "interval":
                type: int
                required: yes
              "concurrencies":
                type: seq
                required: yes
                sequence:
                  - type: int
      "PhotoSurveillanceAction":
        type: seq
        sequence:
          - type: map
            required: yes
            mapping:
              "id":
                type: int
                required: yes
```

```
            "duration":
              type: int
              required: yes
            "preconditions":
              type: seq
              sequence:
                - type: int
    "TraversePathSegmentAction":
      type: seq
      sequence:
        - type: map
          required: yes
          mapping:
            "id":
              type: int
              required: yes
            "duration":
              type: int
              required: yes
            "preconditions":
              type: seq
              sequence:
                - type: int
            "coordinates":
              type: seq
              sequence:
                - type: float
"Asset":
  type: map
  required: yes
  mapping:
    "ARDrone":
      type: seq
      sequence:
        - type: map
          required: yes
          mapping:
            "id":
              type: int
              required: yes
            "endurance":
              type: int
            "actions":
              type: seq
              required: yes
              sequence:
                - type: int
    "Hummingbird":
      type: seq
      sequence:
        - type: map
          required: yes
          mapping:
            "id":
              type: int
              required: yes
            "endurance":
              type: int
            "actions":
              type: seq
              required: yes
              sequence:
                - type: int
```

Appendix F

Mission Verification Artifacts

Listing F.1: YAML input for Mission 1a.

```
Action:
  HoverAction:
    - id: 1
      duration: 69
      coordinates: [-112.1334152514208, 36.08362433106716]
Asset:
  ARDrone:
    - id: 1
      actions: [1]
```

Listing F.2: DTMC output for Mission 1a.

```
dtmc

const int max_e1 = 70; // maximum endurance
const int max_d1 = 69; // maximum duration

module ARDrone1

    e1 : [0..max_e1] init max_e1; // endurance

    [actn1] e1>0 & d1>0 -> (e1'=e1-1);
    []      e1=0 | d1=0 -> true;

endmodule

module HoverAction1

    d1 : [0..max_d1] init max_d1; // duration

    [actn1] d1>0 & e1>0 -> (d1'=d1-1);

endmodule
```

Listing F.3: PCTL output for Mission 1a.

```
P=? [ F d1=0 ]
```

Listing F.4: Log output for Mission 1a.

```
Model checking: P=? [ F d1=0 ]

Building model...

Computing reachable states...

Reachability (BFS): 70 iterations in 0.01 seconds (average 0.000171, setup 0.00)

Time for model construction: 0.034 seconds.

Type:        DTMC
```

```
States:      70 (1 initial)
Transitions: 70

Transition matrix: 720 nodes (2 terminal), 70 minterms, vars: 14r/14c

Prob0: 70 iterations in 0.01 seconds (average 0.000086, setup 0.00)

Prob1: 1 iterations in 0.00 seconds (average 0.000000, setup 0.00)

yes = 70, no = 0, maybe = 0

Value in the initial state: 1.0

Time for model checking: 0.0070 seconds.

Result: 1.0 (value in the initial state)
```

Listing F.5: YAML input for Mission 1d.

```
Action:
  HoverAction:
    - id: 1
      duration: 40
      coordinates: [-112.1334152514208, 36.08362433106716]
    - id: 2
      duration: 30
      preconditions: [1]
Asset:
  ARDrone:
    - id: 1
      actions: [1, 2]
```

Listing F.6: DTMC output for Mission 1d.

```
dtmc

const int max_e1 = 70; // maximum endurance
const int max_d1 = 40; // maximum duration
const int max_d2 = 30; // maximum duration

module ARDrone1

    e1 : [0..max_e1] init max_e1; // endurance

    [actn1] e1>0 & d1>0 -> (e1'=e1-1);
    [actn2] e1>0 & d2>0 -> (e1'=e1-1);
    []      e1=0 | d2=0 -> true;

endmodule

module HoverAction1

    d1 : [0..max_d1] init max_d1; // duration

    [actn1] d1>0 & e1>0 -> (d1'=d1-1);

endmodule

module HoverAction2

    d2 : [0..max_d2] init max_d2; // duration

    [actn2] d1=0 & d2>0 & e1>0 -> (d2'=d2-1);

endmodule
```

Listing F.7: PCTL output for Mission 1d.

```
P=? [ F d2=0 ]
```

Listing F.8: Log output for Mission 1d.

```
Model checking: P=? [ F d2=0 ]

Building model...

Computing reachable states...

Reachability (BFS): 71 iterations in 0.01 seconds (average 0.000099, setup 0.00)

Time for model construction: 0.043 seconds.

Type:        DTMC
States:      71 (1 initial)
Transitions: 71

Transition matrix: 1031 nodes (2 terminal), 71 minterms, vars: 18r/18c

Prob0: 71 iterations in 0.01 seconds (average 0.000127, setup 0.00)

Prob1: 1 iterations in 0.00 seconds (average 0.000000, setup 0.00)

yes = 71, no = 0, maybe = 0

Value in the initial state: 1.0

Time for model checking: 0.0090 seconds.

Result: 1.0 (value in the initial state)
```

Listing F.9: YAML input for Mission 1g.

```
Action:
  HoverAction:
    - id: 1
      duration: 40
      coordinates: [-112.1334152514208, 36.08362433106716]
    - id: 2
      duration: 40
      preconditions: [1]
    - id: 3
      duration: 40
      preconditions: [2]
Asset:
  Hummingbird:
    - id: 1
      actions: [1, 2, 3]
```

Listing F.10: DTMC output for Mission 1g.

```
dtmc

const int max_e1 = 120; // maximum endurance
const int max_d1 = 40;  // maximum duration
const int max_d2 = 40;  // maximum duration
const int max_d3 = 40;  // maximum duration

module Hummingbird1

    e1 : [0..max_e1] init max_e1; // endurance

    [actn1] e1>0 & d1>0 -> (e1'=e1-1);
    [actn2] e1>0 & d2>0 -> (e1'=e1-1);
    [actn3] e1>0 & d3>0 -> (e1'=e1-1);
    []      e1=0 | d3=0 -> true;

endmodule

module HoverAction1

    d1 : [0..max_d1] init max_d1; // duration
```

```
    [actn1] d1>0 & e1>0 -> (d1'=d1-1);

endmodule

module HoverAction2

    d2 : [0..max_d2] init max_d2; // duration

    [actn2] d1=0 & d2>0 & e1>0 -> (d2'=d2-1);

endmodule

module HoverAction3 = HoverAction2[d2=d3, max_d2=max_d3, actn2=actn3, d1=d2]
    endmodule
```

Listing F.11: PCTL output for Mission 1g.

```
P=? [ F d3=0 ]
```

Listing F.12: Log output for Mission 1g.

```
Model checking: P=? [ F d3=0 ]

Building model...

Computing reachable states...

Reachability (BFS): 121 iterations in 0.03 seconds (average 0.000240, setup 0.00)

Time for model construction: 0.188 seconds.

Type:        DTMC
States:      121 (1 initial)
Transitions: 121

Transition matrix: 2639 nodes (2 terminal), 121 minterms, vars: 25r/25c

Prob0: 121 iterations in 0.03 seconds (average 0.000281, setup 0.00)

Prob1: 1 iterations in 0.00 seconds (average 0.000000, setup 0.00)

yes = 121, no = 0, maybe = 0

Value in the initial state: 1.0

Time for model checking: 0.033 seconds.

Result: 1.0 (value in the initial state)
```

Listing F.13: YAML input for Mission 2a.

```
Action:
  HoverAction:
    - id: 1
      duration: 40
      coordinates: [-112.1334152514208, 36.08362433106716]
    - id: 2
      duration: 29
      preconditions: [1]
    - id: 3
      duration: 39
      coordinates: [-112.1199316051625, 36.07823185748754]
Asset:
  ARDrone:
    - id: 1
      actions: [1, 2]
  Hummingbird:
    - id: 2
      actions: [3]
```

Listing F.14: DTMC output for Mission 2a.

```
dtmc

const int max_e1 = 70;  // maximum endurance
const int max_d1 = 40;  // maximum duration
const int max_d2 = 29;  // maximum duration
const int max_e2 = 120; // maximum endurance
const int max_d3 = 39;  // maximum duration

module ARDrone1

    e1 : [0..max_e1] init max_e1; // endurance

    [actn1] e1>0 & d1>0 -> (e1'=e1-1);
    [actn2] e1>0 & d2>0 -> (e1'=e1-1);
    []      e1=0 | d2=0 -> true;

endmodule

module HoverAction1

    d1 : [0..max_d1] init max_d1; // duration

    [actn1] d1>0 & e1>0 -> (d1'=d1-1);

endmodule

module HoverAction2

    d2 : [0..max_d2] init max_d2; // duration

    [actn2] d1=0 & d2>0 & e1>0 -> (d2'=d2-1);

endmodule

module Hummingbird2

    e2 : [0..max_e2] init max_e2; // endurance

    [actn3] e2>0 & d3>0 -> (e2'=e2-1);
    []      e2=0 | d3=0 -> true;

endmodule

module HoverAction3 = HoverAction1[d1=d3, max_d1=max_d3, actn1=actn3, e1=e2]
    endmodule
```

Listing F.15: PCTL output for Mission 2a.

```
P=? [ F d2=0 & d3=0 ]
```

Listing F.16: Log output for Mission 2a.

```
Model checking: P=? [ F d2=0&d3=0 ]

Building model...

Computing reachable states...

Reachability (BFS): 109 iterations in 0.13 seconds (average 0.001220, setup 0.00)

Time for model construction: 0.204 seconds.

Type:        DTMC
States:      2800 (1 initial)
Transitions: 5599

Transition matrix: 2541 nodes (3 terminal), 5599 minterms, vars: 31r/31c

Prob0: 109 iterations in 0.20 seconds (average 0.001789, setup 0.00)
```

```
Prob1: 1 iterations in 0.00 seconds (average 0.000000, setup 0.00)

yes = 2800, no = 0, maybe = 0

Value in the initial state: 1.0

Time for model checking: 0.202 seconds.

Result: 1.0 (value in the initial state)
```

Listing F.17: YAML input for Mission 2b.

```
Action:
  HoverAction:
    - id: 1
      duration: 40
      coordinates: [-112.1334152514208, 36.08362433106716]
    - id: 2
      duration: 29
      preconditions: [1, 3]
    - id: 3
      duration: 39
      coordinates: [-112.1199316051625, 36.07823185748754]
Asset:
  ARDrone:
    - id: 1
      actions: [1, 2]
  Hummingbird:
    - id: 2
      actions: [3]
```

Listing F.18: DTMC output for Mission 2b.

```
dtmc

const int max_e1 = 70;  // maximum endurance
const int max_d1 = 40;  // maximum duration
const int max_d2 = 29;  // maximum duration
const int max_e2 = 120; // maximum endurance
const int max_d3 = 39;  // maximum duration

module ARDrone1

    e1 : [0..max_e1] init max_e1; // endurance

    [asst1] e1>0 & d1>0 -> (e1'=e1-1);
    [actn2] e1>0 & d2>0 -> (e1'=e1-1);
    []      e1=0 | d2=0 -> true;

endmodule

module HoverAction1

    d1 : [0..max_d1] init max_d1; // duration

    [asst1] d1>0 & e1>0 -> (d1'=d1-1);

endmodule

module HoverAction2

    d2 : [0..max_d2] init max_d2; // duration

    [actn2] d1=0 & d3=0 & d2>0 & e1>0 -> (d2'=d2-1);

endmodule

module Hummingbird2

    e2 : [0..max_e2] init max_e2; // endurance
```

```
    [asst1] e2>0 & d3>0 -> (e2'=e2-1);
    [asst1] e2=0 | d3=0 -> true;

endmodule

module HoverAction3

    d3 : [0..max_d3] init max_d3; // duration

    [asst1] d3>0 & e2>0 -> (d3'=d3-1);
    [asst1] d3=0        -> true;

endmodule
```

Listing F.19: PCTL output for Mission 2b.

```
P=? [ F d2=0 ]
```

Listing F.20: Log output for Mission 2b.

```
Model checking: P=? [ F d2=0 ]

Building model...

Computing reachable states...

Reachability (BFS): 70 iterations in 0.02 seconds (average 0.000300, setup 0.00)

Time for model construction: 0.101 seconds.

