Declarative Logic-based Pareto-Optimal Agent Decision Making

Tonmoay Deb, Mingi Jeong, Cristian Molinaro, Andrea Pugliese, Alberto Quattrini Li, Eugene Santos Jr., V.S. Subrahmanian, Youzhi Zhang

***Abstract*—There are many applications where an autonomous agent can perform many sets of actions. It must choose one set of actions based on some behavioral constraints on the agent. Past work has used deontic logic to *declaratively* express such constraints in logic, and developed the concept of a feasible status set (FSS), a set of actions that satisfy these constraints. However, multiple FSSs may exist and an agent needs to choose one in order to act. As there may be many different objective functions to evaluate status sets, we propose the novel concept of *Pareto-optimal feasible status sets* or POSS. We show that checking if a status set is a POSS is co-NP-hard. We develop an algorithm to find a POSS and in special cases when the objective functions are monotonic (or anti-monotonic), we further develop more efficient algorithms. Finally, we conduct experiments to show the efficacy of our approach and we discuss possible ways to handle multiple Pareto-optimal Status Sets.**

I. Introduction

Autonomous agents are becoming increasingly important in the real-world. A good example is self-driving cars (SDC for short) where agents already control several functions, such as lane changes and speed changes in Tesla vehicles [1]. Another example involves proposals for nuclear power plants involving agents that can increase coolant pressure, temperature, and more [2]. Autonomous agents are also being proposed for use with implantable medical devices [3]. These are critical applications. They are characterized by certain common features:

Declarative Operating Rules. The agents involved need to take actions while respecting declaratively specified behavioral requirements, i.e., the desired behavior should be specified in an easy to understand high-level language such as logic, not code specifying how that desired behavior is to be accomplished. For instance, a self-driving car should be forbidden to move into a lane when the location it is moving to is going to be occupied by another vehicle. It may be obligatory for an autonomous agent to shut off certain processes when the coolant level in a power plant drops below some threshold. An agent managing an implantable device may be permitted but not obliged to warn the user when there is a danger of a non-life threatening malfunction. All such behavioral requirements should be stated in a declarative language that is easy to understand for domain experts.

T. Deb and V.S. Subrahmanian are with Northwestern University, USA

M. Jeong, A. Quattrini Li, and E. Santos Jr. are with Dartmouth College, USA.

C. Molinaro and A. Pugliese are with University of Calabria, Italy.

Y. Zhang is with the Centre for Artificial Intelligence and Robotics, Hong Kong Institute of Science & Innovation, Chinese Academy of Sciences.

Concurrent Actions. The agents may perform zero, one or more actions simultaneously, e.g., shut off a process, send messages to other agents and/or human users.

Constraints on Actions. There are constraints on sets of actions that can be done concurrently, e.g., coolant pressure cannot be increased and decreased at the same time. Certain combinations of actions may lead to impossible or undesirable states (e.g., one where there is a nuclear leak). Such constraints can be expressed easily in high-level logical languages.

Autonomy. The agents are autonomous, i.e., they can make a conscious choice between different sets of actions that they can take at a given time.

Multiple Objectives. The agents may measure the desirability of a set of actions along multiple dimensions, e.g., annoyance to user if she gets too many alerts, maximizing safety of the environment considered, cost, time, and more.

Deontic logic [4], [5] has been studied for more than 50 years. It extends classical logic to support reasoning with the effects of actions on the state of the world. In multi-agent applications, agents should operate under certain behavioral constraints. In self-driving cars, for instance, agents should obey the rules of the road. They may be permitted to do certain things in some conditions, forbidden from doing things in other conditions, obliged to do some things in yet other circumstances, and more. Deontic logic therefore studies the permissions, obligations, and forbidden modalities and develops the logical foundations of their interactions both with each other, with classical logic and actions.

A declarative deontic logic framework within which we can express what the agent is permitted to do, obliged to do, and forbidden to do in various situations has already been proposed by [6], [7]. Their "IMPACT" framework defines "agent programs" that encode desired declarative agent behaviors, the syntactic concept of a status set, and the semantic concept of a feasible status set (FSS). Intuitively, an FSS captures a set of actions that the agent can perform, compatible with its operating rules, constraints on actions, concurrency constraints, and the deontic logic modalities. IMPACT was shown in [8] to support easy articulation of desired high-level behavioral requirements for 3 broad applications: transportation, supply chain management, and an online store. However, IMPACT does not incorporate any objective functions. Subsequently, [9] proposed the concept of optimal status sets in which an agent can choose a feasible status set (and hence a set of actions to perform) that optimizes a single objective function, but multiple objective functions are not allowed.

Real world agents may consider many factors. A nuclear

©2024 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. DOI: 10.1109/TCYB.2024.3394316

power monitoring agent may wish to minimize the number of alerts sent to the engineering team while simultaneously maximizing safety. This requires consideration of two orthogonal but incomparable objective functions. In general, no single solution might simultaneously optimize all objective functions. A typical approach to deal with this is *Pareto-optimality* [10]: a solution is *Pareto-dominated* if there is another solution that strictly improves some objective function value without degrading the other objective functions' values; and, a solution is *Pareto-optimal* if it is not Pareto-dominated. The *Pareto frontier* is the set of all Pareto-optimal solutions, all of which are considered equally good. To the best of our knowledge, the combination of logical methods and Pareto-optimality has not been studied before.

In this paper, we combine deontic logic [11] and Pareto-optimality. Specifically, we make the following contributions:

1) We propose the new concept of a Pareto-Optimal (Feasible) Status Set, or POSS for short, which combines deontic logic and Pareto optimality. It combines the power of logic and the power of optimization. We show that the problem of checking if a given status set is Pareto-optimal is co-NP-hard, and it is co-NP-complete under some reasonable assumptions.
2) We develop the first algorithm to find a POSS for a given agent-state pair.
3) We develop the first algorithms to compute POSS's when the objective functions are monotonic (or anti-monotonic).
4) We report on a prototype implementation of our framework, showing that POSS works well on a realistic collaborative SDC scenario, where we vary several parameters and assess their impact on performance.

The paper is organized as follows. Section II discusses related work. Section III provides a motivating example of a futuristic collaborative SDC scenario in which multiple cars collaborate to achieve their objectives. Section IV provides a brief overview of IMPACT [6], [7], [8]. Section V extends IMPACT so that agents consider multiple objective functions and introduces Pareto-optimal (Feasible) Status Sets. It then studies the complexity of the problem. Section VI presents exact and heuristic algorithms to solve this problem. Section VII presents an experimental assessment of these algorithms. Section VIII discusses possible ways to handle the situation where a Pareto front has multiple Pareto-optimal Status Sets. Section IX describes limitations and outlook for future work. Section X concludes the paper.

II. Related Work

We build upon deontic logic based agents introduced by [6], [8]. While there is plenty of previous work on multiagent systems (e.g., see [12], [13], [14]), to the best of our knowledge, there is only one effort [9] that tries to build agents that optimize their actions in the presence of both deontic behavioral rules and constraints. [9] is limited to one objective function, while our approach can handle several. [15] proposes the jDALMAS system, which includes a preference structure based on a theory of normative positions [16]. They consider a partial ordering on actions to be taken by an agent, but do not consider explicit *numerical* objective functions. [15] does not consider objective functions. In addition, we develop novel algorithms for weakly/strongly monotonic and anti-monotonic objective functions, whereas neither [9] or [15] consider such specialized objective functions.

[17] provides an excellent overview of logic-based agent systems, but does not say much about deontic logic (except for the jDALMAS effort mentioned above) or optimization, suggesting that there is a lot of room for work in this space.

There have been many numeric approaches to Pareto optimization [18], [19], [20], [21], [22], [23] that do not involve logic. All of these algorithms focus on searching for optimal solutions over the feasible solution space, but they do not consider how to generate feasible solutions over a *logical* solution space, which is fundamental for the logical approach. In multi-agent settings, [24] proposes a distributed approach to find a Pareto-optimal solution. [25] looks at a very specific scheduling problem where two agents compete to work on a machine: one agent tries to minimize the number of delayed jobs it initiated, while the other agent wants to maximize a different quantity associated with its jobs. [26] studies a similar situation. [27] combines deterministic policy gradients with Pareto optimization to develop good recommender systems. [28] provides an excellent view of agent-based methods for network traffic management. While these are important efforts, none of them combine logic and optimization. The behaviors of these agents are not declaratively specified and in some cases, optimization focuses on very specific objective functions. In contrast, we provide declarative deontic logic based constraints[1] that are easy to explain to stakeholders and show how our objective functions can be easily optimized. We present different types of algorithms depending on the different properties of the objective functions (e.g., no restrictions on the objective function, weakly/strongly monotonic, and weakly/strongly anti-monotonic). Additionally, we propose approximation algorithms.

Future work could examine the use of probabilistic and/or defeasible deontic rules in situations where there is uncertainty about the state and/or where there is uncertainty in whether certain behavioral norms can be relaxed [32], [33].

III. Motivating Example

Consider a divided highway as shown in Figure 1. Cars are traveling from left to right on one side of the highway which can be thought of as a matrix. For simplicity, in this example,

[1]A logical *theory* consists of a set of formulas (which include rules) in logic. An *interpretation* is an assignment of truth values to atomic formulas. A *model* of a logical theory is an interpretation that satisfies all the formulas in the logical theory. We can therefore see an analogy between integer 0-1 constraints and logic. Just as numeric 0-1 constraints, such as $x_a + x_b \geq 1$, constrain the space of solutions, logical formulas (including rules) constrain the space of interpretations that can be models. For instance, considering the logical formula $(a \vee b)$, the models are the interpretations that make at least one of a, b true. With the rule $a \rightarrow b$ acting as a constraint on the space of interpretations, we limit interest to those interpretations that either make b true or a false, or both. The articulation of how logical formulas and rules can be viewed as constraints goes back several decades. We refer the reader to [29], [30], [31] for a detailed exposition on why logical rules can be viewed as constraints. That said, not all constraints can be viewed as logical rules.

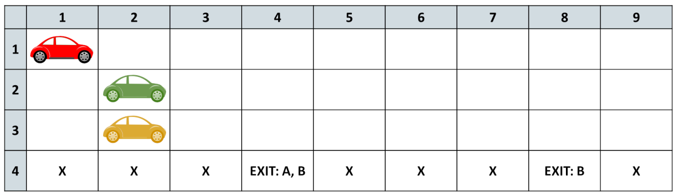


Fig. 1. A highway represented as a matrix (cars traveling from left to right).

the number of cars is fixed. Some cells are marked with an “X” to indicate that there is no road there. Some cells are marked “EXIT” to specify that there is an exit at that location. The exit also shows the destination (location A or B). A car that exits at location $(4, 4)$ can make it to both locations A and B, while one that exits at $(4, 8)$ can only get to B. Initially, the red car is traveling at 2 cells/second, while the green and orange cars are traveling at 1 cell/second.

A. State

We assume the existence of an arbitrary but fixed logical language within which the state can be expressed. We assume readers are familiar with standard expressions such as constants, variables, predicate symbols, atoms, and formulas in logic [34]. Following Prolog convention [35], we denote variables with upper case symbols—everything else will be denoted via lower case symbols.

At any point t in time, the *state* is a set of ground (i.e., containing constants only) logical atoms. In our motivating example, we use atoms of the following form:

1) $at(car, x, y, t)$ describes the location (x, y) of a car at time t, e.g., $at(red, 1, 1, 1)$ says that at time 1, the red car is at location $(1, 1)$.
2) $speed(car, s, t)$ is the speed of a car at time t, e.g., $speed(red, 2, 1)$ says that at time 1, the red car is traveling at 2 cells/second.
3) $dest(car, loc)$ specifies the destination of a car, e.g., $dest(red, B)$ says that the red car’s destination is B. This means the red car can take either exit in Figure 1.
4) $exit(y, loc)$ specifies where there is an exit and the location it leads to, e.g., $exit(8, B)$ says that there is an exit to B at location $(4, 8)$ (for simplicity, in this example, we assume exits are always in the bottom lane which is why the x value is not explicitly stated).

The table below shows the initial state S_0 of our running example—additionally, the initial state stores information on two exits at locations $(4, 4)$ and $(4, 8)$ leading to A, B and B, respectively. All three agents know this initial state.

car	*at*	*speed*	*dest*
red	(1,1)	2	B
green	(2,2)	1	A
orange	(3,2)	1	B

Furthermore, we assume the existence of a derived predicate $pred\_at(car, x, y, t{+}t')$ that predicts the location (x, y) of car at time $t + t'$, assuming inertia, i.e., that the car continues at its current speed without making any changes. This predicate can be readily derived from the at and $speed$ predicates.

