Verifying autonomous decision making against environment assumptions: An experience report

Hoang Tung Dinh
KU Leuven
Dept. of Computer Science, imec-DistriNet
B-3001 Leuven, Belgium
Email: hoangtung.dinh@cs.kuleuven.be

Tom Holvoet
KU Leuven
Dept. of Computer Science, imec-DistriNet
B-3001 Leuven, Belgium
Email: tom.holvoet@cs.kuleuven.be

***Abstract*—Discrete decision making is a crucial software component of autonomous systems. Since many autonomous systems are safety-critical, it is important to have their decision making formally verified. Model checking is a well-known technique in computer science that can automatically verify the correctness of a system. In this paper we report our experience on applying different model checkers, including ProB, SPIN, TLC, Alloy and NuSMV, on verifying the discrete decision making of an autonomous UAV in an industrial application: pylon inspection. We study how the decision making logic of the UAV and the assumptions on its operating environment can be represented in each model checker and conduct a performance evaluation. The results demonstrate that only model checkers based on bounded model checking and symbolic model checking, that is, Alloy and NuSMV, are able to verify the decision making of the UAV in our case study.**

I. Introduction

In an autonomous system, discrete decision making concerns performing symbolic reasoning to decide which high-level actions the system should execute, based on semantic information received from perception components. Each high-level action then executes motion planning and control algorithms to bring the system to its desired states. Examples of high-level actions of an autonomous UAV system are *take-off* and *fly-to-home*. Since autonomous systems operate in an open environment, they need to deal with a huge number of possible situations, leading to the high complexity in their decision making. Figure 1 shows the position of the decision making component in a typical software stack of autonomous systems [1], [2].

Since many autonomous systems are safety-critical, it is important to have their decision making formally verified. Model checking is a computer science technique that can automatically verify the correctness of a system. Given a specification and a desired property, model checking automatically explores all possible execution traces of the specified system to check if the property always holds. If the property can be falsified, model checking returns a counter-example demonstrating an execution trace that falsifies the property. The counter-example is valuable information for one to understand and improve the correctness of the system.

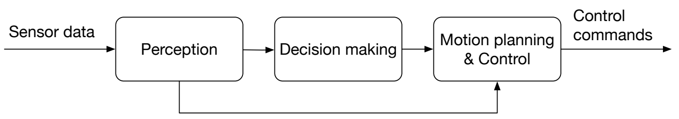

Fig. 1. The typical software stack for an autonomous system.

Existing works [3], [4], [5], [6] have shown a great potential of using model checking for verifying autonomous decision making. However, model checkers differ greatly in both their specification languages and computation techniques. It requires extensive expertise and effort to use such model checkers for a specific application. Applications of model checkers are therefore tailored and implemented in a specific way to match the respective specification languages and computation techniques.

Given the variety of model checkers available, it is difficult to select a suitable model checker for any other specific application. Yet, selecting the right model checker can improve verification results significantly. For example, in [7], the SPIN model checker was employed to verify the decision making of a domestic home care robot. Later, in a follow-up paper [8], the authors showed that by using the NuSMV model checker and carefully hand-crafting the NuSMV specification, they can take into account more sophisticated information while being able to verify more properties (some properties cannot be verified by SPIN) at a shorter computation time. To the best of our knowledge, there exists no work on comparing the usability of different model checkers for autonomous decision making.

In this paper we report our experience on applying different model checkers in a case study of autonomous UAVs for pylon inspection and provide insights into the modeling and performance aspects of each model checker. The decision making in the case study was developed within the scope of the SafeDroneWare project[1] and has been deployed in a real UAV platform, the DJI Matrice 100[2]. We select five different model checkers with different underlaying model checking techniques, including ProB [9], SPIN [10], TLC [11] , Alloy [12] and NuSMV [13]. Since there is a single UAV and quantitative properties are not considered in the case study,

[1]https://www.imec-int.com/en/what-we-offer/research-portfolio/safedroneware
[2]https://www.dji.com/be/matrice100

multi-agent model checkers and probabilistic model checkers such as MCMAS [14] and PRISM [15] are not considered in this paper.

Since the UAV operates in an open, non-deterministic and unstructured environment, it is not possible to create a high-fidelity and formal environment model for model checking. Instead, we aim at verifying desired properties against a set of logical assumptions on the behavior of the environment [4]. The set of assumptions, as a result, represent the interaction between the UAV system and the environment. Specifying the environment as a set of assumptions is also advantageous in that they explicitly represent the boundary conditions in which the UAV can guarantee the correctness of its decision making.

Concretely, our contribution is twofold. First, we study how the discrete decision making, desired properties and assumptions of the UAV in a pylon inspection case study can be modeled in each model checker. Second, we evaluate the performance of each model checker on verifying the UAV's discrete decision making. The evaluation results show that only Alloy and NuSMV can verify all properties in our case study.

This paper is organized as follows. Section II discusses related work. Section III presents the formal definition of discrete decision making policies and the case study. Section IV describes how the verification problem can be formulated in each model checker. Section V presents the performance evaluation of each model checker. Finally, Section VI draws conclusions and outlines future work.