Type:        DTMC
States:      70 (1 initial)
Transitions: 70

Transition matrix: 2456 nodes (2 terminal), 70 minterms, vars: 31r/31c

Prob0: 70 iterations in 0.02 seconds (average 0.000257, setup 0.00)

Prob1: 1 iterations in 0.00 seconds (average 0.000000, setup 0.00)

yes = 70, no = 0, maybe = 0

Value in the initial state: 1.0

Time for model checking: 0.018 seconds.

Result: 1.0 (value in the initial state)
```

Listing F.21: YAML input for Mission 2e.

```
Action:
  HoverAction:
    - id: 1
      duration: 40
      coordinates: [-112.1334152514208, 36.08362433106716]
    - id: 2
      duration: 29
      preconditions: [1, 3, 4]
    - id: 3
      duration: 40
      coordinates: [-112.1199316051625, 36.07823185748754]
    - id: 4
      duration: 40
      coordinates: [-112.1075818315498, 36.08838518690878]
Asset:
  ARDrone:
    - id: 1
      actions: [1, 2]
  Hummingbird:
    - id: 2
```

```
      actions: [3]
    - id: 3
      actions: [4]
```

Listing F.22: DTMC output for Mission 2e.

```
dtmc

const int max_e1 = 70;  // maximum endurance
const int max_d1 = 40;  // maximum duration
const int max_d2 = 29;  // maximum duration
const int max_e2 = 120; // maximum endurance
const int max_d3 = 40;  // maximum duration
const int max_e3 = 120; // maximum endurance
const int max_d4 = 40;  // maximum duration

module ARDrone1

    e1 : [0..max_e1] init max_e1; // endurance

    [asst1] e1>0 & d1>0 -> (e1'=e1-1);
    [actn2] e1>0 & d2>0 -> (e1'=e1-1);
    []      e1=0 | d2=0 -> true;

endmodule

module HoverAction1

    d1 : [0..max_d1] init max_d1; // duration

    [asst1] d1>0 & e1>0 -> (d1'=d1-1);

endmodule

module HoverAction2

    d2 : [0..max_d2] init max_d2; // duration

    [actn2] d1=0 & d3=0 & d4=0 & d2>0 & e1>0 -> (d2'=d2-1);

endmodule

module Hummingbird2

    e2 : [0..max_e2] init max_e2; // endurance

    [asst1] e2>0 & d3>0 -> (e2'=e2-1);
    [asst1] e2=0 | d3=0 -> true;

endmodule

module HoverAction3

    d3 : [0..max_d3] init max_d3; // duration

    [asst1] d3>0 & e2>0 -> (d3'=d3-1);
    [asst1] d3=0        -> true;

endmodule

module Hummingbird3 = Hummingbird2[e2=e3, max_e2=max_e3, d3=d4] endmodule
module HoverAction4 = HoverAction3[d3=d4, max_d3=max_d4, e2=e3] endmodule
```

Listing F.23: PCTL output for Mission 2e.

```
P=? [ F d2=0 ]
```

Listing F.24: Log output for Mission 2e.

```
Model checking: P=? [ F d2=0 ]
```

```
Building model...

Computing reachable states...

Reachability (BFS): 70 iterations in 0.03 seconds (average 0.000457, setup 0.00)

Time for model construction: 0.192 seconds.

Type:        DTMC
States:      70 (1 initial)
Transitions: 70

Transition matrix: 3539 nodes (2 terminal), 70 minterms, vars: 44r/44c

Prob0: 70 iterations in 0.02 seconds (average 0.000343, setup 0.00)

Prob1: 1 iterations in 0.00 seconds (average 0.000000, setup 0.00)

yes = 70, no = 0, maybe = 0

Value in the initial state: 1.0

Time for model checking: 0.026 seconds.

Result: 1.0 (value in the initial state)
```

Listing F.25: YAML input for Mission 2g.

```
Action:
  HoverAction:
    - id: 1
      duration: 40
      coordinates: [-112.1334152514208, 36.08362433106716]
    - id: 2
      duration: 40
      preconditions: [1]
    - id: 3
      duration: 39
      preconditions: [2, 4]
    - id: 4
      duration: 39
      preconditions: [1]
      coordinates: [-112.1199316051625, 36.07823185748754]
    - id: 5
      duration: 29
      preconditions: [4]
Asset:
  Hummingbird:
    - id: 1
      actions: [1, 2, 3]
  ARDrone:
    - id: 2
      actions: [4, 5]
```

Listing F.26: DTMC output for Mission 2g.

```
dtmc

const int max_e1 = 120; // maximum endurance
const int max_d1 = 40;  // maximum duration
const int max_d2 = 40;  // maximum duration
const int max_d3 = 39;  // maximum duration
const int max_e2 = 70;  // maximum endurance
const int max_d4 = 39;  // maximum duration
const int max_d5 = 29;  // maximum duration

module Hummingbird1

    e1 : [0..max_e1] init max_e1; // endurance
```

```
    [actn1] e1>0 & d1>0 -> (e1'=e1-1);
    [asst1] e1>0 & d2>0 -> (e1'=e1-1);
    [asst1] e1>0 & d3>0 -> (e1'=e1-1);
    [asst1] e1=0 | d3=0 -> true;

endmodule

module HoverAction1

    d1 : [0..max_d1] init max_d1; // duration

    [actn1] d1>0 & e1>0 -> (d1'=d1-1);

endmodule

module HoverAction2

    d2 : [0..max_d2] init max_d2; // duration

    [asst1] d1=0 & d2>0 & e1>0 -> (d2'=d2-1);
    [asst1]        d2=0        -> true;

endmodule

module HoverAction3

    d3 : [0..max_d3] init max_d3; // duration

    [asst1] d2>0                       -> true;
    [asst1] d2=0 & d4=0 & d3>0 & e1>0 -> (d3'=d3-1);
    [asst1]               d3=0        -> true;

endmodule

module ARDrone2

    e2 : [0..max_e2] init max_e2; // endurance

    [asst1] e2>0 & d4>0 -> (e2'=e2-1);
    [asst1] e2>0 & d5>0 -> (e2'=e2-1);
    [asst1] e2=0 | d5=0 -> true;

endmodule

module HoverAction4

    d4 : [0..max_d4] init max_d4; // duration

    [asst1] d4>0 & e2>0 -> (d4'=d4-1);
    [asst1] d4=0        -> true;

endmodule

module HoverAction5

    d5 : [0..max_d5] init max_d5; // duration

    [asst1] d4>0               -> true;
    [asst1] d4=0 & d5>0 & e2>0 -> (d5'=d5-1);
    [asst1]        d5=0        -> true;

endmodule
```

Listing F.27: PCTL output for Mission 2g.

```
P=? [ F d3=0 & d5=0 ]
```

Listing F.28: Log output for Mission 2g.

```
Model checking: P=? [ F d3=0&d5=0 ]
```

```
Building model...

Computing reachable states...

Reachability (BFS): 120 iterations in 0.05 seconds (average 0.000442, setup 0.00)

Time for model construction: 0.325 seconds.

Type:        DTMC
States:      120 (1 initial)
Transitions: 120

Transition matrix: 5292 nodes (2 terminal), 120 minterms, vars: 43r/43c

Prob0: 120 iterations in 0.07 seconds (average 0.000617, setup 0.00)

Prob1: 1 iterations in 0.00 seconds (average 0.000000, setup 0.00)

yes = 120, no = 0, maybe = 0

Value in the initial state: 1.0

Time for model checking: 0.072 seconds.

Result: 1.0 (value in the initial state)
```

Listing F.29: YAML input for Mission 2j.

```
Action:
  HoverAction:
    - id: 1
      duration: 40
      coordinates: [-112.1334152514208, 36.08362433106716]
    - id: 2
      duration: 40
      preconditions: [1]
    - id: 3
      duration: 40
      preconditions: [2, 5]
    - id: 4
      duration: 40
      coordinates: [-112.1199316051625, 36.07823185748754]
    - id: 5
      duration: 39
      preconditions: [4]
Asset:
  Hummingbird:
    - id: 1
      actions: [1, 2, 3]
    - id: 2
      actions: [4, 5]
```

Listing F.30: DTMC output for Mission 2j.

```
dtmc

const int max_e1 = 120; // maximum endurance
const int max_d1 = 40;  // maximum duration
const int max_d2 = 40;  // maximum duration
const int max_d3 = 40;  // maximum duration
const int max_e2 = 120; // maximum endurance
const int max_d4 = 40;  // maximum duration
const int max_d5 = 39;  // maximum duration

module Hummingbird1

    e1 : [0..max_e1] init max_e1; // endurance

    [asst1] e1>0 & d1>0 -> (e1'=e1-1);
    [asst1] e1>0 & d2>0 -> (e1'=e1-1);
    [actn3] e1>0 & d3>0 -> (e1'=e1-1);
```

```
    []      e1=0 | d3=0 -> true;

endmodule

module HoverAction1

    d1 : [0..max_d1] init max_d1; // duration

    [asst1] d1>0 & e1>0 -> (d1'=d1-1);
    [asst1] d1=0        -> true;

endmodule

module HoverAction2

    d2 : [0..max_d2] init max_d2; // duration

    [asst1] d1>0                -> true;
    [asst1] d1=0 & d2>0 & e1>0 -> (d2'=d2-1);

endmodule

module HoverAction3

    d3 : [0..max_d3] init max_d3; // duration

    [actn3] d2=0 & d5=0 & d3>0 & e1>0 -> (d3'=d3-1);

endmodule

module Hummingbird2

    e2 : [0..max_e2] init max_e2; // endurance

    [asst1] e2>0 & d4>0 -> (e2'=e2-1);
    [asst1] e2>0 & d5>0 -> (e2'=e2-1);
    [asst1] e2=0 | d5=0 -> true;

endmodule

module HoverAction4

    d4 : [0..max_d4] init max_d4; // duration

    [asst1] d4>0 & e2>0 -> (d4'=d4-1);
    [asst1] d4=0        -> true;

endmodule

module HoverAction5

    d5 : [0..max_d5] init max_d5; // duration

    [asst1] d4>0                -> true;
    [asst1] d4=0 & d5>0 & e2>0 -> (d5'=d5-1);
    [asst1]        d5=0        -> true;

endmodule
```

Listing F.31: PCTL output for Mission 2j.

```
P=? [ F d3=0 ]
```

Listing F.32: Log output for Mission 2j.

```
Model checking: P=? [ F d3=0 ]

Building model...

Computing reachable states...
```

```
Reachability (BFS): 121 iterations in 0.09 seconds (average 0.000752, setup 0.00)

Time for model construction: 0.445 seconds.

Type:        DTMC
States:      121 (1 initial)
Transitions: 121

Transition matrix: 6636 nodes (2 terminal), 121 minterms, vars: 44r/44c

Prob0: 121 iterations in 0.08 seconds (average 0.000645, setup 0.00)

Prob1: 1 iterations in 0.00 seconds (average 0.001000, setup 0.00)

yes = 121, no = 0, maybe = 0

Value in the initial state: 1.0

Time for model checking: 0.077 seconds.

Result: 1.0 (value in the initial state)
```

Listing F.33: YAML input for Mission 2m.

```
Action:
  HoverAction:
    - id: 1
      duration: 40
      coordinates: [-112.1334152514208, 36.08362433106716]
    - id: 2
      duration: 40
      preconditions: [1]
    - id: 3
      duration: 40
      preconditions: [2, 5]
    - id: 4
      duration: 40
      coordinates: [-112.1199316051625, 36.07823185748754]
    - id: 5
      duration: 39
      preconditions: [4]
    - id: 6
      duration: 40
      preconditions: [5]
Asset:
  Hummingbird:
    - id: 1
      actions: [1, 2, 3]
    - id: 2
      actions: [4, 5, 6]
```

Listing F.34: DTMC output for Mission 2m.