B. Agent Actions

We assume the existence of a language with a set of *action symbols*, which generate *action atoms* (or simply actions) using the constants and variables from the language used to express a state above. In our motivating example, the cars are capable of taking the following actions:

1) $accel(car, s_1, s_2, t)$ says car accelerates from speed s_1 to s_2 at time t. Here $s_1 < s_2$.
2) $decel(car, s_1, s_2, t)$ says car decelerates from speed s_1 to s_2 at time t. Here $s_1 > s_2$.
3) $continue(car, t)$ keeps car going at its current speed at time t. So if the red car executes the action $continue(red, 1)$ at time 1, it will end up at location $(1, 3)$ at time 2.
4) $go\_left(car, t)$ moves car one lane to the left. So if the green car performs this action in its initial state, then it will end up at time 2 at $(1, 3)$ (which would lead to a collision if the red car performed the action in the preceding bullet).
5) $go\_right(car, t)$ moves car one lane to the right. So if the green car performs this action in its initial state, then it will end up at time 2 at $(3, 3)$ (which would lead to a collision with the orange car if that car were to execute the “continue” action at time 1).
6) $exit(car, x, y, t)$ says car is going to exit the highway at location (x, y) at time t.
7) $req(car1, car2, action, t)$ says that $car1$ requests $car2$ for permission to perform $action$ at time t. For instance, $req(green, red, go\_left(green, 2, 2, 1, 1), 1)$ has green telling red that it would like to shift lanes to the left at time 1 from location $(2, 2)$ going to a current speed of 1. This is like a turn signal. But green can perform this action only if red responds that it will slow down or shift to the right in order to avoid a collision.
8) $ok(car1, car2, action, t)$. Here $car2$ agrees to the request by $car1$ to perform $action$ at time t.
9) $deny(car1, car2, action, t)$ is the opposite situation: $car2$ does not agree to the request by $car1$ to perform $action$ at time t.

Assumption. Without loss of generality, we assume that one tick of time is enough for a car to make a request and receive a response and take an action.[2]

Each action α has a precondition $Pre(\alpha)$ which is a logical condition, an add list $Add(\alpha)$, and a delete list $Del(\alpha)$, both of which are sets of ground atoms. Action α is *executable* in state S_t if $Pre(\alpha)$ is true in S_t—if it is executed, then $Del(\alpha)$ is deleted from S_t while $Add(\alpha)$ is added to S_t in order to yield the new state.

As an example, for the action $\alpha = accel(car, s_1, s_2, t)$, we have $Pre(\alpha) = speed(car, s_1, t)\ \&\ (s_1 < s_2)$, $Del(\alpha) = speed(car, s_1, t)$, and $Add(\alpha) = speed(car, s_2, t + 1)$.

Autonomy. Cars can make decisions autonomously. One car may deny (or not respond) to a request from another car.

[2]One time unit t can be thought as having three parts: by $(t + 0.33)$, a car sends one or more messages to other cars, by $(t + 0.67)$ it receives responses, and it decides what to do before $(t + 1)$ and does it exactly at $(t + 1)$.

Collaboration. The messaging actions ($req, ok, deny$) enable agents to collaborate.

In general, we assume that an application domain has an associated set of action symbols and that we can define a notion of (ground) action atoms in the usual way [8], [36], [37]. The above shows a specific set of action symbols and action atoms in our running SDC example.

IV. Background: IMPACT Agents

We assume that arbitrary but fixed sets of actions and predicate symbols describing the state have been chosen as illustrated via the SDC example in the preceding section.

A. Agent Program

Every agent has an associated "agent program" that governs what the agent can and cannot do. In this section, we recall these definitions from [8]. If α is an action, then $\mathbf{F}\alpha$, $\mathbf{P}\alpha$, $\mathbf{O}\alpha$, $\mathbf{Do}\alpha$ are *status atoms* indicating that an action is forbidden, permitted, obligatory, and to be done, respectively.

An *operating rule* (or just rule) is an expression of the form

$$SA \leftarrow \chi \;\&\; SA_{\mathit{1}} \;\&\; \ldots \;\&\; SA_n$$

where $SA, SA_{\mathit{1}}, \ldots, SA_n$ are status atoms and χ is a logical condition (expressed using the predicate symbols). Intuitively, this rule says that if χ is true in the current state and if status atoms $SA_{\mathit{1}}, \ldots, SA_n$ are all true, then SA must also be true. These rules impose constraints—for example, the rule $\mathbf{F}\alpha \leftarrow \mathbf{Do}\beta$ imposes the logical constraint that if action β is done, then action α is forbidden.

An *agent program* is a finite set of rules.

Example 1. *The red car's allowed behavior can be expressed by the rules reported in Figure 2.*

$$\begin{array}{rcl}
\mathbf{P}accel(red, S1, S2, T) & \leftarrow & 1 \leq S2 \leq 3. \\
\mathbf{P}continue(red, T) & \leftarrow & speed(red, S, T) \,\&\, 1 \leq S \leq 3. \\
\mathbf{P}decel(red, S1, S2, T) & \leftarrow & 1 \leq S2 \leq 3. \\
\mathbf{F}accel(red, S1, S2, T) & \leftarrow & S2 > 3. \\
\mathbf{F}decel(red, S1, S2, T) & \leftarrow & S2 < 1. \\
\mathbf{F}continue(red, T) & \leftarrow & speed(red, S, T) \,\&\, S > 3. \\
\mathbf{F}continue(red, T) & \leftarrow & speed(red, S, T) \,\&\, S < 1. \\
\mathbf{P}exit(red, 4, 4, T) & \leftarrow & at(red, 3, 4, T). \\
\mathbf{P}exit(red, 4, 8, T) & \leftarrow & at(red, 3, 8, T). \\
\mathbf{F}go\_left(red, T) & \leftarrow & at(red, X, Y, T) \,\&\, X = 1. \\
\mathbf{F}go\_right(red, T) & \leftarrow & at(red, X, Y, T) \,\&\, X = 3. \\
\mathbf{P}go\_left(red, T) & \leftarrow & at(red, X, Y, T) \,\&\, X > 1. \\
\mathbf{P}go\_right(red, T) & \leftarrow & at(red, X, Y, T) \,\&\, X < 3. \\
\mathbf{O}deny(Car1, red, & & \\
go\_left(Car1, X, Y, S, T), T) & \leftarrow & pred\_at(red, X', Y', T+1) \,\&\, \\
 & & pred\_at(Car1, X', Y', T+1) \,\&\, \\
 & & \mathbf{Do}req(Car1, red, \\
 & & go\_left(Car1, X, Y, S, T), T).
\end{array}$$

Fig. 2. Red car's agent program.

The first seven rules say that the red car is allowed to have a speed in the range $[1, 3]$. *This is a logical constraint which ensures that the red car cannot have a speed outside such a range. The next two rules say that the red car can take either of the two exits on the highway (as both lead to its destination, B) when it is near the exits. The following four rules say the car cannot go left from the leftmost lane, nor can it go right from the rightmost lane (exit action is not considered a right turn but a different action), while it is permitted to go left (resp., right) when there is a lane on the left (resp., right). The last rule for the red car exhibits selfish behavior. It always denies requests that cause it to change its current behavior. All of these rules thus operate as logical constraints on actions.*

The agent program for the green car is identical to that of the red car except for three differences: (i) it cannot reach a speed greater than 2, (ii) it is obliged to take the first possible exit, and (iii) the last rule makes the green car's behavior kinder and more cooperative as it is willing to adjust its own behavior when other cars request a move.

The agent program for the orange car is identical to that of the green car but it must stick to a constant speed of 1 and it is permitted to exit at either of the two exits.

An agent program specifies constraints on the agent's behavior: what the agent is obliged to do or forbidden from doing in certain situations and what it is permitted but not required to do. Of course, the precondition of any permitted action must be true in a given state. Thus, these rules act as logical constraints on the agent's behavior.

B. Concurrent Action

An agent might choose to simultaneously do multiple things in a given state (e.g., a car may both accelerate and change lanes at the same time). In this case, we define a function called $\mathsf{conc}(A, S_t)$ which takes a set of actions A and state S_t as input and returns a new state S_{t+1}. [8] defines multiple possible ways of defining concurrent action execution.

C. Integrity Constraints

We can also write a set of integrity constraints defining valid states. Agents must not to take actions which would lead to a state that violates the integrity constraints. For instance, we would like an integrity constraint which says that an agent must not enter the same place as another agent. In general, an integrity constraint is either a *denial constraint* or a *definite constraint*, which we define below. If $A_1, \ldots, A_n$ are atoms (including atoms involving comparison operators), then a denial constraint has the form

$$\leftarrow A_1 \,\&\, \cdots \,\&\, A_n.$$

This denial constraint says that not all of $A_1, \ldots, A_n$ can be true in a given state. For example,

$$\leftarrow at(Car1, X, Y, T) \,\&\, at(Car2, X, Y, T) \,\&\, Car1 \neq Car2$$

is a denial constraint that says that two different cars cannot be in the same place at the same time (as this would be a collision). Many other denial constraints can be written for our sample SDC scenario. Again, these are all logical constraints on what can and cannot be done in a given state.

If $A_0, A_1, \ldots, A_n$ are atoms (atoms involving comparison operators are also allowed), then a *definite constraint* is an expression of the form

$$A_0 \leftarrow A_1 \,\&\, \cdots \,\&\, A_n.$$

Intuitively, a definite constraint says that if $A_1, \ldots, A_n$ are all true in a given state, then A_0 must also be true in that state. For example, the definite constraint $Loc1 = Loc2 \leftarrow dest(Car, Loc1)\ \&\ dest(Car, Loc2)$ says that a given car has only one destination.

D. Action Constraints

Finally, we allow the specification of a form of logical constraints called action constraints with the same syntax of the integrity constraints previously introduced, but involving action atoms instead of ordinary atoms. For instance, in our SDC scenario, $\leftarrow\ go\_left(Car, T)\ \&\ go\_right(Car, T)$ says that a car cannot try to move both left and right at the same time, $\leftarrow\ accel(Car, S1, S2, T)\ \&\ decel(Car, S1', S2', T)$ says it cannot both accelerate and decelerate at the same time, and $\leftarrow ok(Car1, Car2, Action, T)\ \&\ deny(Car1, Car2, Action, T)$ says it cannot both OK and deny the same request.

E. Status Set Semantics

In this section, we describe the semantics of agent programs from [6]. A *status set SS* is a finite set of ground status atoms. There are many status sets that can be consistent with a given state and a given agent program. We call such status sets *feasible* and they are defined as follows.

Definition 1. *A status set SS is* feasible *w.r.t. a state S_t, an agent program P, a set of integrity constraints IC, and a set of action constraints AC, iff:*

1) $\boldsymbol{O}\alpha \in SS \rightarrow \boldsymbol{P}\alpha \in SS$;
2) $\boldsymbol{O}\alpha \in SS \rightarrow \boldsymbol{Do}\alpha \in SS$;
3) $\boldsymbol{Do}\alpha \in SS \rightarrow \boldsymbol{P}\alpha \in SS$;
4) $\boldsymbol{P}\alpha \in SS \rightarrow \boldsymbol{F}\alpha \notin SS$;
5) $\boldsymbol{P}\alpha \in SS \rightarrow Pre(\alpha)$ *is true in* S_t;
6) *If* $SA \leftarrow \chi\ \&\ SA_1\ \&\ \ldots\ \&\ SA_n$ *is a ground instance of an operating rule in the agent program P and χ is true in state S_t and* $\{SA_1, \ldots, SA_n\} \subseteq SS$, *then* $SA \in SS$.
7) $\{\alpha \mid \boldsymbol{Do}\alpha \in SS\}$ *satisfies the action constraints in AC;*
8) *If S_t satisfies IC, then the new state* $\mathsf{conc}(\{\alpha \mid \boldsymbol{Do}\alpha \in SS\}, S_t)$ *satisfies IC.*

Given a set of numeric constraints, a "solution" is an assignment of values to the variables in those constraints that ensures that all the numeric constraints are satisfied. Feasible status sets are sets of ground status atoms which are assigned a 0-1 truth value (those in the set are 1, those not in the set are 0) which satisfy a given agent program in a given state. Thus, the rules in the agent program and the state act as logical constraints that determine which status sets are feasible and which ones are not.

Example 2. *Consider the (initial) state presented in Section III-A, the red car agent program in Example 1, and the integrity and action constraints discussed in Sections IV-C and IV-D, respectively. Let's focus on the red car. Suppose the red car has not received any request by other cars, and* conc *performs all actions in parallel determining the new positions of the red car given its speed, lane, etc.*

The status set SS consisting of the following status atoms is feasible:

$\boldsymbol{P}accel(red, 2, 3, 1), \boldsymbol{P}continue(red, 1), \boldsymbol{P}decel(red, 2, 1, 1),$
$\boldsymbol{F}go\_left(red, 1), \boldsymbol{P}go\_right(red, 1), \boldsymbol{Do}continue(red, 1),$
$\boldsymbol{F}accel(red, S1, S2, 1)$ *for every $S1$ and every* $S2 > 3$,
$\boldsymbol{F}decel(red, S1, S2, 1)$ *for every $S1$ and every* $S2 < 1$.