II. Related work

Model checking is a widely used formal method. Model checking can automatically search and verify whether there is a possible execution trace of the system violating the desired properties. There has been a great interest in applying formal methods in general and model checking techniques in particular for guaranteeing the safety of autonomous systems [16].

A large body of work in the literature focuses on verifying autonomous decision making implemented in agent programming languages. In [7], [3], SPIN is used to verify the behavior of a domestic home care robot. The decision making logic of the robot is first modeled in Brahms, a language to model rational agent, and then is translated to PROMELA, SPIN's specification language, for verification. Dennis et al. [4], [17] use the AJPF model checker [18] to verify decision making logic implemented in the agent programming language GWENDOLEN.

Both SPIN and AJPF are also used to verify a UAV control system [19]. Choi et al. [5] use MCMAS, a model checker for multi-agent systems, to verify different heterogeneous multi-agent systems. The authors in [8], [6] model-check the behavior of a domestic home care robot by translating its behavior to an intermediate form. From the intermediate form, NuSMV models are constructed automatically and verified.

So far, significant progress has been made in demonstrating how model checking can be applied to verify autonomous decision making. However, as remarked by many authors, a common limitation when applying model checking is its computational complexity. In addition, the selection of model checkers is often based on personal experience. As model checkers differ greatly from their specification languages to the verification algorithms, it is not obvious which model checkers are suitable to a particular problem.

III. Case study

A. Decision making policy

In the case study, the UAV employs a decision making policy as its symbolic reasoning mechanism. Decision making policy is a popular mechanism to represent the decision making logic of autonomous systems, for example, being the result of advanced planning or learning processes [20], [21], [22], [23]. Formally, a decision making policy is defined as follows.

A state vector is a set of discrete state variables $\mathbf{S} = \{S_1, \ldots, S_n\}$ where each variable S_i takes on values in some finite domain $Dom(S_i)$. $\mathbf{A} = \{a_1, \ldots, a_m\}$ is a finite set of actions. A decision making policy $\pi : S_1 \times \cdots \times S_n \rightarrow \mathbf{A_{exec}}$ is a complete function mapping each value of the state vector to a set of actions $\mathbf{A_{exec}} \subset \mathbf{A}$ to be executed. Table I illustrates an example decision making of an autonomous UAV. Note that, while in the example, the sets of actions executed are either empty or contain only one action, it is possible for a decision making policy to have multiple actions executed at a time.

TABLE I
An example policy

S_{flying}	S_{dest}	$S_{battery}$	Actions
$landed$	$not\_reached$	$above$	$\{take\_off\}$
$landed$	$not\_reached$	$below$	$\{\}$
$flying$	$not\_reached$	$above$	$\{navigate\_to\_point\}$
$flying$	$not\_reached$	$below$	$\{land\}$
$flying$	$reached$	$above$	$\{land\}$
$flying$	$reached$	$below$	$\{land\}$
$landed$	$reached$	$above$	$\{\}$
$landed$	$reached$	$below$	$\{\}$

At runtime, the UAV updates the value of the state vector at a fixed frequency, for example, every 100 milliseconds, by monitoring the environment and its internal state. Whenever the state vector changes, the UAV activates or deactivates the actions based on the decision making policy. For the verification problem considered in this paper, we assume that state variables are always monitored correctly. Note that, it is possible for a decision making policy to take uncertainty into account by explicitly representing the uncertainty as the values of state variables. For example, if there is a situation where the UAV does not know whether it is flying or landed, the situation can be explicitly represented by adding a $unknown$ value to the domain of the state variable S_{flying}.

B. Pylon inspection

Our case study concerns verifying the decision making of an autonomous UAV in the pylon inspection application. The task of the UAV is to autonomously inspect a pylon while taking into account safety requirements. At the beginning of each mission, the UAV must wait for permission from the human operator and loads a mission configuration that contains the location of the pylon and the desired flight pattern around the pylon, before performing the autonomous flight. During the flight, for safety, the human operator must always be able to intervene and manually control the UAV at any time by sending a request signal. The human operator can also send a signal to abort the mission and let the UAV autonomously returns to the home location. The UAV has an obstacle detection component that can detect whether there is any obstacle blocking its flight path. If the UAV is blocked for an extensive duration, it must notify the human operator and propose alternative options such as aborting the mission and going home or following an alternative flight path. If the obstacle detection component stops sending information for an extended period, for example, 0.5 seconds, it is considered as failed and the UAV must hover until new information is received. The UAV communicates with the human operator via a communication link that has three states: *stable*, *degraded* (only critical information can be sent) and *lost* (no information can be sent). The UAV must also constantly monitor the status of its obstacle detection component, communication link and battery. In case of contingency events, the UAV must react properly such as notifying the human operator or terminating the mission and flying to home. The UAV can also monitor its flying state, its current location and whether it has successfully sent a notification to the human operator.

The complete decision making policy consists of 16 state variables with 221184 possible combined state values. Based on the values of the state variables, at each time step, the UAV selects a set of actions to be executed from the 17 predefined actions such as $take\_off$, $land$, $go\_to\_landing\_zone$, $manual\_control$ and $notify\_critical\_battery$.