```
dtmc

const int max_e1 = 120; // maximum endurance
const int max_d1 = 40;  // maximum duration
const int max_d2 = 40;  // maximum duration
const int max_d3 = 40;  // maximum duration
const int max_e2 = 120; // maximum endurance
const int max_d4 = 40;  // maximum duration
const int max_d5 = 39;  // maximum duration
const int max_d6 = 40;  // maximum duration

module Hummingbird1

    e1 : [0..max_e1] init max_e1; // endurance

    [asst1] e1>0 & d1>0 -> (e1'=e1-1);
    [asst1] e1>0 & d2>0 -> (e1'=e1-1);
    [asst1] e1>0 & d3>0 -> (e1'=e1-1);
```

```
    [asst1] e1=0 | d3=0 -> true;

endmodule

module HoverAction1

    d1 : [0..max_d1] init max_d1; // duration

    [asst1] d1>0 & e1>0 -> (d1'=d1-1);
    [asst1] d1=0        -> true;

endmodule

module HoverAction2

    d2 : [0..max_d2] init max_d2; // duration

    [asst1] d1>0               -> true;
    [asst1] d1=0 & d2>0 & e1>0 -> (d2'=d2-1);
    [asst1]        d2=0        -> true;

endmodule

module HoverAction3

    d3 : [0..max_d3] init max_d3; // duration

    [asst1] d2>0                       -> true;
    [asst1] d2=0 & d5=0 & d3>0 & e1>0 -> (d3'=d3-1);
    [asst1]               d3=0         -> true;

endmodule

module Hummingbird2

    e2 : [0..max_e2] init max_e2; // endurance

    [asst1] e2>0 & d4>0 -> (e2'=e2-1);
    [asst1] e2>0 & d5>0 -> (e2'=e2-1);
    [asst1] e2>0 & d6>0 -> (e2'=e2-1);
    [asst1] e2=0 | d6=0 -> true;

endmodule

module HoverAction4

    d4 : [0..max_d4] init max_d4; // duration

    [asst1] d4>0 & e2>0 -> (d4'=d4-1);
    [asst1] d4=0        -> true;

endmodule

module HoverAction5

    d5 : [0..max_d5] init max_d5; // duration

    [asst1] d4>0               -> true;
    [asst1] d4=0 & d5>0 & e2>0 -> (d5'=d5-1);
    [asst1]        d5=0        -> true;

endmodule

module HoverAction6

    d6 : [0..max_d6] init max_d6; // duration

    [asst1] d5>0               -> true;
    [asst1] d5=0 & d6>0 & e2>0 -> (d6'=d6-1);
    [asst1]        d6=0        -> true;

endmodule
```

Listing F.35: PCTL output for Mission 2m.

```
P=? [ F d3=0 & d6=0 ]
```

Listing F.36: Log output for Mission 2m.

```
Model checking: P=? [ F d3=0&d6=0 ]

Building model...

Computing reachable states...

Reachability (BFS): 121 iterations in 0.10 seconds (average 0.000843, setup 0.00)

Time for model construction: 0.554 seconds.

Type:        DTMC
States:      121 (1 initial)
Transitions: 121

Transition matrix: 8660 nodes (2 terminal), 121 minterms, vars: 50r/50c

Prob0: 121 iterations in 0.11 seconds (average 0.000876, setup 0.00)

Prob1: 1 iterations in 0.00 seconds (average 0.000000, setup 0.00)

yes = 121, no = 0, maybe = 0

Value in the initial state: 1.0

Time for model checking: 0.107 seconds.

Result: 1.0 (value in the initial state)
```

Listing F.37: YAML input for Mission 2p.

```
Action:
  HoverAction:
    - id: 1
      duration: 40
      coordinates: [-112.1334152514208, 36.08362433106716]
    - id: 2
      duration: 40
      preconditions: [1, 4]
    - id: 3
      duration: 40
      preconditions: [2, 5]
    - id: 4
      duration: 40
      coordinates: [-112.1199316051625, 36.07823185748754]
    - id: 5
      duration: 40
      preconditions: [4]
    - id: 6
      duration: 40
      preconditions: [5]
Asset:
  Hummingbird:
    - id: 1
      actions: [1, 2, 3]
    - id: 2
      actions: [4, 5, 6]
```

Listing F.38: DTMC output for Mission 2p.

```
dtmc

const int max_e1 = 120; // maximum endurance
const int max_d1 = 40;  // maximum duration
const int max_d2 = 40;  // maximum duration
const int max_d3 = 40;  // maximum duration
```

```
const int max_e2 = 120; // maximum endurance
const int max_d4 = 40;  // maximum duration
const int max_d5 = 40;  // maximum duration
const int max_d6 = 40;  // maximum duration

module Hummingbird1

    e1 : [0..max_e1] init max_e1; // endurance

    [asst1] e1>0 & d1>0 -> (e1'=e1-1);
    [asst1] e1>0 & d2>0 -> (e1'=e1-1);
    [asst1] e1>0 & d3>0 -> (e1'=e1-1);
    [asst1] e1=0 | d3=0 -> true;

endmodule

module HoverAction1

    d1 : [0..max_d1] init max_d1; // duration

    [asst1] d1>0 & e1>0 -> (d1'=d1-1);
    [asst1] d1=0        -> true;

endmodule

module HoverAction2

    d2 : [0..max_d2] init max_d2; // duration

    [asst1] d1>0                        -> true;
    [asst1] d1=0 & d4=0 & d2>0 & e1>0 -> (d2'=d2-1);
    [asst1]               d2=0        -> true;

endmodule

module HoverAction3

    d3 : [0..max_d3] init max_d3; // duration

    [asst1] d2>0                        -> true;
    [asst1] d2=0 & d5=0 & d3>0 & e1>0 -> (d3'=d3-1);
    [asst1]               d3=0        -> true;

endmodule

module Hummingbird2

    e2 : [0..max_e2] init max_e2; // endurance

    [asst1] e2>0 & d4>0 -> (e2'=e2-1);
    [asst1] e2>0 & d5>0 -> (e2'=e2-1);
    [asst1] e2>0 & d6>0 -> (e2'=e2-1);
    [asst1] e2=0 | d6=0 -> true;

endmodule

module HoverAction4

    d4 : [0..max_d4] init max_d4; // duration

    [asst1] d4>0 & e2>0 -> (d4'=d4-1);
    [asst1] d4=0        -> true;

endmodule

module HoverAction5

    d5 : [0..max_d5] init max_d5; // duration

    [asst1] d4>0               -> true;
    [asst1] d4=0 & d5>0 & e2>0 -> (d5'=d5-1);
    [asst1]        d5=0        -> true;
```

```
endmodule

module HoverAction6

    d6 : [0..max_d6] init max_d6; // duration

    [asst1] d5>0                -> true;
    [asst1] d5=0 & d6>0 & e2>0 -> (d6'=d6-1);
    [asst1]        d6=0         -> true;

endmodule
```

Listing F.39: PCTL output for Mission 2p.

```
P=? [ F d3=0 & d6=0 ]
```

Listing F.40: Log output for Mission 2p.

```
Model checking: P=? [ F d3=0&d6=0 ]

Building model...

Computing reachable states...

Reachability (BFS): 121 iterations in 0.08 seconds (average 0.000694, setup 0.00)

Time for model construction: 0.611 seconds.

Type:        DTMC
States:      121 (1 initial)
Transitions: 121

Transition matrix: 8689 nodes (2 terminal), 121 minterms, vars: 50r/50c

Prob0: 121 iterations in 0.10 seconds (average 0.000810, setup 0.00)

Prob1: 1 iterations in 0.00 seconds (average 0.001000, setup 0.00)

yes = 121, no = 0, maybe = 0

Value in the initial state: 1.0

Time for model checking: 0.1 seconds.

Result: 1.0 (value in the initial state)
```

Listing F.41: YAML input for Mission 2r.

```
Action:
  HoverAction:
    - id: 1
      duration: 60
      coordinates: [-112.1334152514208, 36.08362433106716]
    - id: 2
      duration: 30
      preconditions: [1, 4]
    - id: 3
      duration: 30
      coordinates: [-112.1199316051625, 36.07823185748754]
    - id: 4
      duration: 30
      preconditions: [3, 5]
    - id: 5
      duration: 29
      coordinates: [-112.1075818315498, 36.08838518690878]
Asset:
  Hummingbird:
    - id: 1
      actions: [1, 2]
    - id: 2
```

```
      actions: [3, 4]
    - id: 3
      actions: [5]
```

Listing F.42: DTMC output for Mission 2r.

```
dtmc

const int max_e1 = 120; // maximum endurance
const int max_d1 = 60;  // maximum duration
const int max_d2 = 30;  // maximum duration
const int max_e2 = 120; // maximum endurance
const int max_d3 = 30;  // maximum duration
const int max_d4 = 30;  // maximum duration
const int max_e3 = 120; // maximum endurance
const int max_d5 = 29;  // maximum duration

module Hummingbird1

    e1 : [0..max_e1] init max_e1; // endurance

    [asst1] e1>0 & d1>0 -> (e1'=e1-1);
    [actn2] e1>0 & d2>0 -> (e1'=e1-1);
    []      e1=0 | d2=0 -> true;

endmodule

module HoverAction1

    d1 : [0..max_d1] init max_d1; // duration

    [asst1] d1>0 & e1>0 -> (d1'=d1-1);

endmodule

module HoverAction2

    d2 : [0..max_d2] init max_d2; // duration

    [actn2] d1=0 & d4=0 & d2>0 & e1>0 -> (d2'=d2-1);

endmodule

module Hummingbird2

    e2 : [0..max_e2] init max_e2; // endurance

    [asst1] e2>0 & d3>0 -> (e2'=e2-1);
    [asst1] e2>0 & d4>0 -> (e2'=e2-1);
    [asst1] e2=0 | d4=0 -> true;

endmodule

module HoverAction3

    d3 : [0..max_d3] init max_d3; // duration

    [asst1] d3>0 & e2>0 -> (d3'=d3-1);
    [asst1] d3=0        -> true;

endmodule

module HoverAction4

    d4 : [0..max_d4] init max_d4; // duration

    [asst1] d3>0                       -> true;
    [asst1] d3=0 & d5=0 & d4>0 & e2>0 -> (d4'=d4-1);
    [asst1]               d4=0         -> true;

endmodule
```

```
module Hummingbird3

    e3 : [0..max_e3] init max_e3; // endurance

    [asst1] e3>0 & d5>0 -> (e3'=e3-1);
    [asst1] e3=0 | d5=0 -> true;

endmodule

module HoverAction5

    d5 : [0..max_d5] init max_d5; // duration

    [asst1] d5>0 & e3>0 -> (d5'=d5-1);
    [asst1] d5=0        -> true;

endmodule
```

Listing F.43: PCTL output for Mission 2r.

```
P=? [ F d2=0 ]
```

Listing F.44: Log output for Mission 2r.

```
Model checking: P=? [ F d2=0 ]

Building model...

Computing reachable states...

Reachability (BFS): 91 iterations in 0.05 seconds (average 0.000571, setup 0.00)

Time for model construction: 0.392 seconds.

Type:        DTMC
States:      91 (1 initial)
Transitions: 91

Transition matrix: 4675 nodes (2 terminal), 91 minterms, vars: 47r/47c

Prob0: 91 iterations in 0.04 seconds (average 0.000429, setup 0.00)

Prob1: 1 iterations in 0.00 seconds (average 0.000000, setup 0.00)

yes = 91, no = 0, maybe = 0

Value in the initial state: 1.0

Time for model checking: 0.04 seconds.

Result: 1.0 (value in the initial state)
```

Listing F.45: YAML input for Mission 2u.

```
Action:
  HoverAction:
    - id: 1
      duration: 60
      coordinates: [-112.1334152514208, 36.08362433106716]
    - id: 2
      duration: 30
      preconditions: [1, 5]
    - id: 3
      duration: 30
      preconditions: [2]
    - id: 4
      duration: 30
      coordinates: [-112.1199316051625, 36.07823185748754]
    - id: 5
      duration: 30
```

```
      preconditions: [4, 6]
    - id: 6
      duration: 30
      coordinates: [-112.1075818315498, 36.08838518690878]
Asset:
  Hummingbird:
    - id: 1
      actions: [1, 2, 3]
    - id: 2
      actions: [4, 5]
    - id: 3
      actions: [6]
```

Listing F.46: DTMC output for Mission 2u.