In fact, as per Definition 1, the status set SS above satisfies

- *Conditions 1)–4), which can be easily verified;*
- *Condition 5), assuming that for each $\boldsymbol{P}\alpha$ in SS, the current state satisfies α's preconditions;*
- *Condition 6), as each status atom that should be derived from the agent program is indeed in SS;*
- *Condition 7), as all action constraints are satisfied by the $\boldsymbol{Do}\alpha$ status atoms in SS;*
- *Condition 8), as the new state satisfies the ICs.*

V. Pareto-optimal (Feasible) Status Sets

In any given state, an agent might have 0, 1, or several feasible status sets. Each feasible status (FSS) set SS has an associated set $\mathbf{Do}(SS) = \{\alpha \mid \mathbf{Do}\alpha \in SS\}$ of actions to be done if the agent chooses SS. Given an agent program, state, action and integrity constraints, FSSs are like solutions, just as sets of numeric constraints have solutions. Which FSS should an agent choose and act in accordance with?

In our SDC scenario, there can be different criteria a car might follow, e.g., a first criterion might minimize lane shifts (to increase safety); a second criterion might be to leave the highway at the exit closest to the destination. One feasible status set SS_1 might have it stay in the current lane, feasible status set SS_2 might make the car change lane on the right bringing it closer to the exit, while feasible status set SS_3 might make the car change lane on the left, making it further from the exit. Thus, SS_1 and SS_2 are incomparable in that SS_1 optimizes the first criterion but not the second, while the opposite holds for SS_2. On the other hand, SS_3 is strictly worse than both SS_1 and SS_2 and should be ruled out. Thus, an agent may use one or more criteria to select which of the several feasible status sets to base its actions on; such criteria are expressed via objective functions, defined below.

Definition 2. *An* objective function *objf is a mapping that assigns a real number to any given feasible status set SS. objf is said to be:*

1) weakly monotonic *iff for any pair SS_1, SS_2 of feasible status sets,* $SS_1 \subseteq SS_2 \rightarrow objf(SS_1) \leq objf(SS_2)$;
2) strongly monotonic *iff for any pair SS_1, SS_2 of feasible status sets,* $\{\alpha \mid \boldsymbol{Do}\alpha \in SS_1\} \subseteq \{\alpha \mid \boldsymbol{Do}\alpha \in SS_2\} \rightarrow objf(SS_1) \leq objf(SS_2)$;
3) weakly anti-monotonic *iff for any pair SS_1, SS_2 of feasible status sets,* $SS_1 \subseteq SS_2 \rightarrow objf(SS_2) \leq objf(SS_1)$;
4) strongly anti-monotonic *iff for any pair SS_1, SS_2 of feasible status sets,* $\{\alpha \mid \boldsymbol{Do}\alpha \in SS_1\} \subseteq \{\alpha \mid \boldsymbol{Do}\alpha \in SS_2\} \rightarrow objf(SS_2) \leq objf(SS_1)$.

In the previous definition, the higher *objf*(SS), the better SS. As an example, an objective function that minimizes the number of lane shifts is defined as follows:

$$objf(SS) = -|\ \{\mathbf{Do}\, go\_left(car, t) \in SS\} \cup \{\mathbf{Do}\, go\_right(car', t') \in SS\}|.$$

We assume that each agent has an associated non-empty, finite set *OF* of objective functions. An agent will act in accordance with a feasible status set that is Pareto-optimal w.r.t. this set of functions.

Definition 3. *A feasible status set $SS^\star$ is* Pareto-optimal *w.r.t. a set OF of objective functions iff there is no other feasible status set SS such that for all objf $\in$ OF objf$(SS) \geq$ objf$(SS^\star)$ and for some objf $\in$ OF objf$(SS) >$ objf$(SS^\star)$.*

It is important to note that the above definition is key—it ties together the logical notion of a feasible status set (which is like a "solution" over a numeric domain) with the numeric notion of an objective function.

When only one objective function is present (i.e., $|OF| = 1$), Pareto-optimality coincides with the classical formulation of a (single objective function) optimization problem over the logical domain. That is, an optimal solution is a solution such that there is no other solution with a strictly better value for the objective function. In fact, with only one objective function *objf*, Definition 3 states that a feasible status set $SS^\star$ is Pareto-optimal iff there is no other feasible status set SS such that *objf*$(SS) >$ *objf*$(SS^\star)$.

In general, there could be zero, one, or many Pareto-optimal feasible status sets. In this case, we can choose one in several ways. One possibility is to choose any solution randomly—this is what is done in classical numerical optimization. However, additional options are also possible. We discuss these in Section VIII.

We investigated the complexity of the central problem of deciding whether a given status set is a Pareto-optimal feasible status set. We start with the following proposition, which establishes an upper-bound under reasonable conditions.

Proposition 1. *If the agent program, the integrity constraints, the action constraints, and the action predicate names are fixed, and* conc *and the objective functions can be computed in polynomial time, then deciding whether a given status set SS is a Pareto-optimal feasible status set is in co-NP.*

Proof. We first show that deciding whether a status set SS' is feasible can be done in polynomial time under the assumptions in the statement. Conditions 1)–5) of Definition 1 can be clearly verified in polynomial time. Condition 6) can be verified in polynomial time because the agent program is fixed (and thus, there is a polynomial number of ground instances of operating rules). Condition 7) can be verified in polynomial time because the action constrains are fixed. Condition 8) can be verified in polynomial time because *(i)* conc can be computed in polynomial time, *(ii)* checking constraint satisfaction can be done in polynomial time, since the integrity constrains are fixed.

We now show that the complementary problem, that is, deciding whether SS is *not* a Pareto-optimal feasible status set, is in NP. We first check whether SS is feasible; if not, then answer yes. As shown above this check can be done in polynomial time. If SS is feasible, then we guess a status set SS', and check that *(i)* SS' is feasible, and *(ii)* for all objective functions *objf*, *objf*$(SS') \geq$ *objf*(SS), and for some objective function *objf*, *objf*$(SS') >$ *objf*(SS). Check *(i)* can be done in polynomial time, as shown above. Check *(ii)* can be done in polynomial time because the objective functions can be computed in polynomial time. Also, SS' has polynomial size, since the actions' predicates are fixed. $\square$

We now turn our attention to the lower-bound and show that deciding whether a given status set is a Pareto-optimal feasible one is co-NP-hard. In particular, co-NP-hardness holds even if the agent program, the integrity constraints, the action constraints (whose set is indeed empty), the action predicate names, and conc are fixed, there is only one fixed objective function, and conc and the objective functions can be computed in polynomial time.

Theorem 1. *Deciding whether a given status set is a Pareto-optimal feasible status set is co-NP-hard.*

Proof. We reduce the NP-hard 3-colorability problem to the complement of our problem, that is, deciding whether a status set SS is *not* a Pareto-optimal feasible status set. An instance of 3-colorability is an undirected graph (V, E), for which it has to be decided whether there exists a *3-coloring*, that is, a way of assigning exactly one of three colors to every vertex in V so that no two adjacent (w.r.t. E) vertices have the same color. We derive an instance of the complement of our problem as follows. The initial state is $S_0 = \{\mathsf{vertex}(v) \mid v \in V\} \cup \{\mathsf{edge}(v, v') \mid (v, v') \in E\} \cup \{\mathsf{color}(c_1), \mathsf{color}(c_2), \mathsf{color}(c_3)\}$. The actions are as follows:

- For $v \in V$, we have action $\mathsf{dummycol}_a(v, c_1)$ with $Pre(\mathsf{dummycol}_a(v, c_1)) = true$, $Del(\mathsf{dummycol}_a(v, c_1)) = \emptyset$, and $Add(\mathsf{dummycol}_a(v, c_1)) = \{\mathsf{dummycol}(v, c_1), \mathsf{colored}(v)\}$.
- For $v \in V$, $c \in \{c_1, c_2, c_3\}$, action $\mathsf{coloring}_a(v, c)$ with $Pre(\mathsf{coloring}_a(v, c)) = true$, $Del(\mathsf{coloring}_a(v, c)) = \emptyset$, and $Add(\mathsf{coloring}_a(v, c)) = \{\mathsf{coloring}(v, c), \mathsf{colored}(v)\}$.
- For each $v \in V$, action $\mathsf{vertex}_a(v)$ with $Pre(\mathsf{vertex}_a(v)) = true$, $Del(\mathsf{vertex}_a(v)) = \emptyset$, and $Add(\mathsf{vertex}_a(v)) = \{\mathsf{vertex}_s(v)\}$.

The agent program contains $\mathbf{Do}\,\mathsf{vertex}_a(X) \leftarrow \mathsf{vertex}(X)$. The integrity constraints are:

$$\begin{array}{l}
\leftarrow \mathsf{coloring}(X, C_1) \,\&\, \mathsf{dummycol}(Y, C_2)\\
\leftarrow \mathsf{edge}(X, Y) \,\&\, \mathsf{coloring}(X, C) \,\&\, \mathsf{coloring}(Y, C)\\
\leftarrow \mathsf{coloring}(X, c_1) \,\&\, \mathsf{coloring}(X, c_2)\\
\leftarrow \mathsf{coloring}(X, c_1) \,\&\, \mathsf{coloring}(X, c_3)\\
\leftarrow \mathsf{coloring}(X, c_2) \,\&\, \mathsf{coloring}(X, c_3)\\
\mathsf{colored}(X) \leftarrow \mathsf{vertex}_s(X)
\end{array}$$

The set of action constraints is empty. We also have $\mathsf{conc}(A, S_t) = S_t \setminus (\bigcup_{\alpha \in A} Del(\alpha)) \cup \bigcup_{\alpha \in A} Add(\alpha)$ and *objf*$(SS) = |\{\mathbf{Do}\,\mathsf{coloring}_a(v, c) \in SS\}|$. The status set SS contains $\mathbf{Do}\,\mathsf{vertex}_a(v)$, $\mathbf{P}\,\mathsf{vertex}_a(v)$, $\mathbf{Do}\,\mathsf{dummycol}_a(v, c_1)$, $\mathbf{P}\,\mathsf{dummycol}_a(v, c_1)$, for each $v \in V$. We now show that (V, E) has a 3-coloring iff SS is *not* a Pareto-optimal feasible status set. First of all, we point out that SS is feasible and *objf*$(SS) = 0$, which can be easily verified.

($\Rightarrow$) Let $\phi : V \to \{c_1, c_2, c_3\}$ be a 3-coloring of (V, E). We first show that the following status set is feasible:

$$\begin{aligned}
SS' = \;& \textstyle\bigcup_{v \in V} \{\mathbf{Do}\,\mathsf{vertex}_a(v), \mathbf{P}\,\mathsf{vertex}_a(v)\} \cup\\
& \textstyle\bigcup_{v \in V} \{\mathbf{Do}\,\mathsf{coloring}_a(v, \phi(v)), \mathbf{P}\,\mathsf{coloring}_a(v, \phi(v))\}
\end{aligned}$$

Conditions 1)–4) of Definition 1 are clearly satisfied by SS'. Condition 5) is satisfied, as all action preconditions are trivially true. Condition 6) is satisfied since for each $\mathsf{vertex}(v)$ in S_t, $\mathbf{Do}\,\mathsf{vertex}_a(v)$ is included in SS'. Condition 7) is satisfied because there are no action constraints. Let us now discuss Condition 8). Notice that S_0 satisfies the ICs. We need to show that $S_1 = \mathsf{conc}(\{\alpha \,|\, \mathbf{Do}\alpha \in SS'\}, S_t)$ satisfies the ICs. By definition of conc, and the actions' Del and Add sets, $S_1 = S_0 \cup \{\mathsf{vertex}_s(v) \mid v \in V\} \cup \bigcup_{v \in V}\{\mathsf{coloring}(v, \phi(v)), \mathsf{colored}(v)\}$. Since ϕ is a 3-coloring, it can be easily verified that all ICs are satisfied by S_1. Hence, SS' is a feasible status set and $\mathit{objf}(SS') = |V|$. W.l.o.g. we can assume the original graph has at least one vertex and thus $\mathit{objf}(SS') > 1$, and thus SS is not Pareto-optimal.