According to the requirements, the decision making policy must guarantee the following properties.

1) The UAV must always land at one of the predefined landing zones.
2) The UAV must never fly without permission.
3) Always notify the human operator when (1) the battery is low, (2) the UAV is being blocked, (3) the communication link is degraded or (4) the obstacle detection component is failed.
4) In a normal circumstance, the UAV should eventually complete the inspection.

It is not possible to guarantee the properties above in all situations. For example, the UAV cannot send a notification to the human operator when the communication link is lost. Thus, it is important to explicitly state all assumptions for the properties to be guaranteed so that one is aware of under which circumstances, the decision making is safe. To find necessary assumptions to guarantee a property with a model checker, first, the property is verified without any assumption. Once an assumption is found based on the counter-examples generated by the model checker, it is added to the specification. The property is then verified again until the model checker confirms that it holds in all situations. Below are some example assumptions for the properties above.

1) At the beginning, the UAV must be landed at a landing zone.
2) The UAV can only change its location while it is flying.
3) The UAV must have full control over its flying state, that is, it can only change its flying state by executing the $take\_off$ or $land$ actions.
4) Once the start permission has been given, it should never be taken back.
5) The communication link never gets lost (that is, the communication link is always either stable or degraded).
6) If the UAV keeps executing the action $take\_off$, eventually it will be flying.
7) If the UAV keeps executing the action $land$, eventually it will be landed.
8) Eventually, the UAV always has stable communication.
9) Eventually, the obstacle detection component never fails.
10) Eventually, the human operator gives the UAV the permission to start the mission.

The decision making policy, the complete list of assumptions for each property and the complete specification for each model checker are available online[3].

IV. Modeling

We now describe at a high level the formal model that needs to be represented in each model checker to perform verification. The execution of the decision making policy is represented by a time series, where each time step stands for a decision making cycle, that is, updating the values of the state variables and selecting the set of actions to be executed. The verification of each property starts without any assumption and there is no constraint on the values of the state variables. At each time step, a state variable can be assigned any value from its domain. It is to represent that the environment is non-deterministic and the UAV system does not have any control over the state variables.

The behavior of the environment is represented by a set of assumptions constraining the transitions of the state variables. Note that, the effects of action execution are also considered as assumptions. For example, an assumption on the interaction between action execution and the environment is that if the UAV keeps executing the action $take\_off$, eventually it will be flying.

The decision making policy determines which actions are executed at each time step based on the value of the state variables. Since a decision making policy often contains a large number of mappings, for example, more than 2×10^5 in our case study, modeling a policy in its original format as described in Section III results in a large model which requires a large amount of memory and computational resource to solve. Therefore, we first transform the decision making policy to a more compact format.

[3]https://github.com/hoangtungdinh/irc2020-supplemental-material

Because a policy can be seen as a truth table where the inputs are the values of the state variables and the outputs are the values of binary variables corresponding to the activation of actions, we use a technique called two-level logic minimization [24] to compactly represent a policy. Two-level logic minimization concerns finding a minimum formula in disjunctive normal form (DNF), that is, a disjunction of conjunctive clauses, of a boolean function or a truth table.

We employ the Espresso algorithm [25] to perform two-level logic minimization. The Espresso algorithm reduces the policy's truth table to a set of formulas in DNF. Each DNF formula represents the condition in which an action is executing. Table II shows the result after applying the two-level logic minimization technique on the example policy in Table I.

TABLE II
AN EXAMPLE REDUCED POLICY

Action	DNF formula
$navigate\_to\_point$	$(S_{flying} = flying \wedge S_{dest} = not\_reached$ $\wedge S_{battery} = above)$
$take\_off$	$(S_{flying} = landed \wedge S_{dest} = not\_reached$ $\wedge S_{battery} = above)$
$land$	$(S_{flying} = flying \wedge S_{dest} = reached) \vee$ $(S_{flying} = flying \wedge S_{battery} = below)$

The DNF representation of decision making policies can be easily encoded in any specification language provided by the model checkers. In contrast, since the properties and assumptions often contain temporal expressions, it is desirable to represent them in a temporal logic such as Linear Temporal Logic (LTL).

We now describe how the decision making policy is modeled in each model checker. For illustration, we include simple and incomplete specifications in the specification languages provided by the model checkers. The illustrative specifications consist of only two state variables, `S_flying` and `S_pylon_inspection`, as well as only two actions, `take_off` and `land`. The DNF formula corresponding to an `action` is denoted by `DNF(action)`.

A. *ProB*

ProB [9] is a model checker for the B-method, a methodology for the formal development of computer systems. ProB allows one to specify the system in different formal languages such as B, Event-B and Z and supports different validation methods including invariant checking, constraint based checking, refinement checking, bounded model checking, symbolic model checking and LTL model checking. As the B language is well-documented and provided with many examples within ProB, we model the decision making policy as a B machine using the B language. Since it is difficult to express temporal expressions in B or Z [26], we use the LTL model checking method as it is the only validation method in ProB allowing the specification of temporal expressions.

We represent the decision making policy as an abstract machine in B. A B-abstract machine includes variables, an invariant on the variables, initial states and operations that can perform deterministic or non-deterministic value assignment for the variables. The variables can be typed by using the invariant to enforce its domain to be a set.