```
dtmc

const int max_e1 = 120; // maximum endurance
const int max_d1 = 60;  // maximum duration
const int max_d2 = 30;  // maximum duration
const int max_d3 = 30;  // maximum duration
const int max_e2 = 120; // maximum endurance
const int max_d4 = 30;  // maximum duration
const int max_d5 = 30;  // maximum duration
const int max_e3 = 120; // maximum endurance
const int max_d6 = 30;  // maximum duration

module Hummingbird1

    e1 : [0..max_e1] init max_e1; // endurance

    [asst1] e1>0 & d1>0 -> (e1'=e1-1);
    [actn2] e1>0 & d2>0 -> (e1'=e1-1);
    [actn3] e1>0 & d3>0 -> (e1'=e1-1);
    []      e1=0 | d3=0 -> true;

endmodule

module HoverAction1

    d1 : [0..max_d1] init max_d1; // duration

    [asst1] d1>0 & e1>0 -> (d1'=d1-1);

endmodule

module HoverAction2

    d2 : [0..max_d2] init max_d2; // duration

    [actn2] d1=0 & d4=0 & d2>0 & e1>0 -> (d2'=d2-1);

endmodule

module HoverAction3

    d3 : [0..max_d3] init max_d3; // duration

    [actn3] d2=0 & d3>0 & e1>0 -> (d3'=d3-1);

endmodule

module Hummingbird2

    e2 : [0..max_e2] init max_e2; // endurance

    [asst1] e2>0 & d4>0 -> (e2'=e2-1);
    [asst1] e2>0 & d5>0 -> (e2'=e2-1);
    [asst1] e2=0 | d5=0 -> true;

endmodule
```

```
module HoverAction4

    d4 : [0..max_d4] init max_d4; // duration

    [asst1] d4>0 & e2>0 -> (d4'=d4-1);
    [asst1] d4=0        -> true;

endmodule

module HoverAction5

    d5 : [0..max_d5] init max_d5; // duration

    [asst1] d4>0                        -> true;
    [asst1] d4=0 & d6=0 & d5>0 & e2>0 -> (d5'=d5-1);
    [asst1]               d5=0          -> true;

endmodule

module Hummingbird3

    e3 : [0..max_e3] init max_e3; // endurance

    [asst1] e3>0 & d6>0 -> (e3'=e3-1);
    [asst1] e3=0 | d6=0 -> true;

endmodule

module HoverAction6

    d6 : [0..max_d6] init max_d6; // duration

    [asst1] d6>0 & e3>0 -> (d6'=d6-1);
    [asst1] d6=0        -> true;

endmodule
```

Listing F.47: PCTL output for Mission 2u.

```
P=? [ F d3=0 ]
```

Listing F.48: Log output for Mission 2u.

```
Model checking: P=? [ F d3=0 ]

Building model...

Computing reachable states...

Reachability (BFS): 121 iterations in 0.10 seconds (average 0.000785, setup 0.00)

Time for model construction: 0.54 seconds.

Type:        DTMC
States:      121 (1 initial)
Transitions: 121

Transition matrix: 6210 nodes (2 terminal), 121 minterms, vars: 52r/52c

Prob0: 121 iterations in 0.07 seconds (average 0.000554, setup 0.00)

Prob1: 1 iterations in 0.00 seconds (average 0.000000, setup 0.00)

yes = 121, no = 0, maybe = 0

Value in the initial state: 1.0

Time for model checking: 0.067 seconds.

Result: 1.0 (value in the initial state)
```

Listing F.49: YAML input for Mission 2v.

```
Action:
  HoverAction:
    - id: 1
      duration: 60
      coordinates: [-112.1334152514208, 36.08362433106716]
    - id: 2
      duration: 30
      preconditions: [1, 4]
    - id: 3
      duration: 30
      coordinates: [-112.1199316051625, 36.07823185748754]
    - id: 4
      duration: 30
      preconditions: [3, 6]
    - id: 5
      duration: 30
      preconditions: [4]
    - id: 6
      duration: 30
      coordinates: [-112.1075818315498, 36.08838518690878]
Asset:
  Hummingbird:
    - id: 1
      actions: [1, 2]
    - id: 2
      actions: [3, 4, 5]
    - id: 3
      actions: [6]
```

Listing F.50: DTMC output for Mission 2v.

```
dtmc

const int max_e1 = 120; // maximum endurance
const int max_d1 = 60;  // maximum duration
const int max_d2 = 30;  // maximum duration
const int max_e2 = 120; // maximum endurance
const int max_d3 = 30;  // maximum duration
const int max_d4 = 30;  // maximum duration
const int max_d5 = 30;  // maximum duration
const int max_e3 = 120; // maximum endurance
const int max_d6 = 30;  // maximum duration

module Hummingbird1

    e1 : [0..max_e1] init max_e1; // endurance

    [asst1] e1>0 & d1>0 -> (e1'=e1-1);
    [asst1] e1>0 & d2>0 -> (e1'=e1-1);
    [asst1] e1=0 | d2=0 -> true;

endmodule

module HoverAction1

    d1 : [0..max_d1] init max_d1; // duration

    [asst1] d1>0 & e1>0 -> (d1'=d1-1);
    [asst1] d1=0        -> true;

endmodule

module HoverAction2

    d2 : [0..max_d2] init max_d2; // duration

    [asst1] d1>0                        -> true;
    [asst1] d1=0 & d4=0 & d2>0 & e1>0 -> (d2'=d2-1);
    [asst1]               d2=0          -> true;

endmodule
```

```
module Hummingbird2

    e2 : [0..max_e2] init max_e2; // endurance

    [asst1] e2>0 & d3>0 -> (e2'=e2-1);
    [asst1] e2>0 & d4>0 -> (e2'=e2-1);
    [asst1] e2>0 & d5>0 -> (e2'=e2-1);
    [asst1] e2=0 | d5=0 -> true;

endmodule

module HoverAction3

    d3 : [0..max_d3] init max_d3; // duration

    [asst1] d3>0 & e2>0 -> (d3'=d3-1);
    [asst1] d3=0        -> true;

endmodule

module HoverAction4

    d4 : [0..max_d4] init max_d4; // duration

    [asst1] d3>0                       -> true;
    [asst1] d3=0 & d6=0 & d4>0 & e2>0 -> (d4'=d4-1);
    [asst1]               d4=0         -> true;

endmodule

module HoverAction5

    d5 : [0..max_d5] init max_d5; // duration

    [asst1] d4>0                -> true;
    [asst1] d4=0 & d5>0 & e2>0 -> (d5'=d5-1);
    [asst1]        d5=0         -> true;

endmodule

module Hummingbird3

    e3 : [0..max_e3] init max_e3; // endurance

    [asst1] e3>0 & d6>0 -> (e3'=e3-1);
    [asst1] e3=0 | d6=0 -> true;

endmodule

module HoverAction6

    d6 : [0..max_d6] init max_d6; // duration

    [asst1] d6>0 & e3>0 -> (d6'=d6-1);
    [asst1] d6=0        -> true;

endmodule
```

Listing F.51: PCTL output for Mission 2v.

```
P=? [ F d2=0 & d5=0 ]
```

Listing F.52: Log output for Mission 2v.

```
Model checking: P=? [ F d2=0&d5=0 ]

Building model...

Computing reachable states...
```

```
Reachability (BFS): 91 iterations in 0.07 seconds (average 0.000714, setup 0.00)

Time for model construction: 0.455 seconds.

Type:        DTMC
States:      91 (1 initial)
Transitions: 91

Transition matrix: 6332 nodes (2 terminal), 91 minterms, vars: 52r/52c

Prob0: 91 iterations in 0.06 seconds (average 0.000692, setup 0.00)

Prob1: 1 iterations in 0.00 seconds (average 0.000000, setup 0.00)

yes = 91, no = 0, maybe = 0

Value in the initial state: 1.0

Time for model checking: 0.064 seconds.

Result: 1.0 (value in the initial state)
```

Listing F.53: YAML input for Mission 2w.

```
Action:
  HoverAction:
    - id: 1
      duration: 60
      coordinates: [-112.1334152514208, 36.08362433106716]
    - id: 2
      duration: 30
      preconditions: [1, 4]
    - id: 3
      duration: 30
      coordinates: [-112.1199316051625, 36.07823185748754]
    - id: 4
      duration: 30
      preconditions: [3, 7]
    - id: 5
      duration: 30
      preconditions: [4]
    - id: 6
      duration: 30
      preconditions: [5]
    - id: 7
      duration: 30
      coordinates: [-112.1075818315498, 36.08838518690878]
Asset:
  Hummingbird:
    - id: 1
      actions: [1, 2]
    - id: 2
      actions: [3, 4, 5, 6]
    - id: 3
      actions: [7]
```

Listing F.54: DTMC output for Mission 2w.

```
dtmc

const int max_e1 = 120; // maximum endurance
const int max_d1 = 60;  // maximum duration
const int max_d2 = 30;  // maximum duration
const int max_e2 = 120; // maximum endurance
const int max_d3 = 30;  // maximum duration
const int max_d4 = 30;  // maximum duration
const int max_d5 = 30;  // maximum duration
const int max_d6 = 30;  // maximum duration
const int max_e3 = 120; // maximum endurance
const int max_d7 = 30;  // maximum duration
```

```
module Hummingbird1

    e1 : [0..max_e1] init max_e1; // endurance

    [asst1] e1>0 & d1>0 -> (e1'=e1-1);
    [asst1] e1>0 & d2>0 -> (e1'=e1-1);
    [asst1] e1=0 | d2=0 -> true;

endmodule

module HoverAction1

    d1 : [0..max_d1] init max_d1; // duration

    [asst1] d1>0 & e1>0 -> (d1'=d1-1);
    [asst1] d1=0        -> true;

endmodule

module HoverAction2

    d2 : [0..max_d2] init max_d2; // duration

    [asst1] d1>0                        -> true;
    [asst1] d1=0 & d4=0 & d2>0 & e1>0 -> (d2'=d2-1);
    [asst1]               d2=0        -> true;

endmodule

module Hummingbird2

    e2 : [0..max_e2] init max_e2; // endurance

    [asst1] e2>0 & d3>0 -> (e2'=e2-1);
    [asst1] e2>0 & d4>0 -> (e2'=e2-1);
    [asst1] e2>0 & d5>0 -> (e2'=e2-1);
    [asst1] e2>0 & d6>0 -> (e2'=e2-1);
    [asst1] e2=0 | d6=0 -> true;

endmodule

module HoverAction3

    d3 : [0..max_d3] init max_d3; // duration

    [asst1] d3>0 & e2>0 -> (d3'=d3-1);
    [asst1] d3=0        -> true;

endmodule

module HoverAction4

    d4 : [0..max_d4] init max_d4; // duration

    [asst1] d3>0                        -> true;
    [asst1] d3=0 & d7=0 & d4>0 & e2>0 -> (d4'=d4-1);
    [asst1]               d4=0        -> true;

endmodule

module HoverAction5

    d5 : [0..max_d5] init max_d5; // duration

    [asst1] d4>0                -> true;
    [asst1] d4=0 & d5>0 & e2>0 -> (d5'=d5-1);
    [asst1]        d5=0         -> true;

endmodule

module HoverAction6

    d6 : [0..max_d6] init max_d6; // duration
```

```
    [asst1] d5>0                -> true;
    [asst1] d5=0 & d6>0 & e2>0 -> (d6'=d6-1);
    [asst1]        d6=0         -> true;

endmodule

module Hummingbird3

    e3 : [0..max_e3] init max_e3; // endurance

    [asst1] e3>0 & d7>0 -> (e3'=e3-1);
    [asst1] e3=0 | d7=0 -> true;

endmodule

module HoverAction7

    d7 : [0..max_d7] init max_d7; // duration

    [asst1] d7>0 & e3>0 -> (d7'=d7-1);
    [asst1] d7=0        -> true;

endmodule
```

Listing F.55: PCTL output for Mission 2w.

```
P=? [ F d2=0 & d6=0 ]
```

Listing F.56: Log output for Mission 2w.

```
Model checking: P=? [ F d2=0&d6=0 ]

Building model...

Computing reachable states...

Reachability (BFS): 121 iterations in 0.11 seconds (average 0.000893, setup 0.00)

Time for model construction: 0.676 seconds.

Type:        DTMC
States:      121 (1 initial)
Transitions: 121

Transition matrix: 7789 nodes (2 terminal), 121 minterms, vars: 57r/57c

Prob0: 121 iterations in 0.09 seconds (average 0.000777, setup 0.00)

Prob1: 1 iterations in 0.00 seconds (average 0.000000, setup 0.00)

yes = 121, no = 0, maybe = 0

Value in the initial state: 1.0

Time for model checking: 0.095 seconds.

Result: 1.0 (value in the initial state)
```

Listing F.57: YAML input for Mission 2x.