($\Leftarrow$) Suppose (V, E) has no 3-coloring. We show that there is no feasible status set SS' containing at least one status atom of the form $\mathbf{Do}\,\mathsf{coloring}_a(v, c)$—which implies that SS is Pareto-optimal. Reasoning by contradiction, suppose SS' exists. In order for SS' to be feasible, it must satisfy Condition 6) of Definition 1, and thus SS' must include $\{\mathbf{Do}\,\mathsf{vertex}_a(v) \mid v \in V\}$. This means that the new state S_1 will include $\{\mathsf{vertex}_s(v) \mid v \in V\}$, as per definition of conc and the Add sets for $\mathsf{vertex}_a(v)$ actions. In order for S_1 to satisfy the last IC, S_1 must include $\{\mathsf{colored}(v) \mid v \in V\}$. Since SS' includes at least one status atom of the form $\mathbf{Do}\,\mathsf{coloring}_a(v, c)$, S_1 includes $\mathsf{coloring}(v, c)$, and thus SS' cannot include any status atom of the form $\mathbf{Do}\,\mathsf{dummycol}_a(v', c_1)$, because otherwise $\mathsf{dummycol}(v', c_1)$ would be in S_1 violating the first IC. Thus, the only way for S_1 to have an atom $\mathsf{colored}(v)$ for each vertex $v \in V$ is that SS' has at least one $\mathbf{Do}\,\mathsf{coloring}_a(v, c)$ status atom for each vertex $v \in V$. Notice that each status atom $\mathbf{Do}\,\mathsf{coloring}_a(v, c)$ yields the atom $\mathsf{coloring}(v, c)$ in S_1. In order for S_1 to satisfy the third to fifth ICs, S_1 must contain at most one $\mathsf{coloring}(v, c)$ atom for each vertex v. Thus, S_1 contains exactly one $\mathsf{coloring}(v, c)$ atom for each vertex v. Notice that S_1 must satisfy also the second IC. Now it is easy to see that the function assigning to each vertex v the color c iff $\mathsf{coloring}(v, c)$ belongs to S_1 is a 3-coloring, which is a contradiction. □

From the results above, we get the following corollary.

Corollary 1. *If the agent program, the integrity constraints, the action constraints, and the action predicate names are fixed, and* conc *and the objective functions can be computed in polynomial time, then deciding whether a given status set is a Pareto-optimal feasible status set is co-NP-complete.*

VI. Algorithms

In this section, we introduce several algorithms to compute Pareto-optimal feasible status sets.

First, we present a "helper" algorithm (used by all other algorithms) to compute the "closure" of a status set (Algorithm 1). Then, we propose a baseline algorithm that can be used with arbitrary sets of objective functions (Algorithm 2). Next, we develop exact algorithms for weakly/strongly anti-monotonic objective functions (Algorithms 3–4). These methods leverage anti-monotonicity to improve on the baseline. Their basic idea is to traverse up a lattice of status sets in a breadth-first fashion, where the lattice is defined w.r.t. set-inclusion (resp., set-inclusion of $\mathbf{Do}\alpha$ atoms) for weakly (resp., strongly) anti-monotonic objective functions. This strategy allows the algorithms to start from the "smallest" possibly feasible status sets, look for a Pareto-optimal feasible one, and move to bigger status sets only if needed.

A similar idea can be applied to weakly and strongly monotonic objective functions, but the lattice is traversed downwards starting from the "biggest" possibly feasible status sets. We found this strategy less effective compared to the anti-monotonic case, because the biggest status sets to start from may contain many contradictory status atoms (e.g., violating action constraints) and moving to smaller feasible ones might require traversing several levels of the lattice. For this reason, with weakly/strongly monotonic objective functions, in order to significantly improve on the baseline algorithm, we introduced heuristics leading to the two approximation algorithms presented in the following (Algorithms 5 and 6).

All algorithms in this section except for the "helper" one take as input: a state S_t, an agent program P, a set IC of integrity constraints, a set AC of action constraints, a conc function, a set OF of objective functions, and a set A of ground actions. Algorithms 5 and 6 have an additional input τ, which is used for the heuristic search and will be discussed later.

A. Helper Algorithm

The Closure algorithm (cf. Algorithm 1) takes as input a status set SS, a current state S_t, an agent program P, and a set DC of denial action constraints. The goal of the algorithm is to compute a status set that includes SS and satisfies Conditions 1)-6) of Definition 1, as well as Condition 7) w.r.t. denial action constraints only, if such a status set exists. If a status set is returned, it might not be feasible, as Condition 7) of Definition 1 w.r.t. definite action constraints, as well as the last condition of Definition 1, still need to be verified.

The algorithm first "closes" SS w.r.t. Conditions 1)–3) of Definition 1 (lines 1–6). It then checks if Conditions 4), 5), and 7) are all satisfied (lines 7–10). If any of them is not satisfied, then $\perp$ is returned. Otherwise, the algorithm iteratively enforces Condition 6) of Definition 1 (lines 11–26), thereby possibly deriving further ground status atoms. While doing so, the algorithms enforces Conditions 1)–3) of Definition 1 (lines 18–21) and checks that Conditions 4)–5) and Condition 7) (w.r.t. the denial action constraints in DC) of Definition 1 remain satisfied w.r.t. the ground status atoms that are being derived (lines 22–25)—once again, if any condition is violated, $\perp$ is returned, otherwise the algorithms keeps adding new ground status atoms until a fixpoint is reached and the resulting set is returned (line 27).

It is worth noting that every ground status atom derived by the algorithm must be in any status set SS' extending SS in order for SS' to be possibly feasible. A status set returned by the algorithm that satisfies also Condition 7) of Definition 1 w.r.t. all action constraints as well as Condition 8) is feasible.

The proposition below states an important property that will be leveraged by the algorithms introduced in the following.

Algorithm 1 Closure

```
Input: A status set $SS$, a state $S_t$, an agent program $P$, and
    a set $DC$ of denial action constraints.
Output: A status set or $\bot$.
1: for each $\mathbf{O}\alpha \in SS$ s.t. $\mathbf{P}\alpha \notin SS$ do
2:     Add $\mathbf{P}\alpha$ to $SS$.
3: for each $\mathbf{O}\alpha \in SS$ s.t. $\mathbf{Do}\alpha \notin SS$ do
4:     Add $\mathbf{Do}\alpha$ to $SS$.
5: for each $\mathbf{Do}\alpha \in SS$ s.t. $\mathbf{P}\alpha \notin SS$ do
6:     Add $\mathbf{P}\alpha$ to $SS$.
7: if there exists $\alpha$ s.t. $(i)$ $\{\mathbf{P}\alpha, \mathbf{F}\alpha\} \subseteq SS$ or $(ii)$ $\mathbf{P}\alpha \in SS$ and
   $Pre(\alpha)$ is false in $S_t$ then
8:     return $\bot$.
9: if $\{\alpha \mid \mathbf{Do}\alpha \in SS\}$ does not satisfy $DC$ then
10:    return $\bot$.
11: $SS' := SS$.
12: repeat
13:    $SS'' := SS'$.
14:    for each ground rule $r$ of $P$ do
15:        Let $r$ be $SA \leftarrow \chi \ \& \ SA_1 \ \& \ \dots \ \& \ SA_n$.
16:        if $\chi$ is true in $S_t$ and $\{SA_1, \dots, SA_n\} \subseteq SS'$ then
17:            Add $SA$ to $SS'$.
18:            if $SA = \mathbf{O}\alpha$ then
19:                Add $\mathbf{P}\alpha$ and $\mathbf{Do}\alpha$ to $SS'$.
20:            else if $SA = \mathbf{Do}\alpha$ then
21:                Add $\mathbf{P}\alpha$ to $SS'$.
22:            if there exists $\alpha$ s.t. $(i)$ $\{\mathbf{P}\alpha, \mathbf{F}\alpha\} \subseteq SS'$ or $(ii)$ $\mathbf{P}\alpha \in$
               $SS'$ and $Pre(\alpha)$ is false in $S_t$ then
23:                return $\bot$.
24:            if $\{\alpha \mid \mathbf{Do}\alpha \in SS'\}$ does not satisfy $DC$ then
25:                return $\bot$.
26: until $SS' = SS''$
27: return $SS'$.
```

Proposition 2. *Let LSS = Closure$(\emptyset, S_t, P, DC)$ for any status S_t, agent program P, and set of denial action constraints DC. If $LSS = \bot$, then there is no feasible status set. If $LSS \neq \bot$, every feasible status set (if any) contains LSS.*

Proof. When Closure is called with $SS = \emptyset$, lines 1–10 have no effect. Then, lines 11-27 are executed, enforcing Conditions 1)–3) and 6) of Definition 1 by possibly deriving new status atoms. Such status atoms must be necessarily contained in any feasible status set containing the empty set, and thus in every feasible status set (if any). Recall that lines 11-27 additionally check whether any of Conditions 4), 5), and 7) of Definition 1 is violated. If a status set violates any of such conditions, then every superset of it violates the same conditions. Thus, Closure returns $\bot$ when the set SS' of status atoms currently computed (which must be included in every feasible status set, if any) violates any of Conditions 4), 5), and 7) (which will be violated by every superset of SS'), that is, there is no feasible status set. If Closure returns a status set, the latter does not violate any of Conditions 4), 5), and 7) and must be contained in every feasible status set, if any. □

In the sequel, we use the following notation. For any program P, we use g_P (resp., χ_P, b_P) to denote the number of ground rules of P (resp., the maximum number of atoms in the condition χ of rules in P, the maximum number of status atoms of rules in P). For any set of constraints C, we use $||C||$ to denote the overall number of atoms in C. As customary, for any set X, we use $|X|$ to denote the cardinality of X. Finally, we use A to denote the set of all ground actions.

Proposition 3. *The worst-case time complexity of Algorithm 1 is $O(|A| \cdot g_P \cdot (\chi_P \cdot |S_t| + |A| \cdot (b_P + lg|A| + |S_t| + ||DC||)))$.*

B. Baseline Algorithm

We now introduce a baseline algorithm (POSS baseline, cf. Algorithm 2) to compute a Pareto-optimal feasible status set (if one exists) with an arbitrary set of objective functions.

Given a set A of actions, we define $SA(A) = \{Op\,\alpha \mid \alpha \in A$ and $Op \in \{\mathbf{F}, \mathbf{P}, \mathbf{O}, \mathbf{Do}\}\}$.

Algorithm 2 POSS baseline

```
Input: A state $S_t$, an agent program $P$,
    a set $IC$ of integrity constraints,
    a set $AC$ of action constraints, a conc function,
    a set $OF$ of objective functions, and
    a set $A$ of ground actions.
Output: A Pareto-optimal feasible status set or $\bot$.
1: Let $DC$ be the set of denial constraints in $AC$.
2: $LSS$ = Closure$(\emptyset, S_t, P, DC)$.
3: if $LSS = \bot$ then
4:     return $\bot$.
5: $\overline{A} := \{\alpha \mid \alpha \in A$ and $(Pre(\alpha)$ is false in $S_t$ or $\mathbf{F}\alpha \in LSS)\}$.
6: $\overline{SA} := \cup_{\alpha \in \overline{A}} \{\mathbf{Do}\alpha, \mathbf{O}\alpha, \mathbf{P}\alpha\}$.
7: $SA := SA(A) \setminus \overline{SA}$.
8: $\mathcal{S} = \emptyset$.
9: for each $SS$ s.t. $LSS \subseteq SS \subseteq SA$ do
10:    if $SS$ is a feasible status set then
11:        Add $SS$ to $\mathcal{S}$.
12: if $\mathcal{S} = \emptyset$ then
13:    return $\bot$.
14: else
15:    return a Pareto-optimal (w.r.t. $OF$) element of $\mathcal{S}$.
```

The algorithm first calls the Closure algorithm with the empty status set, the current state, the agent program, and the denial action constraints in AC, thereby getting LSS (lines 1–2). If LSS is $\bot$, then there is no feasible status set and the algorithm returns $\bot$ (lines 3–4). Otherwise, there might exist feasible status sets, and if any exists it has to contain LSS. For this reason, lines 1–4 will be replicated in all our algorithms reported in the following. Thus, the algorithm looks for feasible status sets that are a superset of LSS (lines 8–11), and if none exists $\bot$ is returned (lines 12-13), otherwise a Pareto-optimal one is returned (lines 14–15). Moreover, a simple pruning is applied when searching for feasible status sets containing LSS. The algorithm ignores status atoms that cannot be in any feasible status set (lines 5–7): these are the $\mathbf{Do}\alpha$, $\mathbf{O}\alpha$, and $\mathbf{P}\alpha$ status atoms for which $Pre(\alpha)$ is false in the current state (see Conditions 1)–3) and 5) of Definition 1) or $\mathbf{F}\alpha$ belongs to LSS (see Conditions 1)–4) of Definition 1). Such a pruning will be applied by all algorithms presented in the following as well.