Each state variable S_i and each action a_j is represented as an abstract variable. The domain of each state variable $Dom(S_i)$ is represented by a set while each action is a boolean variable. Listing 1 illustrates the B-specification of the decision making policy.

Listing 1
PROB MODEL

```
ABSTRACT_VARIABLES
  S_flying ,
  S_pylon_inspection ,
  take_off ,
  land
SETS
  D_flying={flying , on_the_ground};
  D_pylon_inspection={complete , not_complete};
INVARIANT
  S_flying : D_flying &
  S_pylon_inspection : D_pylon_inspection &
  take_off : BOOL &
  land : BOOL
INITIALISATION
  S_flying :: D_flying;
  S_pylon_inspection :: D_pylon_inspection;
  IF DNF(take_off) THEN
    take_off := TRUE ELSE take_off := FALSE END;
  IF DNF(land) THEN
    land := TRUE ELSE land := FALSE END;
OPERATIONS
  update_state =
    BEGIN
    S_flying :: D_flying;
    S_pylon_inspection :: D_pylon_inspection;
    IF DNF(take_off) THEN
      take_off := TRUE ELSE take_off := FALSE END;
    IF DNF(land) THEN
      land := TRUE ELSE land := FALSE END;
    END
END
```

The B-machine contains a single operation. The operation first assigns values of the state variables non-deterministically. After that, the value of each action variable is determined by the values of the state variables and its corresponding DNF formula. The DNF formula for each action is represented using B's *IF-THEN-ELSE* expression. The B machine is initialized in the same manner, that is, by assigning a random value to each state variable and then deciding the value of each action variable based on the values of the state variables.

Desired properties are expressed in LTL. ProB checks LTL properties using an adapted version of the tableau algorithm [26]. Once an assumption is found, it is connected to the corresponding property using the LTL formula of the form $\phi_{assumption} \Rightarrow \phi_{property}$.

Note that, ProB has some preferences to be set by users for each model checking problem. The two most important preferences are the maximum number of initializations and the maximum number of outputs by applying an operation. For the model checking result to be complete, one needs to set those two preferences to the total number of possible state vector values. It is to make sure that the decision making policy is checked against all possible initial states of the state vector and all possible changes in the environment.

B. SPIN

SPIN [10] is a model checker tailored for verifying asynchronous systems. SPIN uses PROMELA, an imperative modeling language specialized in describing concurrent systems, as the system specification language. In PROMELA, one defines variables and processes that manipulate the variables. From any running process, further asynchronous processes can be launched. SPIN allows properties to be specified in LTL. SPIN translates a LTL property to a *never-claim*, a feature in PROMELA which specifies a situation that should never occur, to perform model checking.

Since a decision making policy can be considered as a sequential system, our PROMELA model consists of only one process that performs an infinite decision making loop. Similar to the ProB model, a SPIN's decision making cycle first updates the value of each state variable non-deterministically and then sets the value of each action variable based on the values of the state variables. In the PROMELA model, each action variable is a boolean variable and each state variable is a PROMELA's *subtype* so that its domain is restricted to a set of symbolic constants. Listing 2 illustrates our PROMELA specification for the decision making policy.

Listing 2
SPIN MODEL

```
mtype:D_flying={flying , on_the_ground};
mtype:D_pylon_inspection={complete , not_complete};
mtype:D_flying S_flying
mtype:D_pylon_inspection S_pylon_inspection
bool take_off
bool land
bool valid_state = false
bool first_valid_state = false
inline update_S_flying() {
  if
  :: S_flying = flying
  :: S_flying = on_the_ground
  fi
}
inline update_S_pylon_inspection() {
  if
  :: S_pylon_inspection = complete
  :: S_pylon_inspection = not_completen
  fi
}
inline simulate_one_step() {
  update_S_flying()
  update_S_pylon_inspection()
  if
  :: DNF(take_off) -> take_off = true
  :: else -> take_off = false
  fi
  if
  :: DNF(land) -> land = true
  :: else -> land = false
  fi
}
init {
  atomic {
    simulate_one_step()
    valid_state = true
    first_valid_state = true
  }
  first_valid_state = false
  do
  :: atomic { simulate_one_step() }
  od
}
```

Note that, in PROMELA, variables are always initialized to the default values of their corresponding types. Because of that, the initial value of each state variable is fixed and the initial value of each action variable is not set according to the decision making policy. To represent the non-deterministic initialization of the decision making policy, the PROMELA process first performs an initialization step that assigns values to the state variables non-deterministically and enforces the values of the action variables with respect to their DNFs. The initial values in the PROMELA model before the initialization step is then not taken into account while verifying the properties. We do so by defining two boolean variables *valid_state* and *first_valid_state* to indicate whether the system is in a (first) valid state, that is, after the initialization step. To ignore the initial state of SPIN's variables, all the LTL properties and assumptions are then specified in the following form.

$$valid\_state \Rightarrow (\phi_{assumption} \Rightarrow \phi_{property}) \qquad (1)$$

Expressions related to the initial state of the system such as Assumption 1 can then be represented as follows.