```
Action:
  HoverAction:
    - id: 1
      duration: 60
      coordinates: [-112.1334152514208, 36.08362433106716]
    - id: 2
      duration: 30
      preconditions: [1, 4]
    - id: 3
```

```
      duration: 30
      coordinates: [-112.1199316051625, 36.07823185748754]
    - id: 4
      duration: 30
      preconditions: [3]
    - id: 5
      duration: 30
      preconditions: [4]
    - id: 6
      duration: 30
      preconditions: [5, 7]
    - id: 7
      duration: 90
      coordinates: [-112.1075818315498, 36.08838518690878]
Asset:
  Hummingbird:
    - id: 1
      actions: [1, 2]
    - id: 2
      actions: [3, 4, 5, 6]
    - id: 3
      actions: [7]
```

Listing F.58: DTMC output for Mission 2x.

```
dtmc

const int max_e1 = 120; // maximum endurance
const int max_d1 = 60;  // maximum duration
const int max_d2 = 30;  // maximum duration
const int max_e2 = 120; // maximum endurance
const int max_d3 = 30;  // maximum duration
const int max_d4 = 30;  // maximum duration
const int max_d5 = 30;  // maximum duration
const int max_d6 = 30;  // maximum duration
const int max_e3 = 120; // maximum endurance
const int max_d7 = 90;  // maximum duration

module Hummingbird1

    e1 : [0..max_e1] init max_e1; // endurance

    [asst1] e1>0 & d1>0 -> (e1'=e1-1);
    [asst1] e1>0 & d2>0 -> (e1'=e1-1);
    [asst1] e1=0 | d2=0 -> true;

endmodule

module HoverAction1

    d1 : [0..max_d1] init max_d1; // duration

    [asst1] d1>0 & e1>0 -> (d1'=d1-1);
    [asst1] d1=0        -> true;

endmodule

module HoverAction2

    d2 : [0..max_d2] init max_d2; // duration

    [asst1] d1>0                        -> true;
    [asst1] d1=0 & d4=0 & d2>0 & e1>0 -> (d2'=d2-1);
    [asst1]               d2=0          -> true;

endmodule

module Hummingbird2

    e2 : [0..max_e2] init max_e2; // endurance

    [asst1] e2>0 & d3>0 -> (e2'=e2-1);
```

```
    [asst1] e2>0 & d4>0 -> (e2'=e2-1);
    [asst1] e2>0 & d5>0 -> (e2'=e2-1);
    [asst1] e2>0 & d6>0 -> (e2'=e2-1);
    [asst1] e2=0 | d6=0 -> true;

endmodule

module HoverAction3

    d3 : [0..max_d3] init max_d3; // duration

    [asst1] d3>0 & e2>0 -> (d3'=d3-1);
    [asst1] d3=0        -> true;

endmodule

module HoverAction4

    d4 : [0..max_d4] init max_d4; // duration

    [asst1] d3>0               -> true;
    [asst1] d3=0 & d4>0 & e2>0 -> (d4'=d4-1);
    [asst1]        d4=0        -> true;

endmodule

module HoverAction5

    d5 : [0..max_d5] init max_d5; // duration

    [asst1] d4>0               -> true;
    [asst1] d4=0 & d5>0 & e2>0 -> (d5'=d5-1);
    [asst1]        d5=0        -> true;

endmodule

module HoverAction6

    d6 : [0..max_d6] init max_d6; // duration

    [asst1] d5>0                       -> true;
    [asst1] d5=0 & d7=0 & d6>0 & e2>0 -> (d6'=d6-1);
    [asst1]               d6=0         -> true;

endmodule

module Hummingbird3

    e3 : [0..max_e3] init max_e3; // endurance

    [asst1] e3>0 & d7>0 -> (e3'=e3-1);
    [asst1] e3=0 | d7=0 -> true;

endmodule

module HoverAction7

    d7 : [0..max_d7] init max_d7; // duration

    [asst1] d7>0 & e3>0 -> (d7'=d7-1);
    [asst1] d7=0        -> true;

endmodule
```

Listing F.59: PCTL output for Mission 2x.

```
P=? [ F d2=0 & d6=0 ]
```

Listing F.60: Log output for Mission 2x.

```
Model checking: P=? [ F d2=0&d6=0 ]
```

```
Building model...

Computing reachable states...

Reachability (BFS): 121 iterations in 0.11 seconds (average 0.000934, setup 0.00)

Time for model construction: 0.779 seconds.

Type:        DTMC
States:      121 (1 initial)
Transitions: 121

Transition matrix: 10188 nodes (2 terminal), 121 minterms, vars: 59r/59c

Prob0: 121 iterations in 0.11 seconds (average 0.000876, setup 0.00)

Prob1: 1 iterations in 0.00 seconds (average 0.001000, setup 0.00)

yes = 121, no = 0, maybe = 0

Value in the initial state: 1.0

Time for model checking: 0.11 seconds.

Result: 1.0 (value in the initial state)
```

Listing F.61: YAML input for Mission 3a.

```
Action:
  HoverAction:
    - id: 1
      duration: 69
      coordinates: [-112.1334152514208, 36.08362433106716]
  PhotoSurveillanceAction:
    - id: 2
      duration: 68
Asset:
  ARDrone:
    - id: 1
      actions: [1, 2]
```

Listing F.62: DTMC output for Mission 3a.

```
dtmc

const int max_e1 = 70;  // maximum endurance
const int max_d1 = 69;  // maximum duration
const int max_d2 = 68;  // maximum duration

module ARDrone1

    e1 : [0..max_e1] init max_e1; // endurance

    [actn1] e1>0 & d1>0 -> (e1'=e1-1);
    []      e1=0 | d1=0 -> true;

endmodule

module HoverAction1

    d1 : [0..max_d1] init max_d1; // duration

    [actn1] d1>0 & e1>0 -> (d1'=d1-1);

endmodule

module PhotoSurveillanceAction2

    d2 : [0..max_d2] init max_d2; // duration
```

```
    [actn1] d2>0 & e1>0 -> (d2'=d2-1);
    [actn1] d2=0        -> true;

endmodule
```

Listing F.63: PCTL output for Mission 3a.

```
P=? [ F d1=0 & d2=0 ]
```

Listing F.64: Log output for Mission 3a.

```
Model checking: P=? [ F d2=0&d3=0 ]

Building model...

Computing reachable states...

Reachability (BFS): 71 iterations in 0.02 seconds (average 0.000254, setup 0.00)

Time for model construction: 0.071 seconds.

Type:        DTMC
States:      71 (1 initial)
Transitions: 71

Transition matrix: 2282 nodes (2 terminal), 71 minterms, vars: 25r/25c

Prob0: 71 iterations in 0.02 seconds (average 0.000254, setup 0.00)

Prob1: 1 iterations in 0.00 seconds (average 0.001000, setup 0.00)

yes = 71, no = 0, maybe = 0

Value in the initial state: 1.0

Time for model checking: 0.019 seconds.

Result: 1.0 (value in the initial state)
```

Listing F.65: YAML input for Mission 3e.

```
Action:
  HoverAction:
    - id: 1
      duration: 40
      coordinates: [-112.1334152514208, 36.08362433106716]
    - id: 2
      duration: 30
      preconditions: [1]
  PhotoSurveillanceAction:
    - id: 3
      duration: 69
Asset:
  ARDrone:
    - id: 1
      actions: [1, 2, 3]
```

Listing F.66: DTMC output for Mission 3e.

```
dtmc

const int max_e1 = 70;  // maximum endurance
const int max_d1 = 40;  // maximum duration
const int max_d2 = 30;  // maximum duration
const int max_d3 = 69;  // maximum duration

module ARDrone1

    e1 : [0..max_e1] init max_e1; // endurance
```

```
    [actn1] e1>0 & d1>0 -> (e1'=e1-1);
    [actn2] e1>0 & d2>0 -> (e1'=e1-1);
    []      e1=0 | d2=0 -> true;

endmodule

module HoverAction1

    d1 : [0..max_d1] init max_d1; // duration

    [actn1] d1>0 & e1>0 -> (d1'=d1-1);

endmodule

module HoverAction2

    d2 : [0..max_d2] init max_d2; // duration

    [actn2] d1=0 & d2>0 & e1>0 -> (d2'=d2-1);

endmodule

module PhotoSurveillanceAction3

    d3 : [0..max_d3] init max_d3; // duration

    [actn1] d3>0 & e1>0 -> (d3'=d3-1);
    [actn1] d3=0        -> true;

    [actn2] d3>0 & e1>0 -> (d3'=d3-1);
    [actn2] d3=0        -> true;

endmodule
```

Listing F.67: PCTL output for Mission 3e.

```
P=? [ F d2=0 & d3=0 ]
```

Listing F.68: Log output for Mission 3e.

```
Model checking: P=? [ F d2=0&d3=0 ]

Building model...

Computing reachable states...

Reachability (BFS): 71 iterations in 0.02 seconds (average 0.000254, setup 0.00)

Time for model construction: 0.071 seconds.

Type:        DTMC
States:      71 (1 initial)
Transitions: 71

Transition matrix: 2282 nodes (2 terminal), 71 minterms, vars: 25r/25c

Prob0: 71 iterations in 0.02 seconds (average 0.000254, setup 0.00)

Prob1: 1 iterations in 0.00 seconds (average 0.001000, setup 0.00)

yes = 71, no = 0, maybe = 0

Value in the initial state: 1.0

Time for model checking: 0.019 seconds.

Result: 1.0 (value in the initial state)
```

Listing F.69: YAML input for Mission 3h.

```
Action:
  HoverAction:
    - id: 1
      duration: 40
      coordinates: [-112.1334152514208, 36.08362433106716]
    - id: 2
      duration: 30
      preconditions: [1]
  PhotoSurveillanceAction:
    - id: 3
      duration: 29
      preconditions: [1]
Asset:
  ARDrone:
    - id: 1
      actions: [1, 2, 3]
```

Listing F.70: DTMC output for Mission 3h.

```
dtmc

const int max_e1 = 70;  // maximum endurance
const int max_d1 = 40;  // maximum duration
const int max_d2 = 30;  // maximum duration
const int max_d3 = 29;  // maximum duration

module ARDrone1

    e1 : [0..max_e1] init max_e1; // endurance

    [actn1] e1>0 & d1>0 -> (e1'=e1-1);
    [actn2] e1>0 & d2>0 -> (e1'=e1-1);
    []      e1=0 | d2=0 -> true;

endmodule

module HoverAction1

    d1 : [0..max_d1] init max_d1; // duration

    [actn1] d1>0 & e1>0 -> (d1'=d1-1);

endmodule

module HoverAction2

    d2 : [0..max_d2] init max_d2; // duration

    [actn2] d1=0 & d2>0 & e1>0 -> (d2'=d2-1);

endmodule

module PhotoSurveillanceAction3

    d3 : [0..max_d3] init max_d3; // duration

    [actn2] d1=0 & d3>0 & e1>0 -> (d3'=d3-1);
    [actn2]        d3=0        -> true;

endmodule
```

Listing F.71: PCTL output for Mission 3h.

```
P=? [ F d2=0 & d3=0 ]
```

Listing F.72: Log output for Mission 3h.

```
Model checking: P=? [ F d2=0&d3=0 ]
```

```
Building model...

Computing reachable states...

Reachability (BFS): 71 iterations in 0.01 seconds (average 0.000127, setup 0.00)

Time for model construction: 0.054 seconds.

Type:        DTMC
States:      71 (1 initial)
Transitions: 71

Transition matrix: 1349 nodes (2 terminal), 71 minterms, vars: 23r/23c

Prob0: 71 iterations in 0.01 seconds (average 0.000183, setup 0.00)

Prob1: 1 iterations in 0.00 seconds (average 0.000000, setup 0.00)

yes = 71, no = 0, maybe = 0

Value in the initial state: 1.0

Time for model checking: 0.015 seconds.

Result: 1.0 (value in the initial state)
```

Listing F.73: YAML input for Mission 3k.

```
Action:
  HoverAction:
    - id: 1
      duration: 40
      coordinates: [-112.1334152514208, 36.08362433106716]
    - id: 2
      duration: 30
      preconditions: [1]
  PhotoSurveillanceAction:
    - id: 3
      duration: 39
    - id: 4
      duration: 29
      preconditions: [3]
Asset:
  ARDrone:
    - id: 1
      actions: [1, 2, 3, 4]
```

Listing F.74: DTMC output for Mission 3k.