Theorem 2. *Algorithm 2 correctly computes a Pareto-optimal feasible status set.*

Proof. By Proposition 2, if $LSS = \bot$ in line 3, then there is no feasible status set and the algorithm correctly returns $\bot$. Otherwise, by Proposition 2, LSS is a status set that must be contained in every feasible status set, if one exists. The algorithm looks for feasible status sets SS s.t. $LSS \subseteq SS \subseteq SA$, and returns a Pareto-optimal one among them, if at least one feasible status set has been found. So, to prove correctness, we need to show that no feasible status set is missed by the algorithm, that is, there is no feasible status set SS s.t. $SS \subsetneq LSS$ or $SS \supsetneq SA$. Proposition 2 implies that there cannot be any feasible status set SS s.t. $SS \subsetneq LSS$. Notice that each status atom $\mathbf{P}\alpha$ s.t. $Pre(\alpha)$ is false in S_t or $\mathbf{F}\alpha \in LSS$ cannot be included in any feasible status set. For such $\mathbf{P}\alpha$ status atoms, the status atoms $\mathbf{O}\alpha$ and $\mathbf{Do}\alpha$ cannot be included in any feasible status set too, because of Conditions 1) and 3) of Definition 1. Thus, lines 5–7 safely disregard the status atoms in $\overline{SA}$, as they cannot belong to any feasible status set, and hence there cannot be a feasible status set $SS \supsetneq SA$. □

Proposition 4. *The worst-case time complexity of Algorithm 2 is* $O(|A|^2 \cdot g_P \cdot ||DC|| + 2^{2|A|} \cdot f_{OF}(A) + 2^{|A|} \cdot (|A| \cdot lg|A| + |A| \cdot |S_t| + g_P \cdot (|S_t| \cdot \chi_P + |A| \cdot b_P) + |A| \cdot ||AC|| + |S_t| \cdot ||IC|| + f_{\mathsf{conc}}(|A|, |S_t|)))$*, where* f_{OF} *(resp.,* f_{conc}*) is the function measuring the worst-case time complexity of evaluating the objective functions in* OF *(resp.,* conc*).*

The numbers of cars and lanes affect number of rules in the program and the size of the constraints (g_P, $||IC||$, $||DC||$) as well as the number of actions ($|A|$). Such observations apply also to the other algorithms presented in the following.

C. Weakly and Strongly Anti-Monotonic Algorithms

We propose algorithms to compute Pareto-optimal feasible status sets in the presence of weakly (cf. Algorithm 3) and strongly (cf. Algorithm 4) *anti-monotonic* objective functions.

Let us start with Algorithm 3. The basic idea of the algorithm is to traverse a lattice (w.r.t. set-inclusion) of status sets where the bottom element is the set LSS computed in lines 1–2. In particular, the lattice is traversed upwards starting from LSS in a breadth-first fashion. In lines 1–6, the algorithm applies the same pruning discussed before for the baseline algorithm. Then, SA consists of the status atoms that might be added to LSS (line 7). In lines 8–18, the algorithm performs the aforementioned traversal of the lattice, one level at a time, starting from LSS, where each level is built by adding one status atom to each status set of the previous level (see lines 15–18). When a feasible status set exists in a level, a Pareto-optimal one is returned, otherwise the next level is considered. It is worth noting that each level is built only if needed and the lattice is not entirely materialized at once, which yields computational benefits in terms of both run time and memory usage. Eventually, if no feasible status set has been encountered, $\perp$ is returned (line 19).

Algorithm 3 POSS weakly-anti-monotonic

```
Input: A state $S_t$, an agent program $P$,
       a set $IC$ of integrity constraints,
       a set $AC$ of action constraints, a conc function,
       a set $OF$ of weakly anti-monotonic objective functions, and
       a set $A$ of ground actions.
Output: A Pareto-optimal feasible status set or $\perp$.
 1: Let $DC$ be the set of denial constraints in $AC$.
 2: $LSS = \mathsf{Closure}(\emptyset, S_t, P, DC)$.
 3: if $LSS = \perp$ then
 4:     return $\perp$.
 5: $\overline{A} := \{\alpha \mid \alpha \in A$ and $(Pre(\alpha)$ is false in $S_t$ or $\mathbf{F}\alpha \in LSS)\}$.
 6: $\overline{SA} := \cup_{\alpha \in \overline{A}} \{\mathbf{Do}\alpha, \mathbf{O}\alpha, \mathbf{P}\alpha\}$.
 7: $SA := SA(A) \setminus (\overline{SA} \cup LSS)$.
 8: $ToInspect := \{LSS\}$.
 9: while $ToInspect \neq \emptyset$ do
10:     $Candidates := ToInspect$.
11:     $ToInspect := \emptyset$.
12:     if $Candidates$ has a feasible status set then
13:         return a Pareto-optimal (w.r.t. $OF$) feasible status set of
            $Candidates$.
14:     else
15:         for each $Cand$ in $Candidates$ do
16:             for each $Op\,\alpha \in (SA \setminus Cand)$ do
17:                 if $(Cand \cup \{Op\,\alpha\}) \notin ToInspect$ then
18:                     Add $Cand \cup \{Op\,\alpha\}$ to $ToInspect$.
19: return $\perp$.
```

Theorem 3. *Algorithm 3 correctly computes a Pareto-optimal feasible status set.*

Proof. The same argument in the proof of Theorem 2 applies to lines 1–6 of Algorithm 3. Thus, the status atoms in $LSS \cup SA$ are the only ones that can possibly belong to a feasible status set. It is easy to see that (in lines 8–19) the algorithm starts from LSS and then iteratively considers bigger status sets, where at each iteration (of the **while** loop in lines 9–18) status sets that are incomparable w.r.t. set-inclusion are considered. At a generic iteration, if a feasible status is found that is Pareto-optimal among those considered in that iteration, then it must be Pareto-optimal also w.r.t. bigger status sets, because objective functions are weakly anti-monotonic. □

Algorithm 4 deals with strongly anti-monotonic objective functions, and behaves like Algorithm 3, except that the lattice is built w.r.t. set-inclusion of $\mathbf{Do}\alpha$ status atoms.

Theorem 4. *Algorithm 4 correctly computes a Pareto-optimal feasible status set.*

Proof. The same argument in the proof of Theorem 3 applies, noting that status sets are compared w.r.t. $\mathbf{Do}\alpha$ status atoms, because objective functions are strongly monotonic. □

The worst-case time complexity of Algorithms 3 and 4 is the one stated in Proposition 4, as in the worst case, $O(2^{|A|})$ candidate status sets still need to be inspected. While this is a theoretical analysis in the *worst case*, we will show in Section VII that Algorithms 3 and 4 indeed provide computational benefits over the baseline in practice.

D. Weakly and Strongly Monotonic Algorithms

In this section, we introduce approximation algorithms for weakly and strongly *monotonic* objective functions.

Let us start with Algorithm 5, which deals with weakly-monotonic objective functions. The basic idea is to start with the biggest "possibly feasible" status sets, and then move to smaller ones if needed (i.e., if no feasible status set has been found). Lines 1–5 are analogous to the ones of the algorithms discussed so far. In lines 6–14, the algorithm builds the biggest possibly feasible status sets to start from, applying

Algorithm 4 POSS strongly-anti-monotonic

```
Input: A state $S_t$, an agent program $P$,
    a set $IC$ of integrity constraints,
    a set $AC$ of action constraints, a conc function,
    a set $OF$ of strongly anti-monotonic objective functions, and
    a set $A$ of ground actions.
Output: A Pareto-optimal feasible status set or $\bot$.
 1: Let $DC$ be the set of denial constraints in $AC$.
 2: $LSS = \mathsf{Closure}(\emptyset, S_t, P, DC)$.
 3: if $LSS = \bot$ then
 4:     return $\bot$.
 5: $\overline{A} := \{\alpha \mid \alpha \in A$ and $(Pre(\alpha)$ is false in $S_t$ or $\mathbf{F}\alpha \in LSS)\}$.
 6: $\overline{SA} := \cup_{\alpha \in \overline{A}} \{\mathbf{Do}\alpha, \mathbf{O}\alpha, \mathbf{P}\alpha\}$.
 7: $SA := SA(A) \setminus (\overline{SA} \cup LSS)$.
 8: $SA\text{-}Do := \{\mathbf{Do}\alpha \mid \mathbf{Do}\alpha \in SA\}$.
 9: $SA\text{-}FPO := SA \setminus SA\text{-}Do$.
10: $ToInspect := \{LSS \cup X \mid X \subseteq SA\text{-}FPO\}$.
11: while $ToInspect \neq \emptyset$ do
12:     $Candidates := ToInspect$.
13:     $ToInspect := \emptyset$.
14:     if $Candidates$ has a feasible status set then
15:         return a Pareto-optimal (w.r.t. OF) feasible status set of $Candidates$.
16:     else
17:         for each $Cand$ in $Candidates$ do
18:             for each $\mathbf{Do}\alpha \in (SA\text{-}Do \setminus Cand)$ do
19:                 if $(Cand \cup \{\mathbf{Do}\alpha\}) \notin ToInspect$ then
20:                     Add $Cand \cup \{\mathbf{Do}\alpha\}$ to $ToInspect$.
21: return $\bot$.
```

Algorithm 5 POSS weakly-monotonic-approximate

```
Input: A state $S_t$, an agent program $P$,
    a set $IC$ of integrity constraints,
    a set $AC$ of action constraints, a conc function,
    a set $OF$ of weakly monotonic objective functions,
    a set $A$ of ground actions, and
    an integer $\tau$ (number of samples for randomly picking).
Output: A feasible status set or $\bot$.
 1: Let $DC$ be the set of denial constraints in $AC$.
 2: $LSS = \mathsf{Closure}(\emptyset, S_t, P, DC)$.
 3: if $LSS = \bot$ then
 4:     return $\bot$.
 5: $\overline{A} := \{\alpha \mid \alpha \in A$ and $(Pre(\alpha)$ is false in $S_t$ or $\mathbf{F}\alpha \in LSS)\}$.
 6: $ToInspect := \{LSS \cup \{\mathbf{F}\alpha \mid \alpha \in \overline{A}\}\}$.
 7: for each $\alpha \in (A \setminus \overline{A})$ do
 8:     $Tmp := \emptyset$.
 9:     for each $SS \in ToInspect$ do
10:         if $\mathbf{P}\alpha \notin SS$ then
11:             Add $SS \cup \{\mathbf{F}\alpha\}$ to $Tmp$.
12:         if $\mathbf{F}\alpha \notin SS$ then
13:             Add $SS \cup \{\mathbf{O}\alpha, \mathbf{Do}\alpha, \mathbf{P}\alpha\}$ to $Tmp$.
14:     $ToInspect := Tmp$.
15: $Done := \emptyset$.
16: while $ToInspect \neq \emptyset$ do
17:     $Candidates := random(\tau, ToInspect)$.
18:     $ToInspect := ToInspect \setminus Candidates$.
19:     if $Candidates$ has a feasible status set then
20:         return a Pareto-optimal (w.r.t. OF) feasible status set of $Candidates$.
21:     else
22:         for each $Cand$ in $Candidates$ do
23:             for each $Op\,\alpha \in Cand$ do
24:                 if $Op\,\alpha \notin LSS$ and $(Cand \setminus \{Op\,\alpha\}) \notin ToInspect$ and $(Cand \setminus \{Op\,\alpha\}) \notin Done$ then
25:                     if $Op = \mathbf{O}$ or $Op = \mathbf{F}$ then
26:                         Add $Cand \setminus \{Op\,\alpha\}$ to $ToInspect$.
27:                     if $Op = \mathbf{Do}$ and $\mathbf{O}\alpha \notin Cand$ then
28:                         Add $Cand \setminus \{Op\,\alpha\}$ to $ToInspect$.
29:                     if $Op = \mathbf{P}$ and $\mathbf{O}\alpha \notin Cand$ and $\mathbf{Do}\alpha \notin Cand$ then
30:                         Add $Cand \setminus \{Op\,\alpha\}$ to $ToInspect$.
31:         Add $Candidates$ to $Done$.
32: return $\bot$.
```

different pruning strategies that rule out status sets that are not feasible for sure. First, the algorithm rules out status atoms of the form $\mathbf{O}\alpha$, $\mathbf{Do}\alpha$, and $\mathbf{P}\alpha$ for which $Pre(\alpha)$ is not satisfied in the current state or $\mathbf{F}\alpha$ belongs to LSS—moreover, for such actions α, all $\mathbf{F}\alpha$ status atoms are included, as they will not conflict for sure with any $\mathbf{O}\alpha$, $\mathbf{Do}\alpha$, or $\mathbf{P}\alpha$ status atom (line 6). Second, for actions α not satisfying the aforementioned conditions, to construct the biggest status sets, either $\{\mathbf{F}\alpha\}$ or $\{\mathbf{O}\alpha, \mathbf{Do}\alpha, \mathbf{P}\alpha\}$ is considered (lines 7–14) to avoid status sets that would not be feasible. The **while** loop in lines 16–31 starts from the biggest status sets and moves to smaller ones if no feasible one has been found. At each iteration, only τ (randomly picked) status sets from $ToInspect$ are considered (see lines 17–18), where τ is an additional input of the algorithm. The status sets in $ToInspect$ that are not chosen by the random sampling are still left in $ToInspect$ for later inspection. This allows the algorithm to move faster to lower levels of the status set lattice, which pays off in terms of running time, as we show in our experimental evaluation. Of course, the algorithm might return sub-optimal feasible status sets, because when a set of feasible status sets is considered and a Pareto-optimal one is determined among them (lines 19–20), some other better feasible status sets might have been ignored (not being chosen by the random sampling).