$$first\_valid\_state \Rightarrow (S_{flying} = landed) \qquad (2)$$

C. TLC

TLC [11] is the model checker for TLA+ (Temporal Logic of Actions), a declarative language to describe non-deterministic concurrent systems. In TLC, both the system specification and the properties can be modeled as TLA+ formulas. TLC can check TLA+ specifications of the following form [27].

$$Spec \triangleq Init \wedge \Box[Next]_{vars} \wedge Liveness \qquad (3)$$

$Init$ is a predicate constraining the initial states of the variables in the specification. The $Next$ predicate constrains the relation between the variable values before and after a step. A TLA+ specification must allow *stuttering steps*, that is, steps in which all the variables stay unchanged. The $vars$ subscript is a set of all the variables in the specification. The symbol $[Next]_{vars}$ is a compact way to state that the values of all the variables in the next step must either satisfy the $Next$ predicate or remain unchanged. Finally, $Liveness$ is the conjunction of liveness formulas.

Listing 3 illustrates our TLA+ specification. Since TLA+ variables are untyped, we define predicates to enforce the domain of each variable. In our specification, the $Init$ predicate is the conjunction of the two predicates: $InitTypeInvariant$ to constrain the initial values of the variables to be in their domain and $InitPolicy$ to assign the values of the action variables according to their DNFs. Similarly, two predicates

$NextTypeInvariant$ and $NextPolicy$ in the $Next$ predicate are used to constrain the variables after each step. Because there is no constraint on the state variables except the type invariant, their values are allowed to change non-deterministically at each time step. Note that, in Listing 3 we do not include the $Liveness$ predicate in the $Spec$ since $Liveness$ is not required to represent a decision making policy.

Listing 3
TLA+MODEL

```
—— MODULE decision_making ——
VARIABLE
  S_flying, S_pylon_inspection, take_off, land
vars ≜ <<S_flying, S_pylon_inspection,
          take_off, land>>
D_flying ≜ {"flying", "on_the_ground"};
D_pylon_inspection ≜ {"complete", "not_complete"};
InitTypeInvariant ≜
  ∧ S_flying ∈ D_flying
  ∧ S_pylon_inspection ∈ D_pylon_inspection
  ∧ take_off ∈ BOOLEAN
  ∧ land ∈ BOOLEAN
NextTypeInvariant ≜
  ∧ S_flying' ∈ D_flying
  ∧ S_pylon_inspection' ∈ D_pylon_inspection
  ∧ take_off' ∈ BOOLEAN
  ∧ land' ∈ BOOLEAN
InitPolicy ≜
  ∧ IF DNF(take_off) THEN take_off ELSE ¬take_off
  ∧ IF DNF(land) THEN land ELSE ¬land
NextPolicy ≜
  ∧ IF DNF(take_off') THEN take_off' ELSE ¬take_off'
  ∧ IF DNF(land') THEN land' ELSE ¬land'
Init ≜ InitTypeInvariant ∧ InitPolicy
Next ≜ NextTypeInvariant ∧ NextPolicy
Spec ≜ Init ∧ □[Next]_vars
```

Because TLC does not check arbitrary temporal formulas, one could not encode both assumptions and properties in the temporal formula of the form $\phi_{assumption} \Rightarrow \phi_{property}$. To overcome this problem, we must add the assumption to TLA+ $Spec$. Assumptions on the initial states of the system, e.g., Assumption 1, can be added to the $Init$ predicate. Assumptions related to the transitions of the state variables such as Assumption 2 can be added to the $Next$ predicate. Assumptions on the liveness of the system such as Assumption 6 can be added to the $Liveness$ predicate. Different from ProB and SPIN where only knowledge about LTL is required for one to model assumptions, using TLA+, one must have a deep understanding of the TLA+ specification to decide where to add an assumption. Nevertheless, we found TLA+ expressive enough to encode all assumptions and properties in our case study.

D. Alloy

Similar to TLA+, Alloy [12] is a declarative language. However, unlike TLA+ which is based on temporal logic, Alloy is based on first-order logic. Two main concepts in Alloy are *signature* and *fact*. A *signature* declares atoms or types and their relations to other signatures. A *fact* defines a constraint on the elements of the model. Properties in Alloy are also represented as first-order logic constraints. An Alloy model can be verified using Alloy Analyzer, a SAT-based constraint solver. Alloy Analyzer verifies a property by performing an exhaustive search for an instance of the Alloy model with a bounded size that violates the property.

Listing 4 illustrates our Alloy model. An execution trace of a decision making policy is a linear ordering of states and actions. Since Alloy is based on first-order logic, it does not have the notion of time or state orders. Fortunately, Alloy provides the *util/ordering* library which allows users to impose a linear ordering on a set of atoms.