```
dtmc

const int max_e1 = 70;  // maximum endurance
const int max_d1 = 40;  // maximum duration
const int max_d2 = 30;  // maximum duration
const int max_d3 = 39;  // maximum duration
const int max_d4 = 29;  // maximum duration

module ARDrone1

    e1 : [0..max_e1] init max_e1; // endurance

    [actn1] e1>0 & d1>0 -> (e1'=e1-1);
    [actn2] e1>0 & d2>0 -> (e1'=e1-1);
    []      e1=0 | d2=0 -> true;

endmodule

module HoverAction1

    d1 : [0..max_d1] init max_d1; // duration
```

```
    [actn1] d1>0 & e1>0 -> (d1'=d1-1);

endmodule

module HoverAction2

    d2 : [0..max_d2] init max_d2; // duration

    [actn2] d1=0 & d2>0 & e1>0 -> (d2'=d2-1);

endmodule

module PhotoSurveillanceAction3

    d3 : [0..max_d3] init max_d3; // duration

    [actn1] d3>0 & e1>0 -> (d3'=d3-1);
    [actn1] d3=0        -> true;

    [actn2] d3>0 & e1>0 -> (d3'=d3-1);
    [actn2] d3=0        -> true;

endmodule

module PhotoSurveillanceAction4

    d4 : [0..max_d4] init max_d4; // duration

    [actn1] d3>0               -> true;
    [actn1] d3=0 & d4>0 & e1>0 -> (d4'=d4-1);
    [actn1]        d4=0        -> true;

    [actn2] d3>0               -> true;
    [actn2] d3=0 & d4>0 & e1>0 -> (d4'=d4-1);
    [actn2]        d4=0        -> true;

endmodule
```

Listing F.75: PCTL output for Mission 3k.

```
P=? [ F d2=0 & d4=0 ]
```

Listing F.76: Log output for Mission 3k.

```
Model checking: P=? [ F d2=0&d4=0 ]

Building model...

Computing reachable states...

Reachability (BFS): 71 iterations in 0.02 seconds (average 0.000282, setup 0.00)

Time for model construction: 0.076 seconds.

Type:        DTMC
States:      71 (1 initial)
Transitions: 71

Transition matrix: 2577 nodes (2 terminal), 71 minterms, vars: 29r/29c

Prob0: 71 iterations in 0.02 seconds (average 0.000296, setup 0.00)

Prob1: 1 iterations in 0.00 seconds (average 0.000000, setup 0.00)

yes = 71, no = 0, maybe = 0

Value in the initial state: 1.0

Time for model checking: 0.021 seconds.

Result: 1.0 (value in the initial state)
```

Listing F.77: YAML input for Mission 3o.

```
Action:
  HoverAction:
    - id: 1
      duration: 40
      coordinates: [-112.1334152514208, 36.08362433106716]
    - id: 2
      duration: 30
      preconditions: [1]
  PhotoSurveillanceAction:
    - id: 3
      duration: 39
    - id: 4
      duration: 29
      preconditions: [1, 3]
Asset:
  ARDrone:
    - id: 1
      actions: [1, 2, 3, 4]
```

Listing F.78: DTMC output for Mission 3o.

```
dtmc

const int max_e1 = 70;  // maximum endurance
const int max_d1 = 40;  // maximum duration
const int max_d2 = 30;  // maximum duration
const int max_d3 = 39;  // maximum duration
const int max_d4 = 29;  // maximum duration

module ARDrone1

    e1 : [0..max_e1] init max_e1; // endurance

    [actn1] e1>0 & d1>0 -> (e1'=e1-1);
    [actn2] e1>0 & d2>0 -> (e1'=e1-1);
    []      e1=0 | d2=0 -> true;

endmodule

module HoverAction1

    d1 : [0..max_d1] init max_d1; // duration

    [actn1] d1>0 & e1>0 -> (d1'=d1-1);

endmodule

module HoverAction2

    d2 : [0..max_d2] init max_d2; // duration

    [actn2] d1=0 & d2>0 & e1>0 -> (d2'=d2-1);

endmodule

module PhotoSurveillanceAction3

    d3 : [0..max_d3] init max_d3; // duration

    [actn1] d3>0 & e1>0 -> (d3'=d3-1);
    [actn1] d3=0        -> true;

    [actn2] d3>0 & e1>0 -> (d3'=d3-1);
    [actn2] d3=0        -> true;

endmodule

module PhotoSurveillanceAction4

    d4 : [0..max_d4] init max_d4; // duration
```

```
    [actn1] d1>0 | d3>0                -> true;
    [actn1] d1=0 & d3=0 & d4>0 & e1>0 -> (d4'=d4-1);
    [actn1]               d4=0        -> true;

    [actn2] d1>0 | d3>0                -> true;
    [actn2] d1=0 & d3=0 & d4>0 & e1>0 -> (d4'=d4-1);
    [actn2]               d4=0        -> true;

endmodule
```

Listing F.79: PCTL output for Mission 3o.

```
P=? [ F d2=0 & d4=0 ]
```

Listing F.80: Log output for Mission 3o.

```
Model checking: P=? [ F d2=0&d4=0 ]

Building model...

Computing reachable states...

Reachability (BFS): 71 iterations in 0.02 seconds (average 0.000310, setup 0.00)

Time for model construction: 0.03 seconds.

Type:        DTMC
States:      71 (1 initial)
Transitions: 71

Transition matrix: 2588 nodes (2 terminal), 71 minterms, vars: 29r/29c

Prob0: 71 iterations in 0.02 seconds (average 0.000268, setup 0.00)

Prob1: 1 iterations in 0.00 seconds (average 0.000000, setup 0.00)

yes = 71, no = 0, maybe = 0

Value in the initial state: 1.0

Time for model checking: 0.02 seconds.

Result: 1.0 (value in the initial state)
```

Listing F.81: YAML input for Mission 4a.

```
Action:
  TraversePathSegmentAction:
    - id: 1
      duration: 30
      coordinates: [-112.1456898399667, 36.08223661335747, -112.1329057930546,
          36.09138166778199]
    - id: 2
      duration: 30
      coordinates: [-112.1136958212283, 36.08434312651666]
      preconditions: [1]
    - id: 4
      duration: 30
      coordinates: [-112.086330390576, 36.09066857566722]
      preconditions: [3]
  HoverAction:
    - id: 3
      duration: 30
      preconditions: [2]
Asset:
  Hummingbird:
    - id: 1
      actions: [1, 2, 3, 4]
```

Listing F.82: DTMC output for Mission 4a.

```
dtmc

const int max_e1 = 120; // maximum endurance
const int max_d1 = 30;  // maximum duration
const int max_d2 = 30;  // maximum duration
const int max_d3 = 30;  // maximum duration
const int max_d4 = 30;  // maximum duration

module Hummingbird1

    e1 : [0..max_e1] init max_e1; // endurance

    [actn1] e1>0 & d1>0 -> (e1'=e1-1);
    [actn2] e1>0 & d2>0 -> (e1'=e1-1);
    [actn3] e1>0 & d3>0 -> (e1'=e1-1);
    [actn4] e1>0 & d4>0 -> (e1'=e1-1);
    []      e1=0 | d4=0 -> true;

endmodule

module TraversePathSegmentAction1

    d1 : [0..max_d1] init max_d1; // duration

    [actn1] d1>0 & e1>0 -> (d1'=d1-1);

endmodule

module TraversePathSegmentAction2

    d2 : [0..max_d2] init max_d2; // duration

    [actn2] d1=0 & d2>0 & e1>0 -> (d2'=d2-1);

endmodule

module HoverAction3 = TraversePathSegmentAction2[d2=d3, max_d2=max_d3, actn2=
    actn3, d1=d2] endmodule
module TraversePathSegmentAction4 = HoverAction3[d3=d4, max_d3=max_d4, actn3=
    actn4, d2=d3] endmodule
```

Listing F.83: PCTL output for Mission 4a.

```
P=? [ F d4=0 ]
```

Listing F.84: Log output for Mission 4a.

```
Model checking: P=? [ F d4=0 ]

Building model...

Computing reachable states...

Reachability (BFS): 121 iterations in 0.02 seconds (average 0.000182, setup 0.00)

Time for model construction: 0.238 seconds.

Type:        DTMC
States:      121 (1 initial)
Transitions: 121

Transition matrix: 2880 nodes (2 terminal), 121 minterms, vars: 27r/27c

Prob0: 121 iterations in 0.03 seconds (average 0.000273, setup 0.00)

Prob1: 1 iterations in 0.00 seconds (average 0.000000, setup 0.00)

yes = 121, no = 0, maybe = 0

Value in the initial state: 1.0
```

```
Time for model checking: 0.034 seconds.

Result: 1.0 (value in the initial state)
```

Listing F.85: YAML input for Mission 4b.

```
Action:
  TraversePathSegmentAction:
    - id: 1
      duration: 30
      coordinates: [-112.1456898399667, 36.08223661335747, -112.1329057930546,
          36.09138166778199]
    - id: 2
      duration: 30
      coordinates: [-112.1136958212283, 36.08434312651666]
      preconditions: [1]
    - id: 4
      duration: 30
      coordinates: [-112.086330390576, 36.09066857566722]
      preconditions: [3]
  HoverAction:
    - id: 3
      duration: 30
      preconditions: [2]
Asset:
  Hummingbird:
    - id: 1
      actions: [1, 2, 3, 4]
```

Listing F.86: DTMC output for Mission 4b.

```
dtmc

const int max_e1 = 120; // maximum endurance
const int max_d1 = 30;  // maximum duration
const int max_d2 = 30;  // maximum duration
const int max_d3 = 30;  // maximum duration
const int max_d4 = 30;  // maximum duration

module Hummingbird1

    e1 : [0..max_e1] init max_e1; // endurance

    [actn1] e1>0 & d1>0 -> (e1'=e1-1);
    [actn2] e1>0 & d2>0 -> (e1'=e1-1);
    [actn3] e1>0 & d3>0 -> (e1'=e1-1);
    [actn4] e1>0 & d4>0 -> (e1'=e1-1);
    []      e1=0 | d4=0 -> true;

endmodule

module TraversePathSegmentAction1

    d1 : [0..max_d1] init max_d1; // duration

    [actn1] d1>0 & e1>0 -> (d1'=d1-1);

endmodule

module TraversePathSegmentAction2

    d2 : [0..max_d2] init max_d2; // duration

    [actn2] d1=0 & d2>0 & e1>0 -> (d2'=d2-1);

endmodule

module HoverAction3 = TraversePathSegmentAction2[d2=d3, max_d2=max_d3, actn2=
    actn3, d1=d2] endmodule
module TraversePathSegmentAction4 = HoverAction3[d3=d4, max_d3=max_d4, actn3=
```

```
    actn4, d2=d3] endmodule

//

const int start2 = 15; // threat area incursion start for
    TraversePathSegmentAction2
const int start3 = 30; // threat area incursion start for HoverAction1
const int start4 = 30; // threat area incursion start for
    TraversePathSegmentAction3
const int finish2 = 0; // threat area incursion end for
    TraversePathSegmentAction2
const int finish3 = 0; // threat area incursion end for HoverAction1
const int finish4 = 5; // threat area incursion end for
    TraversePathSegmentAction3
formula actn2_tai = d2>finish2 & d2<=start2; // threat area incursion during
    TraversePathSegmentAction2
formula actn3_tai = d3>finish3 & d3<=start3; // threat area incursion during
    HoverAction1
formula actn4_tai = d4>finish4 & d4<=start4; // threat area incursion during
    TraversePathSegmentAction3

module Hummingbird1_Survivability

    a1d : bool init false; // Hummingbird 1 destroyed

    [actn2] !a1d &  actn2_tai -> 0.99:(a1d'=false) + 0.01:(a1d'=true);
    [actn2]  a1d | !actn2_tai -> true;

    [actn3] !a1d &  actn3_tai -> 0.99:(a1d'=false) + 0.01:(a1d'=true);
    [actn3]  a1d | !actn3_tai -> true;

    [actn4] !a1d &  actn4_tai -> 0.99:(a1d'=false) + 0.01:(a1d'=true);
    [actn4]  a1d | !actn4_tai -> true;

endmodule

formula duration2 = start2 - finish2;
formula duration3 = start3 - finish3;
formula duration4 = start4 - finish4;

formula tkad1 = duration2 + duration3 + duration4; // total threat area action
    durations
formula raf1 = 0 / tkad1; // risk acceptability factor
```

Listing F.87: PCTL output for Mission 4b.