Smaller status sets are built from the current ones by deleting a single status atom (lines 23–30), following the following criteria. A status atom of the form $\mathbf{Do}\alpha$ is deleted from a status set if the latter does not contain $\mathbf{O}\alpha$ (lines 27–28), because otherwise the deletion would yield a non-feasible status set—see Condition 2) of Definition 1. Likewise, a status atom of the form $\mathbf{P}\alpha$ is deleted from a status set if the latter contains neither $\mathbf{O}\alpha$ nor $\mathbf{Do}\alpha$ (lines 29–30), because otherwise the deletion would yield a non-feasible status set—see Conditions 1) and 3) of Definition 1. A status atom of the form $\mathbf{O}\alpha$ or $\mathbf{F}\alpha$ is deleted without checking further conditions (lines 25–26), as their deletion does not yield violations of Conditions 1)–4) of Definition 1. The algorithm returns $\bot$ if no feasible status set is eventually found (line 32).

Let us consider now Algorithm 6. The algorithm starts from "possibly feasible" status sets containing as many $\mathbf{Do}\alpha$ status atoms as possible, which are collected into $ToInspect$ (lines 1–9). The algorithm leverages the following ideas discussed before: it moves to smaller status sets if no feasible one has been found; it applies the sampling approach of Algorithm 5; it reduces the number of status sets to be considered when initializing $ToInspect$ (in lines 7–9) and when a lower level of the lattice has to be built (see lines 18–25).

The worst-case time complexity of Algorithms 5 and 6 is the one stated in Proposition 4, as in the worst case, $O(2^{|A|})$ candidate status sets still need to be inspected. While this is a theoretical analysis in the *worst case*, Section VII will show that Algorithms 5 and 6 provide computational benefits over the baseline in practice.

Algorithm 6 POSS strongly-monotonic-approximate

```
Input: A state $S_t$, an agent program $P$,
       a set $IC$ of integrity constraints,
       a set $AC$ of action constraints, a conc function,
       a set $OF$ of strongly monotonic objective functions,
       a set $A$ of ground actions, and
       an integer $\tau$ (number of samples for randomly picking).
Output: A feasible status set or $\perp$.
1:  Let $DC$ be the set of denial constraints in $AC$.
2:  $LSS = \mathsf{Closure}(\emptyset, S_t, P, DC)$.
3:  if $LSS = \perp$ then
4:      return $\perp$.
5:  $\overline{A} := \{\alpha \mid \alpha \in A$ and $(Pre(\alpha)$ is false in $S_t$ or $\mathbf{F}\alpha \in LSS)\}$.
6:  $A' := A \setminus \overline{A}$.
7:  $SA\text{-}DPO := \bigcup_{\alpha \in A'} \{\mathbf{Do}\alpha, \mathbf{P}\alpha, \mathbf{O}\alpha\}$.
8:  $SA\text{-}F := \{\mathbf{F}\alpha \mid \alpha \in \overline{A}\}$.
9:  $ToInspect := \{LSS \cup SA\text{-}DPO \cup X \mid X \subseteq SA\text{-}F\}$.
10: $Done := \emptyset$.
11: while $ToInspect \neq \emptyset$ do
12:     $Candidates := random(\tau, ToInspect)$.
13:     $ToInspect := ToInspect \setminus Candidates$.
14:     if $Candidates$ has a feasible status set then
15:         return a Pareto-optimal (w.r.t. $OF$) feasible status set of $Candidates$.
16:     else
17:         for each $Cand$ in $Candidates$ do
18:             for each $\mathbf{Do}\alpha \in Cand$ do
19:                 if $\mathbf{Do}\alpha \notin LSS$ then
20:                     if $(Cand \setminus \{\mathbf{Do}\alpha, \mathbf{P}\alpha, \mathbf{O}\alpha\}) \notin ToInspect$ and
                        $(Cand \setminus \{\mathbf{Do}\alpha, \mathbf{P}\alpha, \mathbf{O}\alpha\}) \notin Done$ then
21:                         Add $Cand \setminus \{\mathbf{Do}\alpha, \mathbf{P}\alpha, \mathbf{O}\alpha\}$ to $ToInspect$.
22:                     if $(Cand \setminus \{\mathbf{Do}\alpha, \mathbf{O}\alpha\}) \notin ToInspect$ and $(Cand \setminus$
                        $\{\mathbf{Do}\alpha, \mathbf{O}\alpha\}) \notin Done$ then
23:                         Add $Cand \setminus \{\mathbf{Do}\alpha, \mathbf{O}\alpha\}$ to $ToInspect$.
24:                     if $((Cand \cup \{\mathbf{F}\alpha\}) \setminus \{\mathbf{Do}\alpha, \mathbf{P}\alpha, \mathbf{O}\alpha\}) \notin ToInspect$
                        and $((Cand \cup \{\mathbf{F}\alpha\}) \setminus \{\mathbf{Do}\alpha, \mathbf{P}\alpha, \mathbf{O}\alpha\}) \notin Done$
                        then
25:                         Add $(Cand \cup \{\mathbf{F}\alpha\}) \setminus \{\mathbf{Do}\alpha, \mathbf{P}\alpha, \mathbf{O}\alpha\}$ to
                            $ToInspect$.
26:         Add $Candidates$ to $Done$.
27: return $\perp$.
```

VII. Experimental Assessment

We varied the parameters reported in Table I, where the default value we fixed for a parameter when varying another parameter is highlighted in bold.

TABLE I
Varying parameters (default values in bold).

Parameter	*Values*
Number of red cars	$\{1, \mathbf{10}, 20, 30\}$
Number of orange cars	$\{1, \mathbf{10}, 20, 30\}$
Number of green cars	$\{1, \mathbf{10}, 20, 30\}$
Number of lanes	$\{\mathbf{6}, 8, 10\}$ (plus exit lane)
Highway length	$\{40, 60, \mathbf{80}, 100\}$

In addition:

- We randomly picked the initial position of each car, ensuring that (i) two cars are not in the same cell and (ii) all initial positions are before the 10th cell.
- We fixed the speed of each car as follows:
  - For red cars, randomly picking from $\{1, 2, 3\}$.
  - For green cars: randomly picking from $\{1, 2\}$.
  - For orange cars: equal to 1.
- We randomly picked the destination of each car from $\{A, B, C, D, E\}$.
- We fixed the number of exits to a tenth of the highway length, with the first exit positioned at the 10th cell and with a distance of 10 cells between two consecutive exits.
- We randomly picked the destinations associated with each exit from $\{A, B, C, D, E\}$, ensuring that each destination has at least one associated exit.
- For the approximation algorithms, τ was set to 10 (early experiments showed this to be a good value).
- We used three objective functions: *Lane Shift Penalty* (LSP), which penalizes lane shifts in a status set, *Exit Miss Penalty* (EMP), which penalizes a status set that makes cars miss the exit (in two time steps), and *Change Speed Penalty* (CSP), which penalizes a status set that does not speed up when the exit is too far or does not slow down when the exit is very close. All objective functions are weakly and strongly anti-monotonic; as weakly and strongly monotonic objective functions, we used the same ones but with a flipped sign.

All experiments were performed on a machine with 36 Intel Core i9-10980XE CPUs, 256GB RAM, running Ubuntu 18.04.

A. Runtime

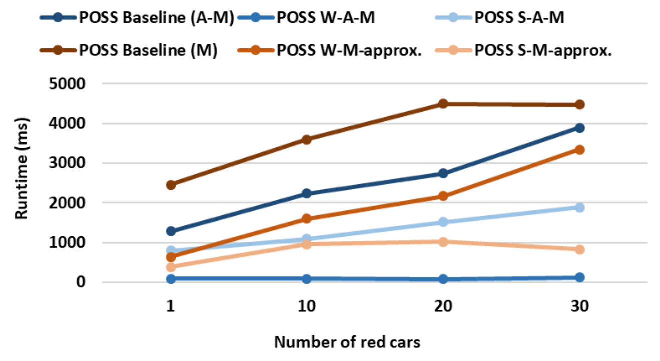
Figure 3 reports the average time needed to compute a POSS when varying the number of cars of each color. In all the figures, (i) POSS Baseline (M) and POSS Baseline (A-M) correspond to Baseline using monotonic and anti-monotonic objective functions, respectively, (ii) orange lines represent algorithms using monotonic objective functions, and (iii) blue lines represent algorithms using anti-monotonic objective functions. The results show that the runtime of POSS W-A-M is the best, followed by POSS S-M-approx. As POSS W-A-M is an exact algorithm, this suggests that we should try to convert true objective functions into similar weakly anti-monotonic ones.

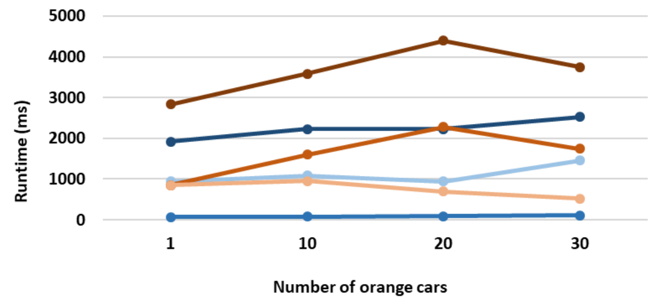
Figures 4 and 5 report the average time needed to compute a POSS when varying the number of lanes and highway length. Again, we see that POSS W-A-M is the fastest approach.

In all, we compared our 4 proposed algorithms with Baseline (i) in 48 cases when varying the number of cars of each color (4 algorithms $\times$ 4 parameter values $\times$ 3 colors), (ii) in 12 cases when varying the number of lanes (4 algorithms $\times$ 3 parameter values), and (iii) in 16 cases when varying the highway length (4 algorithms $\times$ 4 parameter values). The results show that (i) our proposed algorithms are faster than Baseline in all of the 76 cases and (ii) on average, our proposed algorithms run much faster than Baseline—see Table II, where the performance gain of each algorithm Alg is computed as $1 - \frac{time(\mathsf{Alg})}{time(\mathsf{Baseline})}$.

B. Solution Quality for Approximate Algorithms

We also assessed the quality of the solutions computed by our approximate algorithms POSS W-M-approx. and





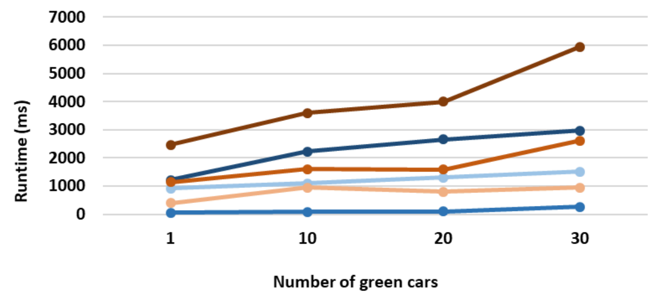


Fig. 3. Runtimes obtained when varying the number of cars of each color.

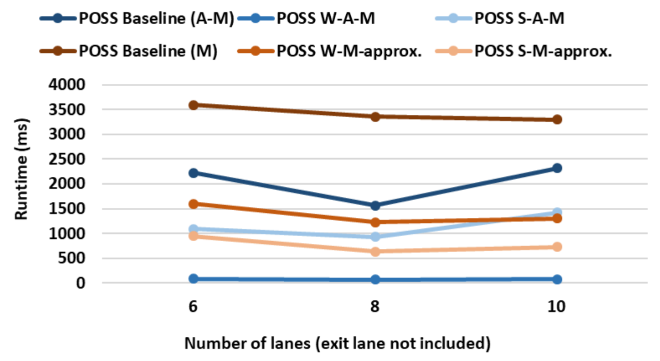


Fig. 4. Runtimes obtained when varying the number of lanes.

TABLE II
AVERAGE PERFORMANCE GAIN VS. BASELINE.