Listing 4
ALLOY MODEL

```
open util/ordering[Step]
abstract sig D_flying {}
one sig flying, on_the_ground extends D_flying {}
abstract sig D_pylon_inspection {}
one sig complete, not_complete extends D_flying {}
abstract sig Action {}
one sig take_off, land extends Action {}
sig Step {
  S_flying: D_flying,
  S_pylon_inspection: D_pylon_inspection,
  Executing: set Action
}
fact {
  all step: Step {
    take_off in step.Executing <=> DNF(take_off)
  }
}
fact {
  all step: Step {
    land in step.Executing <=> DNF(land)
  }
}
```

We specify the decision making policy in Alloy as follows. The type of each state variable is defined as an abstract signature. The values of each domain are defined by concrete signatures extending the abstract signature of the corresponding type. Similarly, we define all the actions as concrete signatures of the type *Action*.

To represent the execution steps of the decision making policy, we define the signature *Step* and impose a linear ordering on its atoms. A *Step* consists of one atom per state variable and a set of *Action*s representing the actions that are executed in that step. No constraint is imposed on the state variables so that they can change non-deterministically at each step. Each action is constrained to be in the set of executing actions or not by a *fact* representing its DNF formula.

Assumptions are also represented as *facts* in Alloy. Since both assumptions and properties must be represented in first-order logic, their temporal expressions are translated to first-order logic constraints on a trace of steps. Translating expression on the initial states, e.g., Assumption 1, or on all states, e.g., Assumption 2, can be done easily by imposing constraints on the first step or all steps. In contrast, more effort is required to translate expressions related to the liveness of the system, e.g., Assumption 6. A liveness expression imposes a constraint on an infinite execution trace while Alloy only performs bounded model checking on a finite trace of the execution of the system.

A typical way to represent an infinite execution trace with a bounded number of execution steps is to create a back loop from the last step to any of the previous steps [28]. The liveness expressions are then translated to constraints imposed on the infinite execution trace. The back loop is modeled in Alloy by adding a fact constraining that the values of the atoms in the last step must be the same as the values of the atoms of a previous step.

Although it requires more effort to represent assumptions and properties in Alloy, similar to TLA+, we found Alloy expressive enough to represent all the assumptions and properties in our case study. Note that, since Alloy Analyzer only searches for counter-examples up to a bounded length specified by users, the verification process performed by Alloy is not complete. Alloy might miss a counter-example if the bounded length is not large enough.

E. NuSMV

NuSMV [13] is a model checker that supports both Binary Decision Tree (BDD) based symbolic model checking and bounded model checking. The NuSMV specification language is tailored to describe finite state machines that manipulate a set of variables. One can specify properties in NuSMV using any LTL or CTL (Computation Tree Logic) formula.

Listing 5 illustrates our NuSMV model. We found that the NuSMV specification is simpler than the specifications in other languages. Since NuSMV supports typed variables and explicit state transitions, modeling a decision making policy in NuSMV is straightforward. Each state variable is represented by a variable with its domain. As the state variables can be freely changed, no transition constraint is defined on the state variables.

Listing 5
NUSMV MODEL

```
MODULE decision_making
VAR
  S_flying: {flying, on_the_ground};
  S_pylon_inspection: {complete, not_complete};
DEFINE
  take_off := DNF(take_off);
  land := DNF(land);
```

NuSMV supports the *DEFINE* declaration that allows one to declare a symbol associated with a common expression. During the verification process, a symbol in the *DEFINE* declaration is not treated as a variable but is considered as a macro and is replaced by the expression it is associated with. Taking the advantage of this NuSMV language feature, instead of defining each action as a boolean variable as in some other specification languages, we define each action as a symbol associated with its DNF formula. Doing so reduces the number of variables in our model as well as the state space at the cost of increasing the size of the property formulas.

As NuSMV provides full support for any LTL formula, modeling assumptions and properties is straightforwad using the formula $\phi_{assumption} \Rightarrow \phi_{property}$.

V. PERFORMANCE EVALUATION

In this section, we present the performance evaluation of the model checkers on verifying the decision making of the UAV in our case study. As our preliminary results show that not all model checkers can cope with the size of the decision making policy with 221184 possible state values and 17 actions, we create a simplified version of the policy with 6 state variables (64 possible state values) and 6 actions. We first evaluate each model checker on the simplified policy to ensure that our modeling is correct and to have insights into the computation time of each model checker with a medium-size policy. After that, the model checkers are evaluated against the complete decision making policy.

For each model checker, we first verify each property without any assumption. When there are missing assumptions, each model checker should be able to return a counter-example demonstrating the violation of the property. The evaluation is performed on a computer with an Intel(R) Core(TM) i7-7700K CPU @ 4.20GHz and 32GB of RAM. For each property, each model checker is allowed to run within 20 minutes. The model checker is terminated if after 20 minutes it cannot conclude whether the property holds or not.

Our first observation is that with the simplified policy, all the model checkers are able to return a counter-example when there are missing assumptions. However, when all necessary assumptions are added, SPIN cannot verify that Property 1 and Property 4 hold and ProB took a long time to verify that Property 4 holds. ProB took more than 31 seconds to verify that the liveness property 4 holds while TLC, Alloy and NuSMV only need about one second. Note that, often, it takes longer for a model checker to conclude that a property holds than to conclude that a property does not hold.

Table III shows the computation time required by each model checker when they verify each property with all necessary assumptions. Note that, with the simplified policy, Property 2 was not taken into account as state variables related to Property 2 were removed from the policy.