```
P=? [ F d4=0 & !a1d & raf1>0.6 ]
```

Listing F.88: Log output for Mission 4b.

```
Model checking: P=? [ F d4=0&!a1d&raf1>0.6 ]

Building model...

Computing reachable states...

Reachability (BFS): 121 iterations in 0.03 seconds (average 0.000207, setup 0.00)

Time for model construction: 0.225 seconds.

Type:        DTMC
States:      196 (1 initial)
Transitions: 266

Transition matrix: 3024 nodes (4 terminal), 266 minterms, vars: 28r/28c

yes = 0, no = 196, maybe = 0

Value in the initial state: 0.0

Time for model checking: 0.0070 seconds.
```

```
Result: 0.0 (value in the initial state)
```

Listing F.89: YAML input for Mission 4d.

```
Action:
  TraversePathSegmentAction:
    - id: 1
      duration: 30
      coordinates: [-112.1456898399667, 36.08223661335747, -112.1329057930546,
          36.09138166778199]
    - id: 2
      duration: 30
      coordinates: [-112.1136958212283, 36.08434312651666]
      preconditions: [1]
    - id: 4
      duration: 30
      coordinates: [-112.086330390576, 36.09066857566722]
      preconditions: [3]
  HoverAction:
    - id: 3
      duration: 30
      preconditions: [2]
  PhotoSurveillanceAction:
    - id: 5
      duration: 43
      preconditions: [2]
Asset:
  Hummingbird:
    - id: 1
      actions: [1, 2, 3, 4, 5]
```

Listing F.90: DTMC output for Mission 4d.

```
dtmc

const int max_e1 = 120; // maximum endurance
const int max_d1 = 30;  // maximum duration
const int max_d2 = 30;  // maximum duration
const int max_d3 = 30;  // maximum duration
const int max_d4 = 30;  // maximum duration
const int max_d5 = 43;  // maximum duration

module Hummingbird1

    e1 : [0..max_e1] init max_e1; // endurance

    [actn1] e1>0 & d1>0 -> (e1'=e1-1);
    [actn2] e1>0 & d2>0 -> (e1'=e1-1);
    [actn3] e1>0 & d3>0 -> (e1'=e1-1);
    [actn4] e1>0 & d4>0 -> (e1'=e1-1);
    []      e1=0 | d4=0 -> true;

endmodule

module TraversePathSegmentAction1

    d1 : [0..max_d1] init max_d1; // duration

    [actn1] d1>0 & e1>0 -> (d1'=d1-1);

endmodule

module TraversePathSegmentAction2

    d2 : [0..max_d2] init max_d2; // duration

    [actn2] d1=0 & d2>0 & e1>0 -> (d2'=d2-1);

endmodule
```

```
module HoverAction3 = TraversePathSegmentAction2[d2=d3, max_d2=max_d3, actn2=
    actn3, d1=d2] endmodule
module TraversePathSegmentAction4 = HoverAction3[d3=d4, max_d3=max_d4, actn3=
    actn4, d2=d3] endmodule

module PhotoSurveillanceAction5

    d5 : [0..max_d5] init max_d5; // duration
    r5 : bool init false;         // running

    [actn3] d2=0 & d5>0 & e1>0 -> (d5'=d5-1)&(r5'=true);
    [actn3]        d5=0        -> (r5'=false);

    [actn4] d2=0 & d5>0 & e1>0 -> (d5'=d5-1)&(r5'=true);
    [actn4]        d5=0        -> (r5'=false);

endmodule

//

const int start2 = 15; // threat area incursion start for
    TraversePathSegmentAction2
const int start3 = 30; // threat area incursion start for HoverAction1
const int start4 = 30; // threat area incursion start for
    TraversePathSegmentAction3
const int finish2 = 0; // threat area incursion end for
    TraversePathSegmentAction2
const int finish3 = 0; // threat area incursion end for HoverAction1
const int finish4 = 5; // threat area incursion end for
    TraversePathSegmentAction3
formula actn2_tai = d2>finish2 & d2<=start2; // threat area incursion during
    TraversePathSegmentAction2
formula actn3_tai = d3>finish3 & d3<=start3; // threat area incursion during
    HoverAction1
formula actn4_tai = d4>finish4 & d4<=start4; // threat area incursion during
    TraversePathSegmentAction3

module Hummingbird1_Survivability

    a1d : bool init false; // Hummingbird 1 destroyed

    [actn2] !a1d &  actn2_tai -> 0.99:(a1d'=false) + 0.01:(a1d'=true);
    [actn2]  a1d | !actn2_tai -> true;

    [actn3] !a1d &  actn3_tai -> 0.99:(a1d'=false) + 0.01:(a1d'=true);
    [actn3]  a1d | !actn3_tai -> true;

    [actn4] !a1d &  actn4_tai -> 0.99:(a1d'=false) + 0.01:(a1d'=true);
    [actn4]  a1d | !actn4_tai -> true;

endmodule

formula duration2 = start2 - finish2;
formula duration3 = start3 - finish3;
formula duration4 = start4 - finish4;

formula tkad1 = duration2 + duration3 + duration4; // total threat area action
    durations

module SensorActionCounter1

    sad1 : [0..tkad1] init 0; // sensor action duration

    [actn3]  actn3_tai &  r5 & sad1<tkad1 -> (sad1'=sad1+1);
    [actn3] !actn3_tai | !r5              -> true;

    [actn4]  actn4_tai &  r5 & sad1<tkad1 -> (sad1'=sad1+1);
    [actn4] !actn4_tai | !r5              -> true;

endmodule

formula raf1 = sad1 / tkad1; // risk acceptability factor
```

Listing F.91: PCTL output for Mission 4d.

```
P=? [ F d4=0 & !a1d & raf1>0.6 ]
```

Listing F.92: Log output for Mission 4d.

```
Model checking: P=? [ F d4=0&!a1d&raf1>0.6 ]

Building model...

Computing reachable states...

Reachability (BFS): 121 iterations in 0.04 seconds (average 0.000355, setup 0.00)

Time for model construction: 0.35 seconds.

Type:        DTMC
States:      196 (1 initial)
Transitions: 266

Transition matrix: 5393 nodes (4 terminal), 266 minterms, vars: 42r/42c

Prob0: 121 iterations in 0.09 seconds (average 0.000702, setup 0.00)

Prob1: 47 iterations in 0.04 seconds (average 0.000872, setup 0.00)

yes = 6, no = 75, maybe = 115

Computing remaining probabilities...
Engine: Hybrid

Building hybrid MTBDD matrix... [levels=42, nodes=4775] [111.9 KB]
Adding explicit sparse matrices... [levels=42, num=1, compact] [0.9 KB]
Creating vector for diagonals... [dist=1, compact] [0.4 KB]
Creating vector for RHS... [dist=2, compact] [0.4 KB]
Allocating iteration vectors... [2 x 1.5 KB]
TOTAL: [116.7 KB]

Starting iterations...

Jacobi: 116 iterations in 0.01 seconds (average 0.000009, setup 0.01)

Value in the initial state: 0.49483865960020695

Time for model checking: 0.158 seconds.

Result: 0.49483865960020695 (value in the initial state)
```

Listing F.93: YAML input for Mission 4f.

```
Action:
  TraversePathSegmentAction:
    - id: 1
      duration: 30
      coordinates: [-112.1456898399667, 36.08223661335747, -112.1329057930546,
          36.091381667781 99]
    - id: 2
      duration: 30
      coordinates: [-112.1136958212283, 36.08434312651666]
      preconditions: [1]
    - id: 4
      duration: 30
      coordinates: [-112.086330390576, 36.09066857566722]
      preconditions: [3]
  HoverAction:
    - id: 3
      duration: 30
      preconditions: [2]
  PhotoSurveillanceAction:
    - id: 5
      duration: 35
      preconditions: [2]
```

```
    - id: 6
      duration: 13
      preconditions: [3]
Asset:
  Hummingbird:
    - id: 1
      actions: [1, 2, 3, 4, 5, 6]
```

Listing F.94: DTMC output for Mission 4f.

```
dtmc

const int max_e1 = 120; // maximum endurance
const int max_d1 = 30;  // maximum duration
const int max_d2 = 30;  // maximum duration
const int max_d3 = 30;  // maximum duration
const int max_d4 = 30;  // maximum duration
const int max_d5 = 35;  // maximum duration
const int max_d6 = 13;  // maximum duration

module Hummingbird1

    e1 : [0..max_e1] init max_e1; // endurance

    [actn1] e1>0 & d1>0 -> (e1'=e1-1);
    [actn2] e1>0 & d2>0 -> (e1'=e1-1);
    [actn3] e1>0 & d3>0 -> (e1'=e1-1);
    [actn4] e1>0 & d4>0 -> (e1'=e1-1);
    []      e1=0 | d4=0 -> true;

endmodule

module TraversePathSegmentAction1

    d1 : [0..max_d1] init max_d1; // duration

    [actn1] d1>0 & e1>0 -> (d1'=d1-1);

endmodule

module TraversePathSegmentAction2

    d2 : [0..max_d2] init max_d2; // duration

    [actn2] d1=0 & d2>0 & e1>0 -> (d2'=d2-1);

endmodule

module HoverAction3 = TraversePathSegmentAction2[d2=d3, max_d2=max_d3, actn2=
    actn3, d1=d2] endmodule
module TraversePathSegmentAction4 = HoverAction3[d3=d4, max_d3=max_d4, actn3=
    actn4, d2=d3] endmodule

module PhotoSurveillanceAction5

    d5 : [0..max_d5] init max_d5; // duration
    r5 : bool init false;         // running

    [actn3] d2=0 & d5>0 & e1>0 -> (d5'=d5-1)&(r5'=true);
    [actn3]        d5=0        -> (r5'=false);

    [actn4] d2=0 & d5>0 & e1>0 -> (d5'=d5-1)&(r5'=true);
    [actn4]        d5=0        -> (r5'=false);

endmodule

module PhotoSurveillanceAction6

    d6 : [0..max_d6] init max_d6; // duration
    r6 : bool init false;         // running

    [actn4] d3=0 & d6>0 & e1>0 -> (d6'=d6-1)&(r6'=true);
```

```
    [actn4]        d6=0        -> (r6'=false);

endmodule

//

const int start2 = 15; // threat area incursion start for
    TraversePathSegmentAction2
const int start3 = 30; // threat area incursion start for HoverAction1
const int start4 = 30; // threat area incursion start for
    TraversePathSegmentAction3
const int finish2 = 0; // threat area incursion end for
    TraversePathSegmentAction2
const int finish3 = 0; // threat area incursion end for HoverAction1
const int finish4 = 5; // threat area incursion end for
    TraversePathSegmentAction3
formula actn2_tai = d2>finish2 & d2<=start2; // threat area incursion during
    TraversePathSegmentAction2
formula actn3_tai = d3>finish3 & d3<=start3; // threat area incursion during
    HoverAction1
formula actn4_tai = d4>finish4 & d4<=start4; // threat area incursion during
    TraversePathSegmentAction3

module Hummingbird1_Survivability

    a1d : bool init false; // Hummingbird 1 destroyed

    [actn2] !a1d &  actn2_tai -> 0.99:(a1d'=false) + 0.01:(a1d'=true);
    [actn2]  a1d | !actn2_tai -> true;

    [actn3] !a1d &  actn3_tai -> 0.99:(a1d'=false) + 0.01:(a1d'=true);
    [actn3]  a1d | !actn3_tai -> true;

    [actn4] !a1d &  actn4_tai -> 0.99:(a1d'=false) + 0.01:(a1d'=true);
    [actn4]  a1d | !actn4_tai -> true;

endmodule

formula duration2 = start2 - finish2;
formula duration3 = start3 - finish3;
formula duration4 = start4 - finish4;

formula tkad1 = duration2 + duration3 + duration4; // total threat area action
    durations

module SensorActionCounter1

    sad1 : [0..tkad1] init 0; // sensor action duration

    [actn3]  actn3_tai &  r5 & sad1<tkad1 -> (sad1'=sad1+1);
    [actn3] !actn3_tai | !r5              -> true;

    [actn4]  actn4_tai &  (r5 | r6) & sad1<tkad1 -> (sad1'=sad1+1);
    [actn4] !actn4_tai | !(r5 | r6)              -> true;

endmodule

formula raf1 = sad1 / tkad1; // risk acceptability factor
```

Listing F.95: PCTL output for Mission 4f.