Varying parameter	POSS W-A-M	POSS S-A-M	POSS W-M-approx.	POSS S-M-approx.
# red cars	95.98%	46.31%	51.58%	79.18%
# orange cars	96.02%	50.60%	56.78%	78.50%
# green cars	94.76%	43.90%	56.47%	80.28%
# lanes	96.17%	43.51%	59.77%	77.52%
Highw. len.	95.98%	60.61%	48.86%	80.90%

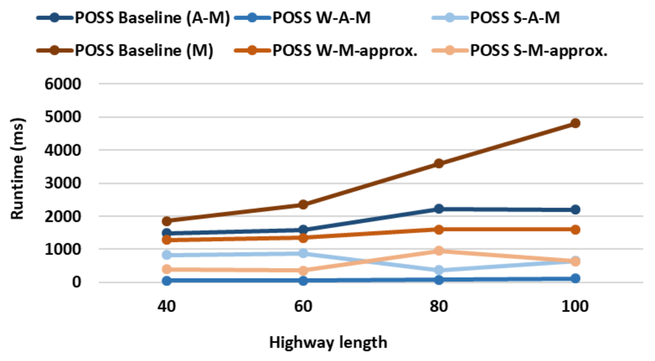


Fig. 5. Runtimes obtained when varying the highway length.

POSS S-M-approx. Inverted generational distance (IGD) and hypervolume (HV) are two popular approaches for measuring the solution quality of approximate algorithms for multi-objective problems [38]. IGD measures the distance between the objective values obtained by the approximate algorithm and the values in the Pareto front (i.e., the set of objective values corresponding to a set of Pareto-optimal feasible status sets), and HV measures the diversity and convergence by calculating the volume between the objective values obtained from the approximate algorithms and specified reference points. Here, our objective functions are weakly or strongly anti-monotonic/monotonic, and the optimal objective values do not vary much. Therefore, we use the IGD approach to measure the quality of the solutions computed by our approximate algorithms. Instead of simply calculating the distance, we looked at the relative quality $\frac{value(objf,\mathsf{Alg})}{value(objf,\mathsf{Baseline})}$, where $value(objf,\mathsf{Alg})$ is the value obtained for objective function $objf$ using Alg. Table III reports the average values of the objective functions introduced above, when varying the various parameters. The results show that both POSS W-M-approx. and POSS S-M-approx. are able to provide good quality solutions. The average relative quality using POSS W-M-approx. ranged from 62.9% to 66.5% for LSP, from 61.8% to 74.3% for EMP, and from 38.2% to 77.4% for CSP. POSS S-M-approx. provided even better results—its average relative quality ranged from 83.6% to 98.4% for LSP, from 93.1% to 99.0% for EMP, and from 76.5% to 98.9% for CSP. Compared to a fully random approximation algorithm, the overall relative quality provided by POSS W-M-approx. and POSS S-M-approx. was much higher (on average, 34.6% for LSP, 32.9% for EMP, and 42.6% for CSP).

TABLE III
AVERAGE RELATIVE QUALITY VS. BASELINE.
LSP IS LANE SHIFT PENALTY, EMP IS EXIT MISS PENALTY, AND CSP IS CHANGE SPEED PENALTY.

Varying parameter	POSS W-M-approx.			POSS S-M-approx.		
	LSP	EMP	CSP	LSP	EMP	CSP
# red cars	64.0%	67.9%	77.4%	95.0%	98.1%	92.1%
# orange cars	66.2%	70.5%	75.3%	95.9%	99.0%	97.9%
# green cars	63.0%	61.8%	76.6%	83.6%	93.1%	76.5%
# lanes	62.9%	74.3%	60.0%	96.3%	97.5%	98.9%
Highw. len.	66.5%	64.4%	38.2%	98.4%	99.0%	88.5%

VIII. Choosing an Optimal Feasible Status Set

There can be situations where the Pareto frontier contains many Pareto-optimal status sets $PF = \{SS_1, \ldots, SS_n\}$. When this happens, the agent in question must choose one of these status sets even though all of these are deemed optimal according to the set of objective functions that were explicitly stated. In this case, many solutions are possible. We briefly discuss these below.

Random Choice. One possibility is for the agent to randomly choose one of the SS_j's and take the actions articulated therein. This method is fast and may be appropriate in cases where the agent needs to act very quickly and a near real-time choice must be made.

Weighted Objective Functions. [39], [40] have suggested that a Pareto-optimal solution be chosen according to weights that the system designer associates with each objective function. In this case, we associate a score with each SS_j in the Pareto frontier which is set to a linear combination (using the weights) of each objective function value. The SS_j with the highest score is then chosen.

Clustering-based Approaches. The clustering-based approach [40], [41], [42], [43] is also an important approach for selecting Pareto-optimal solutions. [41], [42] develop theories and procedures for selecting and clustering multiple criteria solutions. They proposed that the mutually exclusive clusters are determined by (i) the similarities between the solutions, and (ii) the decision-maker's preference structure. The procedures for making a decision include (i) generating optimal solutions, (ii) clustering solutions based on their similarities, and (iii) selecting one or more solutions from each cluster. Specifically, [41] used artificial neural networks (ANN) with variable weights for clustering and then feedforward ANN for selecting the best solution for each cluster. In their procedures [41], [42], the decision-maker is actively involved by comparing and contrasting solutions. [43] extended the clustering approaches formalizing the concept of k representative points of the Pareto front, where Pareto optimal solutions are clustered, and then the Pareto frontier is divided into k clusters. The k-means algorithms are used in [40], [43] for clustering. Recently, a graph-theoretical clustering approach was proposed for finding a reduced set of Pareto optimal solutions [44], where they construct a contact network by mapping each point in the objective space to a node, and connecting nodes that are within a certain distance of each other. One way to use the idea of clustering is to use an off-the-shelf clustering algorithm to cluster PF. Within a cluster $CL_h \subseteq PF$, we choose the status set $SS[h] \in CL_h$ that minimizes the distance to the other members of the cluster (according to a selected distance metric). Such a status set would be like a pseudo-centroid for that cluster. We can then create a graph whose nodes are these pseudo-centroids and whose edges are labeled according to the distance metric and choose the pseudo-centroid with the highest centrality (e.g., betweenness centrality [45] or Pagerank [46]).

IX. Limitations and Future Work

We now describe a few limitations of our work that can also lead to potential future work.

Scalability. In the real-world, there can be thousands of agents (e.g., thousands of cars on a single highway or road at a given point in time) and the responses to the actions of other cars has to be done at lightning speed. While the experiments show that POSS can be solved in 200 milliseconds to 1 second in several cases, there are some important cases where the computation time can be a few seconds. Fast approximation algorithms that provide solutions within milliseconds, yet are guaranteed to be within some approximation error bound of the optimal solution, need to be developed.

Scalable Choice of a Pareto-Optimal Status Set. One of the strengths of this paper is that we can find a Pareto-Optimal Status Set without computing the entire Pareto frontier. Though we have outlined some methods to choose from a Pareto frontier in Section VIII, it is important to adapt our proposed algorithms to find such Pareto-Optimal Status Sets without computing the entire Pareto frontier as that could compromise scalability. We also need to look at methods to extend the approximation algorithms mentioned above to this case.

Error-Tolerance. When multiple agents are operating in the real world, there will be noise and errors, e.g., errors due to sensor malfunction and/or due to communication latency or dropped packets between agents. What does it mean for a Pareto-Optimal Feasible Status Set to be robust to some kind of noise or error? How should our algorithms be changed in order to achieve such robustness without compromising scalability? It is critical to investigate this question further.

Long-Horizon Decision Making. Our proposed approach can effectively find a POSS for the next time step. It is worth investigating how to extend our approach for long-horizon decision making to improve the overall quality of the solution, considering potential actions of other agents, but without compromising scalability.

X. Conclusions

In this paper, we have developed the concept of a multi-agent system in which multiple agents each try to optimize multiple objectives in accordance with an input set of behavioral models and objectives. We specified the behavioral constraints in a high-level deontic logic, so that users and application developers can express their desired logical constraints easily in symbolic form, while simultaneously expressing their objective functions numerically. Our agents can work with *any* behavior model expressed in the deontic logic used here and any set of objective functions.

This paper makes several novel contributions. It is the first paper to consider multiple objective functions when deciding what actions a deontic logic agent should take. Second, we are the first to show co-NP-hardness of deciding whether a given status set is a POSS. Third, we are the first to develop (multiple) algorithms for solving the POSS problem both exactly as well as approximately under varying assumptions on the form of the objective functions and conducted an extensive set of experiments.

While there are opportunities for future research as discussed in Section IX, our work represents a first contribution to the science that integrates deontic logic for high level reasoning and Pareto optimization methods for lower-level reasoning, which can be applied in several real-world applications involving multiple agents.

Acknowledgments

We thank the reviewers for their very helpful comments. Subrahmanian was partly funded by ARO Grant No. W911NF2320240 and ran some experiments using equipment funded by ONR Grant No. N00014-20-1-2407. Jeong and Quattrini acknowledge NSF grants 1923004 and 2144624. Molinaro and Pugliese acknowledge support from Project SERICS (PE00000014) funded under the MUR National Recovery and Resilience Plan by the European Union – NextGenerationEU. Zhang is supported by the InnoHK Fund.

Appendix: Proofs

Proof of Proposition 3. In the following, the worst-case time complexity is always understood. Lines 1–4 can be executed as follows: the status atoms in SS are sorted by their action α, and then the resulting sorted list is scanned checking the condition of the **for each** loops for each traversed element—checking such a condition for a single element can now be done in constant time, since there is a constant number of status atoms with the same action. Assuming that the addition of a new element to SS takes constant time (e.g., using a list), the overall time taken by lines 1–4 is $O(|SS| \cdot lg|SS|)$. The same reasoning applies to lines 5–6, even though the *updated* set SS needs to be traversed, whose cardinality is at most three times the cardinality of the original set SS, and thus the overall time taken by lines 5–6 is $O(|SS| \cdot lg|SS|)$ too.

On line 7, condition (i) can be checked in $O(|SS| \cdot lg|SS|)$ time (again, by first sorting SS as discussed above), while condition (ii) can be checked in $O(|SS| \cdot |S_t|)$ time (here we are considering the size of $Pre(\alpha)$ to be a constant, as the set of actions is fixed). Line 8 takes constant time.

Line 9 takes $O(|SS| \cdot ||DC||)$ time. Lines 10–11 take constant time.

We now consider lines 12–26, which consist of two nested loops. The number of times lines 15–25 are executed is $O(|A| \cdot g_P)$, because the outer loop can make at most $|A|$ iterations, and the inner loop clearly makes g_P iterations. Let us focus on the complexity of lines 15–25 (when executed once). Line 15 takes constant time. Line 16 takes $O(\chi_P \cdot |S_t| + b_P \cdot |SS'|)$ time. Lines 17–21 take constant time (again, here we consider an addition to SS' to take constant time). On line 22, condition (i) can be checked in $O(|SS'| \cdot lg|SS'|)$ time, while condition (ii) can be checked in $O(|SS'| \cdot |S_t|)$ time. Line 23 takes constant time. Line 24 takes $O(|SS'| \cdot ||DC||)$ time. Line 25 takes constant time. So, the overall time complexity of lines 12–26 is $O(|A| \cdot g_P \cdot (\chi_P \cdot |S_t| + b_P \cdot |SS'| + |SS'| \cdot lg|SS'| + |SS'| \cdot |S_t| + |SS'| \cdot ||DC||))$, which can be rewritten as $O(|A| \cdot g_P \cdot (\chi_P \cdot |S_t| + |SS'| \cdot (b_P + lg|SS'| + |S_t| + ||DC||)))$. Notice that $|SS'|$ is $O(|A|)$. Thus, the overall time complexity of lines 12–26 can be rewritten as $O(|A| \cdot g_P \cdot (\chi_P \cdot |S_t| + |A| \cdot (b_P + lg|A| + |S_t| + ||DC||)))$.

Line 27 takes constant time.