TABLE III
THE CHECKING TIME (IN SECONDS) OF THE SIMPLIFIED POLICY.

	ProB	SPIN	TLC	Alloy	NuSMV
Property 1	1.686	-	1.121	0.419	0.009
Property 3	1.618	0.505	1.108	0.372	0.010
Property 4	31.260	-	1.367	0.554	0.012

For the complete policy with 16 state variables and 17 actions, ProB and TLC are not able to verify the properties, even without any assumption. ProB and TLA are not able to show that the properties do not hold and generate counter-examples. SPIN can find a counter-example for Property 1 when there is no assumption in 2:43 minutes. However, SPIN is not always able to verify that a property does not hold and similar to ProB and TLC, SPIN cannot verify any property when all necessary assumptions are added.

In contrast, Alloy and NuSMV are able to verify all the properties with the complete policy. Alloy and NuSMV's computation time is not affected much by the size of the policy. When there are missing assumptions, they both quickly find a counter-example to demonstrate that the property is violated. When all necessary assumptions are added, Alloy can verify each property within few seconds and NuSMV can verify them in less than one second. Table IV shows the computation time of the model checkers to verify the complete policy.

TABLE IV
THE CHECKING TIME (IN SECONDS) OF THE COMPLETE POLICY.

	ProB	SPIN	TLC	Alloy	NuSMV
Property 1	-	-	-	1.028	0.582
Property 2	-	-	-	1.332	0.016
Property 3	-	-	-	1.968	0.016
Property 4	-	-	-	4.013	0.575

The significant difference in the performance of the model checkers can be explained as follows. ProB, SPIN and TLC are *explicit-state* model checkers, that is, they perform verification on an explicit representation of system's states and transitions. As a result, explicit-state model checkers face the state explosion problem. Since the complete policy has a large number of states and actions, no explicit-state model checker can successfully verify any property.

Different from the three model checkers above, NuSMV performs symbolic model checking. Instead of representing the state space explicitly, NuSMV represents the state space symbolically using Binary Decision Diagram (BDD). BDD-based methods are often more efficient than explicit state enumeration methods. Moreover, as discussed in Section IV-E, the NuSMV specification language supports symbol macros, which allows us to represent each action without having to use a boolean variable, which in turn reduces the state space of the problem. This explains why NuSMV has the best performance among the five model checkers.

Alloy tackles the state exploration problem using a different approach. Alloy performs bounded model checking (BMC) using SAT solvers. BMC searches for counter-example of a bounded length k ($k = 50$ in our evaluation). A BMC problem is translated to a propositional satisfiability problem and then can be solved efficiently using an off-the-shelf SAT solver.

The evaluation result shows that explicit-state model checkers such as ProB, SPIN and TLC are not suitable for verifying decision making policies. In contrast, Alloy and NuSMV show great potential when they can handle a large policy with hundreds of thousands of states within a few seconds. Note that, we do not claim that the implemented models are the most optimal ones. Yet, with reasonable effort to get used to the modeling language of each model checker, we believe that the evaluation results are representative enough to reveal the applicability of the investigated model checkers in verifying decision making.

VI. CONCLUSIONS

We studied the suitability of model checkers for verifying decision making in a use case of an autonomous UAV mission. Our conclusions clearly relate to our experience in this use case, and cannot blindly be generalized. Further validations with a broad variety of case studies are required for any more general conclusions. However, our study does show that model checkers are not generally applicable. On the contrary, they may only show benefits in cases with specific characteristics. Our case study, in particular, has a large policy and is characterized by a diversity of properties and assumptions. These characteristics have proven to be useful for providing insights into the limitations of the model checkers.

We applied five different model checkers to verify the discrete decision in our case study. While all investigated model checkers are expressive enough to represent the decision making policy, the environment assumptions and the desired properties, we found that not all model checkers provide the same expressive comfort nor are suitable to verify the decision making policy due to severe limitations in computational performance. Only Alloy and NuSMV are able to verify the entire decision making policy of the UAV. Our evaluation results also indicate that explicit-state model checkers such as ProB, SPIN and TLC are not suitable for this problem. In contrast, model checkers using bounded model checking and symbolic model checking show a great potential. In the future, we plan to further exploit the ability of NuSMV and Alloy by performing verification at a lower level of abstraction of the environment and with multi-agent scenarios.

ACKNOWLEDGMENT

This research is partially funded by the Research Fund KU Leuven. We thank the anonymous reviewers for their helpful comments.

REFERENCES

[1] M. Fisher, L. Dennis, and M. Webster, "Verifying autonomous systems," *Communications of the ACM*, vol. 56, pp. 84–93, 2013.

[2] W. Schwarting, J. Alonso-Mora, and D. Rus, "Planning and decision-making for autonomous vehicles," *Annual Review of Control, Robotics, and Autonomous Systems*, 2018.

[3] M. Webster, C. Dixon, M. Fisher, M. Salem, J. Saunders, K. L. Koay, K. Dautenhahn, and J. Saez-Pons, "Toward Reliable Autonomous Robotic Assistants Through Formal Verification: A Case Study," *IEEE Transactions on Human-Machine Systems*, vol. 46, pp. 186–196, Apr. 2016.