```
P=? [ F d4=0 & !a1d & raf1>0.6 ]
```

Listing F.96: Log output for Mission 4f.

```
Model checking: P=? [ F d4=0&!a1d&raf1>0.6 ]

Building model...

Computing reachable states...
```

```
Reachability (BFS): 121 iterations in 0.04 seconds (average 0.000364, setup 0.00)

Time for model construction: 0.383 seconds.

Type:        DTMC
States:      196 (1 initial)
Transitions: 266

Transition matrix: 5873 nodes (4 terminal), 266 minterms, vars: 47r/47c

Prob0: 121 iterations in 0.09 seconds (average 0.000744, setup 0.00)

Prob1: 47 iterations in 0.04 seconds (average 0.000894, setup 0.00)

yes = 6, no = 75, maybe = 115

Computing remaining probabilities...
Engine: Hybrid

Building hybrid MTBDD matrix... [levels=47, nodes=5245] [122.9 KB]
Adding explicit sparse matrices... [levels=47, num=1, compact] [0.9 KB]
Creating vector for diagonals... [dist=1, compact] [0.4 KB]
Creating vector for RHS... [dist=2, compact] [0.4 KB]
Allocating iteration vectors... [2 x 1.5 KB]
TOTAL: [127.7 KB]

Starting iterations...

Jacobi: 116 iterations in 0.01 seconds (average 0.000009, setup 0.01)

Value in the initial state: 0.49483865960020695

Time for model checking: 0.158 seconds.

Result: 0.49483865960020695 (value in the initial state)
```

Listing F.97: YAML input for Mission 5a.

```
Action:
  HoverAction:
    - id: 1
      duration: 30
    - id: 3
      duration: 30
      preconditions: [2]
  TraversePathSegmentAction:
    - id: 2
      duration: 30
      coordinates: [-118.2738988072612, 34.03893526262756, -118.2710020707466,
          34.03699573489515]
      preconditions: [1]
  LidarAction:
    - id: 4
      interval: 5
      concurrencies: [1]
    - id: 5
      interval: 5
      concurrencies: [3]
Asset:
  Hummingbird:
    - id: 1
      actions: [1, 2, 3]
```

Listing F.98: DTMC output for Mission 5a.

```
dtmc

const int max_e1 = 120; // maximum endurance
const int max_d1 = 30;  // maximum duration
const int max_d2 = 30;  // maximum duration
const int max_d3 = 30;  // maximum duration
```

```
module Hummingbird1

    e1 : [0..max_e1] init max_e1; // endurance

    [actn1] e1>0 & d1>0 -> (e1'=e1-1);
    [actn2] e1>0 & d2>0 -> (e1'=e1-1);
    [actn3] e1>0 & d3>0 -> (e1'=e1-1);
    []      e1=0 | d3=0 -> true;

endmodule

module HoverAction1

    d1 : [0..max_d1] init max_d1; // duration

    [actn1] d1>0 & e1>0 -> (d1'=d1-1);

endmodule

module TraversePathSegmentAction2

    d2 : [0..max_d2] init max_d2; // duration

    [actn2] d1=0 & d2>0 & e1>0 -> (d2'=d2-1);

endmodule

module HoverAction3

    d3 : [0..max_d3] init max_d3; // duration

    [actn3] d2=0 & d3>0 & e1>0 -> (d3'=d3-1);

endmodule

//

const int i4 = 5; // interval
const int i5 = 5; // interval

module LidarAction4

    r4 : [0..max_d1] init 0; // readings

    [actn1]  mod(d1, i4)=0 & r4<max_d1 -> (r4'=r4+1);
    [actn1] !mod(d1, i4)=0             -> true;

endmodule

module LidarAction5 = LidarAction4[r4=r5, max_d1=max_d3, actn1=actn3, d1=d3, i4=
    i5] endmodule
```

Listing F.99: PCTL output for Mission 5a.

```
P=? [ F d3=0 ]
```

Listing F.100: Log output for Mission 5a.

```
Model checking: P=? [ F d3=0 ]

Building model...

Computing reachable states...

Reachability (BFS): 91 iterations in 0.02 seconds (average 0.000253, setup 0.00)

Time for model construction: 0.209 seconds.

Type:        DTMC
States:      91 (1 initial)
```

```
Transitions: 91

Transition matrix: 2570 nodes (2 terminal), 91 minterms, vars: 32r/32c

Prob0: 91 iterations in 0.02 seconds (average 0.000242, setup 0.00)

Prob1: 1 iterations in 0.00 seconds (average 0.000000, setup 0.00)

yes = 91, no = 0, maybe = 0

Value in the initial state: 1.0

Time for model checking: 0.023 seconds.

Result: 1.0 (value in the initial state)
```

Listing F.101: YAML input for Mission 5b.

```
Action:
  HoverAction:
    - id: 1
      duration: 30
    - id: 3
      duration: 30
      preconditions: [2]
  TraversePathSegmentAction:
    - id: 2
      duration: 30
      coordinates: [-118.2738988072612, 34.03893526262756, -118.2710020707466,
          34.03699573489515]
      preconditions: [1]
  LidarAction:
    - id: 4
      interval: 5
      concurrencies: [1]
    - id: 5
      interval: 5
      concurrencies: [3]
Asset:
  Hummingbird:
    - id: 1
      actions: [1, 2, 3]
```

Listing F.102: DTMC output for Mission 5b.

```
dtmc

const int max_e1 = 120; // maximum endurance
const int max_d1 = 30;  // maximum duration
const int max_d2 = 30;  // maximum duration
const int max_d3 = 30;  // maximum duration

module Hummingbird1

    e1 : [0..max_e1] init max_e1; // endurance

    [actn1] e1>0 & d1>0 -> (e1'=e1-1);
    [actn2] e1>0 & d2>0 -> (e1'=e1-1);
    [actn3] e1>0 & d3>0 -> (e1'=e1-1);
    []      e1=0 | d3=0 -> true;

endmodule

module HoverAction1

    d1 : [0..max_d1] init max_d1; // duration

    [actn1] d1>0 & e1>0 -> (d1'=d1-1);

endmodule
```

```
module TraversePathSegmentAction2

    d2 : [0..max_d2] init max_d2; // duration

    [actn2] d1=0 & d2>0 & e1>0 -> (d2'=d2-1);

endmodule

module HoverAction3

    d3 : [0..max_d3] init max_d3; // duration

    [actn3] d2=0 & d3>0 & e1>0 -> (d3'=d3-1);

endmodule

//

const int i4 = 5;                    // interval
const int icf4 = 2;                  // interval calibration factor
formula ci4 = nse4 ? i4 * icf4 : i4; // current interval

module LidarAction4

    r4 : [0..max_d1] init 0; // readings
    nse4 : bool init false;  // nominal speed exceeded

    [actn1]  mod(d1, ci4)=0 & r4<max_d1 &  spd1=2 -> (r4'=r4+1)&(nse4'=true);
    [actn1]  mod(d1, ci4)=0 & r4<max_d1 & !spd1=2 -> (r4'=r4+1)&(nse4'=false);
    [actn1] !mod(d1, ci4)=0                       -> true;

endmodule

module FreewaySection1 // low speed freeway section

    // 0 = minimum speed
    // 1 = nominal speed
    // 2 = nominal speed exceeded
    spd1 : [0..2] init 1;

    [actn1] spd1=0 -> 0.3:(spd1'=0) + 0.7:(spd1'=1);
    [actn1] spd1=1 -> 0.3:(spd1'=0) + 0.3:(spd1'=1) + 0.4:(spd1'=2);
    [actn1] spd1=2 -> 0.6:(spd1'=1) + 0.4:(spd1'=2);

endmodule

//

const int i5 = 5;                    // interval
const int icf5 = 3;                  // interval calibration factor
formula ci5 = nse5 ? i5 * icf5 : i5; // current interval

module LidarAction5

    r5 : [0..max_d3] init 0; // readings
    nse5 : bool init false;  // nominal speed exceeded

    [actn3]  mod(d3, ci5)=0 & r5<max_d3 &  spd2=2 -> (r5'=r5+1)&(nse5'=true);
    [actn3]  mod(d3, ci5)=0 & r5<max_d3 & !spd2=2 -> (r5'=r5+1)&(nse5'=false);
    [actn3] !mod(d3, ci5)=0                       -> true;

endmodule

module FreewaySection2 // high speed freeway section

    // 0 = minimum speed
    // 1 = nominal speed
    // 2 = nominal speed exceeded
    spd2 : [0..2] init 1;

    [actn3] spd2=0 -> 0.1:(spd2'=0) + 0.9:(spd2'=1);
    [actn3] spd2=1 -> 0.2:(spd2'=0) + 0.3:(spd2'=1) + 0.5:(spd2'=2);
    [actn3] spd2=2 -> 0.5:(spd2'=1) + 0.5:(spd2'=2);
```

```
endmodule
```

Listing F.103: PCTL output for Mission 5b.

```
P=? [ F d3=0 ]
P=? [ F nse4 & spd1=0 & !mod(d1, ci4)=0 ]
P=? [ F nse5 & spd2=0 & !mod(d3, ci5)=0 ]
```

Listing F.104: Log output for Mission 5b.

```
Model checking: P=? [ F d3=0 ]

Building model...

Computing reachable states...

Reachability (BFS): 91 iterations in 0.02 seconds (average 0.000242, setup 0.00)

Time for model construction: 0.259 seconds.

Type:        DTMC
States:      5335 (1 initial)
Transitions: 11531

Transition matrix: 3381 nodes (10 terminal), 11531 minterms, vars: 38r/38c

Prob0: 91 iterations in 0.03 seconds (average 0.000308, setup 0.00)

Prob1: 1 iterations in 0.00 seconds (average 0.000000, setup 0.00)

yes = 5335, no = 0, maybe = 0

Value in the initial state: 1.0

Time for model checking: 0.028 seconds.

Result: 1.0 (value in the initial state)



Model checking: P=? [ F nse4&spd1=0&!mod(d1,ci4)=0 ]

Prob0: 9 iterations in 0.01 seconds (average 0.000667, setup 0.00)

Prob1: 25 iterations in 0.01 seconds (average 0.000520, setup 0.00)

yes = 33, no = 5139, maybe = 163

Computing remaining probabilities...
Engine: Hybrid

Building hybrid MTBDD matrix... [levels=38, nodes=1135] [53.2 KB]
Adding explicit sparse matrices... [levels=38, num=1, compact] [6.8 KB]
Creating vector for diagonals... [dist=1, compact] [10.4 KB]
Creating vector for RHS... [dist=2, compact] [10.4 KB]
Allocating iteration vectors... [2 x 41.7 KB]
TOTAL: [164.2 KB]

Starting iterations...

Jacobi: 30 iterations in 0.00 seconds (average 0.000067, setup 0.00)

Value in the initial state: 0.6250983112904365

Time for model checking: 0.048 seconds.

Result: 0.6250983112904365 (value in the initial state)
```

```
Model checking: P=? [ F nse5&spd2=0&!mod(d3,ci5)=0 ]

Prob0: 68 iterations in 0.04 seconds (average 0.000515, setup 0.00)

Prob1: 85 iterations in 0.04 seconds (average 0.000447, setup 0.00)

yes = 720, no = 900, maybe = 3715

Computing remaining probabilities...
Engine: Hybrid

Building hybrid MTBDD matrix... [levels=38, nodes=3464] [162.4 KB]
Adding explicit sparse matrices... [levels=38, num=1, compact] [37.9 KB]
Creating vector for diagonals... [dist=1, compact] [10.4 KB]
Creating vector for RHS... [dist=2, compact] [10.4 KB]
Allocating iteration vectors... [2 x 41.7 KB]
TOTAL: [304.5 KB]

Starting iterations...

Jacobi: 90 iterations in 0.01 seconds (average 0.000089, setup 0.01)

Value in the initial state: 0.6507381247299331

Time for model checking: 0.115 seconds.

Result: 0.6507381247299331 (value in the initial state)
```