From the analysis above, the worst-case time complexity of Algorithm 1 is $O(|A| \cdot g_P \cdot (\chi_P \cdot |S_t| + |A| \cdot (b_P + lg|A| + |S_t| + ||DC||)))$. $\square$

Proof of Proposition 4. In the following, the worst-case time complexity is always understood. The worst-case time complexity of line 2 is as per Proposition 3. Lines 3–4 take constant time. Lines 5–7 take $O(|A| \cdot (|S_t| + g_P))$ time, since the cardinality of LSS is $O(g_P)$. Notice that $|SA|$ is $O(|A|)$. Line 8 takes constant time. Checking whether a status set SS is feasible as per Definition 1 takes $O(|SS| \cdot lg|SS| + |SS| \cdot |S_t| + g_P \cdot (|S_t| \cdot \chi_P + |SS| \cdot b_P) + |SS| \cdot ||AC|| + |S_t| \cdot ||IC|| + f_{\mathsf{conc}}(|SS|, |S_t|))$ time. Lines 9–11 take $O(2^{|A|} \cdot (|A| \cdot lg|A| + |A| \cdot |S_t| + g_P \cdot (|S_t| \cdot \chi_P + |A| \cdot b_P) + |A| \cdot ||AC|| + |S_t| \cdot ||IC|| + f_{\mathsf{conc}}(|A|, |S_t|)))$ time, since, for any status set SS s.t. $LSS \subseteq SS \subseteq SA$, we have $|SS| = O(|A|)$. Lines 12–13 take constant time. Lines 14–15 take $O(2^{2^{|A|}} \cdot f_{OF}(|A|))$. From the analysis above, the overall (worst-case) time complexity of Algorithm 2 is $O(|A|^2 \cdot g_P \cdot ||DC|| + 2^{2^{|A|}} \cdot f_{OF}(A) + 2^{|A|} \cdot (|A| \cdot lg|A| + |A| \cdot |S_t| + g_P \cdot (|S_t| \cdot \chi_P + |A| \cdot b_P) + |A| \cdot ||AC|| + |S_t| \cdot ||IC|| + f_{\mathsf{conc}}(|A|, |S_t|)))$, where $|A|^2 \cdot g_P \cdot ||DC||$ is the part of the complexity of Algorithm 1 (see line 2) that is not dominated by the complexity of the rest of Algorithm 2. $\square$

References

[1] M. Dikmen and C. M. Burns, "Autonomous driving in the real world: Experiences with tesla autopilot and summon," in *AutomotiveUI Conference*, 2016.

[2] D. Lee, H. Kim, Y. Choi, and J. Kim, "Development of autonomous operation agent for normal and emergency situations in nuclear power plants," in *ICSRS Conference*, 2021.

[3] T. Drew and M. Gini, "Implantable medical devices as agents and part of multiagent systems," in *AAMAS Conference*, 2006.

[4] D. Føllesdal and R. Hilpinen, "Deontic logic: An introduction," in *Deontic logic: Introductory and systematic readings*. Springer, 1971, vol. 33, pp. 1–35.

[5] M. Olszewski, X. Parent, and L. Van der Torre, "Permissive and regulative norms in deontic logic," *Journal of Logic and Computation*, p. exad024, 2023.

[6] T. Eiter, V. Subrahmanian, and G. Pick, "Heterogeneous active agents, i: Semantics," *Artificial Intelligence*, vol. 108, no. 1-2, pp. 179–255, 1999.

[7] T. Eiter, V. Subrahmanian, and T. J. Rogers, "Heterogeneous active agents, iii: Polynomially implementable agents," *Artificial Intelligence*, vol. 117, no. 1, pp. 107–167, 2000.

[8] V. Subrahmanian, P. Bonatti, J. Dix, T. Eiter, S. Kraus, R. Ross, F. Ozcan, and J. Dix, *Heterogeneous agent systems*. MIT press, 2000.

[9] B. Stroe, V. Subrahmanian, and S. Dasgupta, "Optimal status sets of heterogeneous agent programs," in *AAMAS Conference*, 2005.

[10] P. M. Pardalos, A. Migdalas, and L. Pitsoulis, *Pareto optimality, game theory and equilibria*. Springer Science & Business Media, 2008, vol. 17.

[11] D. Rönnedal, *An introduction to deontic logic*. CreateSpace Independent Publishing Platform, 2010.

[12] Y. Cai, H. Zhang, H. Su, J. Zhang, and Q. He, "The bipartite edge-based event-triggered output tracking of heterogeneous linear multiagent systems," *IEEE Trans. Cybern.*, vol. 53, no. 2, pp. 967–978, 2023.

[13] H. Shi, M. Wang, and C. Wang, "Leader-follower formation learning control of discrete-time nonlinear multiagent systems," *IEEE Trans. Cybern.*, vol. 53, no. 2, pp. 1184–1194, 2023.

[14] Y. Hao, L. Liu, and G. Feng, "Event-triggered cooperative output regulation of heterogeneous multiagent systems under switching directed topologies," *IEEE Trans. Cybern.*, vol. 53, no. 2, pp. 1026–1038, 2023.

[15] M. Hjelmblom and J. Odelstad, "jDALMAS: A java/prolog framework for deontic action-logic multi-agent systems," in *KES-AMSTA Symposium*, 2009.

[16] L. Lindahl and J. Odelstad, “Normative positions within an algebraic approach to normative systems,” *Journal of Applied Logic*, vol. 2, no. 1, pp. 63–91, 2004.

[17] R. Calegari, G. Ciatto, V. Mascardi, and A. Omicini, “Logic-based technologies for multi-agent systems: Summary of a systematic literature review,” in *AAMAS Conference*, 2021.

[18] Y. Liu, N. Zhu, and M. Li, “Solving many-objective optimization problems by a pareto-based evolutionary algorithm with preprocessing and a penalty mechanism,” *IEEE Trans. Cybern.*, vol. 51, no. 11, pp. 5585–5594, 2021.

[19] J. J. Liang, K. Qiao, K. Yu, B. Qu, C. Yue, W. Guo, and L. Wang, “Utilizing the relationship between unconstrained and constrained pareto fronts for constrained multiobjective optimization,” *IEEE Trans. Cybern.*, vol. 53, no. 6, pp. 3873–3886, 2023.

[20] L. Ma, M. Huang, S. Yang, R. Wang, and X. Wang, “An adaptive localized decision variable analysis approach to large-scale multiobjective and many-objective optimization,” *IEEE Trans. Cybern.*, vol. 52, no. 7, pp. 6684–6696, 2022.

[21] L. Ma, N. Li, Y. Guo, X. Wang, S. Yang, M. Huang, and H. Zhang, “Learning to optimize: Reference vector reinforcement learning adaption to constrained many-objective optimization of industrial copper burdening system,” *IEEE Trans. Cybern.*, vol. 52, no. 12, pp. 12 698–12 711, 2022.

[22] Q. Zhang and H. Li, “MOEA/D: A multiobjective evolutionary algorithm based on decomposition,” *IEEE Trans. Evol. Comput.*, vol. 11, no. 6, pp. 712–731, 2007.

[23] K. Deb and H. Jain, “An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part I: solving problems with box constraints,” *IEEE Trans. Evol. Comput.*, vol. 18, no. 4, pp. 577–601, 2014.

[24] J. Chen and A. H. Sayed, “Distributed pareto optimization via diffusion strategies,” *IEEE J. Sel. Top. Signal Process.*, vol. 7, no. 2, pp. 205–220, 2013.

[25] L. Wan, J. Yuan, and L. Wei, “Pareto optimization scheduling with two competing agents to minimize the number of tardy jobs and the maximum cost,” *Applied Mathematics and Computation*, vol. 273, pp. 912–923, 2016.

[26] Y. Zhang, J. Yuan, C. T. Ng, and T. C. E. Cheng, “Pareto-optimization of three-agent scheduling to minimize the total weighted completion time, weighted number of tardy jobs, and total weighted late work,” *Naval Research Logistics*, vol. 68, no. 3, pp. 378–393, 2021.

[27] X. Chen, Y. Du, L. Xia, and J. Wang, “Reinforcement recommendation with user multi-aspect preference,” in *WWW Conference*, 2021.

[28] F.-Y. Wang, “Agent-based control for networked traffic management systems,” *IEEE Intelligent Systems*, vol. 20, no. 5, pp. 92–96, 2005.

[29] C. Bell, A. Nerode, R. T. Ng, and V. Subrahmanian, “Implementing deductive databases by linear programming,” in *PODS*, 1992.

[30] ——, “Mixed integer programming methods for computing nonmonotonic deductive databases,” *J. ACM*, vol. 41, no. 6, pp. 1178–1215, 1994.

[31] V. Chandru and J. N. Hooker, “Extended Horn Sets In Propositional Logic,” *J. ACM*, vol. 38, no. 1, pp. 205–221, 1991.

[32] V. de Wit, D. Doder, J. J. Meyer *et al.*, “Probabilistic deontic logics for reasoning about uncertain norms,” *Journal of Applied Logics*, vol. 2631, no. 2, p. 193, 2023.

[33] F. Olivieri, G. Governatori, M. Cristani, A. Rotolo, and A. Sattar, “Deontic meta-rules,” *Journal of Logic and Computation*, vol. 34, no. 2, pp. 261–314, 2024.

[34] J. W. Lloyd, *Foundations of logic programming*. Springer Science & Business Media, 2012.

[35] W. F. Clocksin and C. S. Mellish, *Programming in PROLOG*. Springer Science & Business Media, 2003.

[36] S. Costantini and A. Tocchio, “A logic programming language for multi-agent systems,” in *JELIA Conference*, 2002.

[37] J. J. Alferes, F. Banti, and A. Brogi, “An event-condition-action logic programming language,” in *JELIA Conference*, 2006.

[38] C. A. C. Coello, *Evolutionary algorithms for solving multi-objective problems*. Springer, 2007.

[39] M. Ehrgott, *Multicriteria Optimization*. Springer, 2005.

[40] H. A. Taboada, F. Baheranwala, D. W. Coit, and N. Wattanapongsakorn, “Practical solutions for multi-objective optimization: An application to system reliability design problems,” *Reliability Engineering & System Safety*, vol. 92, no. 3, pp. 314–322, 2007.

[41] B. Malakooti and V. Raman, “Clustering and selection of multiple criteria alternatives using unsupervised and supervised neural networks,” *J. Intell. Manuf.*, vol. 11, pp. 435–453, 2000.

[42] B. Malakooti and Z. Yang, “Clustering and group selection of multiple criteria alternatives with application to space-based networks,” *IEEE Trans. Syst. Man Cybern. Part B*, vol. 34, no. 1, pp. 40–51, 2004.

[43] M. Cheikh, B. Jarboui, T. Loukil, and P. Siarry, “A method for selecting pareto optimal solutions in multiobjective optimization,” *Journal of Informatics and mathematical sciences*, vol. 2, no. 1, pp. 51–62, 2010.

[44] S. D. Kahagalage, H. H. Turan, F. Jalalvand, and S. E. Sawah, “A novel graph-theoretical clustering approach to find a reduced set with extreme solutions of pareto optimal solutions for multi-objective optimization problems,” *J. Glob. Optim.*, vol. 86, no. 2, pp. 467–494, 2023.

[45] U. Brandes, “A faster algorithm for betweenness centrality,” *Journal of mathematical sociology*, vol. 25, no. 2, pp. 163–177, 2001.

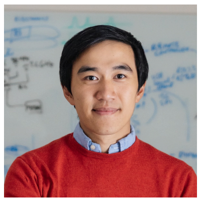
[46] M. Brinkmeier, “Pagerank revisited,” *ACM Trans. Internet Techn.*, vol. 6, no. 3, pp. 282–301, 2006.

Tonmoay Deb is a third-year Ph.D. student at the Department of Computer Science, Northwestern University. He is doing research under the supervision of Professor VS Subrahmanian at Northwestern Security and AI Lab. His research interests are in the intersection of Machine Learning, Computer Vision, Natural Language Processing, and Multi-Agent Systems.

Mingi Jeong is a Ph.D. candidate of Reality and Robotics Lab in the Department of Computer Science at Dartmouth College, USA. His current interests are autonomous navigation, multi-robot system, and maritime collision avoidance decision-making.

Cristian Molinaro is an Associate Professor in the DIMES Department at University of Calabria. His main research interests include knowledge representation and reasoning and explainable AI.

Andrea Pugliese is an Associate Professor in the DIMES Department at University of Calabria. His main research interests include computer security and graph data management.



Alberto Quattrini Li is an assistant professor at the Computer Science Department at Dartmouth College and co-director of the Reality and Robotics Lab. His research interests include autonomous mobile robotics, artificial intelligence, and agents and multiagent systems.

Eugene Santos Jr. (M'93–SM'04–F'12) is the Sydney E. Junkins 1887 Professor of Engineering at the Thayer School of Engineering and Adjunct Professor of Computer Science at Dartmouth College. His current focus is on computational intent, dynamic human behavior, and decision-making with an emphasis on learning nonlinear and emergent behaviors and explainable AI.

V.S. Subrahmanian is the Walter P. Murphy Professor of Computer Science and a Fellow in the Buffett Institute for Global Affairs at Northwestern University. He works at the intersection of AI and security issues.

Youzhi Zhang is an Assistant Professor at the Centre for Artificial Intelligence and Robotics, Hong Kong Institute of Science & Innovation, Chinese Academy of Sciences. His research interests include AI, multi-agent systems, and computational game theory.