[4] L. A. Dennis, M. Fisher, N. K. Lincoln, A. Lisitsa, and S. M. Veres, "Practical verification of decision-making in agent-based autonomous systems," *Automated Software Engineering*, vol. 23, pp. 305–359, Sep. 2016.

[5] J. Choi, S. Kim, and A. Tsourdos, "Verification of heterogeneous multi-agent system using MCMAS," *International Journal of Systems Science*, vol. 46, pp. 634–651, 2015.

[6] P. Gainer, C. Dixon, K. Dautenhahn, M. Fisher, U. Hustadt, J. Saunders, and M. Webster, "CRutoN: Automatic Verification of a Robotic Assistant's Behaviours," in *Critical Systems: Formal Methods and Automated Verification*. Springer, 2017, pp. 119–133.

[7] M. Webster, C. Dixon, M. Fisher, M. Salem, J. Saunders, K. L. Koay, and K. Dautenhahn, "Formal verification of an autonomous personal robotic assistant," *Proc. AAAI FVHMS*, pp. 74–79, 2014.

[8] C. Dixon, M. Webster, J. Saunders, M. Fisher, and K. Dautenhahn, "“The Fridge Door is Open”–Temporal Verification of a Robotic Assistant's Behaviours," in *Advances in Autonomous Robotics Systems*, M. Mistry, A. Leonardis, and C. Melhuish, Eds. Cham: Springer International Publishing, 2014, vol. 8717, pp. 97–108.

[9] M. Leuschel and M. Butler, "ProB: A model checker for B," in *International Symposium of Formal Methods Europe*. Springer, 2003, pp. 855–874.

[10] G. J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Reading, 2004, vol. 1003.

[11] R. Joshi, L. Lamport, J. Matthews, S. Tasiran, M. Tuttle, and Y. Yu, "Checking Cache-Coherence Protocols with TLA+," *Formal Methods in System Design*, vol. 22, pp. 125–131, Mar. 2003.

[12] D. Jackson, *Software Abstractions*. MIT press Cambridge, 2006, vol. 2.

[13] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "Nusmv 2: An opensource tool for symbolic model checking," in *International Conference on Computer Aided Verification*. Springer, 2002, pp. 359–364.

[14] A. Lomuscio, H. Qu, and F. Raimondi, "MCMAS: A model checker for the verification of multi-agent systems," in *International Conference on Computer Aided Verification*. Springer, 2009, pp. 682–688.

[15] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of probabilistic real-time systems," in *International Conference on Computer Aided Verification*. Springer, 2011, pp. 585–591.

[16] M. Luckcuck, M. Farrell, L. A. Dennis, C. Dixon, and M. Fisher, "Formal Specification and Verification of Autonomous Robotic Systems: A Survey," *ACM Comput. Surv.*, vol. 52, pp. 100:1–100:41, Sep. 2019.

[17] L. Dennis, M. Fisher, M. Slavkovik, and M. Webster, "Formal verification of ethical choices in autonomous systems," *Robotics and Autonomous Systems*, vol. 77, pp. 1–14, Mar. 2016.

[18] L. A. Dennis, M. Fisher, M. P. Webster, and R. H. Bordini, "Model checking agent programming languages," *Automated Software Engineering*, vol. 19, pp. 5–63, Mar. 2012.

[19] M. Webster, M. Fisher, N. Cameron, and M. Jump, "Formal Methods for the Certification of Autonomous Unmanned Aircraft Systems," in *Computer Safety, Reliability, and Security*, ser. Lecture Notes in Computer Science, F. Flammini, S. Bologna, and V. Vittorini, Eds. Springer Berlin Heidelberg, 2011, pp. 228–242.

[20] M. T. J. Spaan, T. S. Veiga, and P. U. Lima, "Decision-theoretic planning under uncertainty with information rewards for active cooperative perception," *Autonomous Agents and Multi-Agent Systems*, vol. 29, pp. 1157–1185, Nov. 2015.

[21] J. Fu, V. Ng, F. Bastani, and I.-L. Yen, "Simple and fast strong cyclic planning for fully-observable nondeterministic planning problems," in *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.

[22] H. T. Dinh, M. H. C. Torres, and T. Holvoet, "Sound and Complete Reactive UAV Behavior using Constraint Programming," in *ICAPS Workshop on Planning and Robotics*, 2018.

[23] S. Shalev-Shwartz, S. Shammah, and A. Shashua, "Safe, Multi-Agent, Reinforcement Learning for Autonomous Driving," *arXiv:1610.03295 [cs, stat]*, Oct. 2016.

[24] O. Coudert and T. Sasao, "Two-level logic minimization," in *Logic Synthesis and Verification*. Springer, 2002, pp. 1–27.

[25] R. K. Brayton, G. D. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*. Springer Science & Business Media, 1984, vol. 2.

[26] D. Plagge and M. Leuschel, "Seven at one stroke: LTL model checking for high-level specifications in B, Z, CSP, and more," *International Journal on Software Tools for Technology Transfer*, vol. 12, pp. 9–21, Feb. 2010.

[27] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[28] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking." *Advances in computers*, vol. 58, pp. 117–148, 2003.