Automating Safety Proofs about Cyber-Physical Systems using Rewriting Modulo SMT

Vivek Nigam[2,3] and Carolyn Talcott[1]

[1] SRI International, Menlo Park, USA, clt@csl.sri.com
[2] Federal University of Paraíba, João Pessoa, Brazil, vivek.nigam@gmail.com
[3] Huawei Munich Research Center, Germany

Abstract. Cyber-Physical Systems, such as Autonomous Vehicles (AVs), are operating with high-levels of autonomy allowing them to carry out safety-critical missions with limited human supervision. To ensure that these systems do not cause harm, their safety has to be rigorously verified. Existing works focus mostly on using simulation-based methods which execute simulations on concrete instances of logical scenarios in which systems are expected to function. The level of assurance obtained by these methods is, therefore, limited by the number of simulations that can be carried out. A complementary approach is to produce, instead, proofs that vehicles are safe for all instances of logical scenarios. This paper investigates how Rewriting modulo SMT applied to Soft Agents, a rewriting framework for the specification and verification of Cyber-Physical system, can be used to generate such proofs in an automated fashion. In particular, rewrite rules specify the executable semantics of systems on logical scenarios instead of concrete scenarios. This is accomplished by generating at each execution step a set of (non-linear) constraints whose satisfiability are checked by using SMT-solvers. Intuitively, a model of such set of constraints corresponds to a concrete execution on an instance of the corresponding logical scenario. We demonstrate how to specify and verify scenarios in this framework using an example involving a vehicle platoon. Finally, we investigate the trade-offs between how much of the verification is delegated to search engines (namely Maude) and how much is delegated to SMT-solvers (e.g., Z3).

1 Introduction

Autonomous Vehicles (AVs) are expected to soon reach higher-levels of autonomy, being able to drive through complex environments with no or little human supervision. To achieve this, however, it is necessary to produce a rigorous safety assurance argument [12]. An assurance strategy based on collecting data by running AVs on the streets is not feasible [13] as it would require billions of miles of data for achieving confidence in the results. Symbolic methods based on formal models have been advocated [24] as a means for safety assurance.

A safety assurance strategy begins by first identifying abstract scenarios, called logical scenarios [19], such as lane changing or platooning or pedestrian crossing, in which AVs have to avoid harm. These logical scenarios contain details

about the situations in which a vehicle shall be able to safely operate,[4] such as which types and number of actors, e.g., vehicles, pedestrians, operating assumptions, e.g., range of speeds, and road topology, e.g., number of lanes. The system safety is then verified with respect to each scenario. The challenge, however, is that there are infinitely many instances for any given logical scenario.

To overcome this challenge, existing work can be divided into two different approaches. The first approach [9,16,5] is to use simulation-based methods that run a sufficiently large number of simulations using vehicle simulators [8]. A limitation of this approach is that a possibly large number of simulations need to be generated for each logical scenario, and even then critical situations may be missed. The second approach is to use algorithms [23,1] that are proved to generate safe trajectories under the assumption that the remaining agents behave correctly. These safe planners can then be integrated with advanced (high-performance, but not safe) controllers as fall-back options whenever safety assurance is low [7]. There are two limitations with this approach. The first limitation is that safety proofs have to be constructed manually. The second limitation is that these proofs consider only planning and not other aspects such as sensing, knowledge bases, and communication channels that are used in AV applications [5,16].

This paper's main goal is to address the limitations of these two types of approaches by proposing a rewriting framework, based on Soft Agents [25], that enables the automated construction of vehicle level safety proofs, i.e., produce proofs that AVs are safe *for all instances of a logical scenario*. Such safety proofs provide greater confidence on the safety of AVs, complementing other verification evidence such as simulation-based verification techniques.

Towards achieving this goal, we make the following contributions:

- **Soft Agents Framework with Rewriting Modulo SMT:** We propose an executable symbolic Soft Agents framework [25] where instead of considering concrete values for attributes such as agent's speed, position and acceleration, it represents these values as symbols whose possible values are specified by a set of (real non-linear) constraints. This is accomplished by extending the current Soft Agents framework with Rewriting Modulo SMT [21]. Soft Agent specifications can be executed by using Maude extensions with SMT [14]. In contrast to existing frameworks that can execute only instances of logical scenarios, symbolic soft agents can execute logical scenarios producing symbolic traces, each denoting a possibly infinite number of concrete executions of the logical scenario.
- **Vehicle Platooning Specification:** We demonstrate the Soft Agent framework by using a simple, but realistic vehicle platooning application. We illustrate how vehicle behavior and safety properties can be specified in Soft Agents, explaining how design choices may affect verification performance.
- **Verification Trade-off between Rewriting and Constraint Solving:** For the verification of systems, Soft Agents make uses of rewriting (through Maude [4]) and of SMT-solvers (through Z3 [6]). In particular, rewriting cap-

[4] Also called Operational Design Domain (ODD).

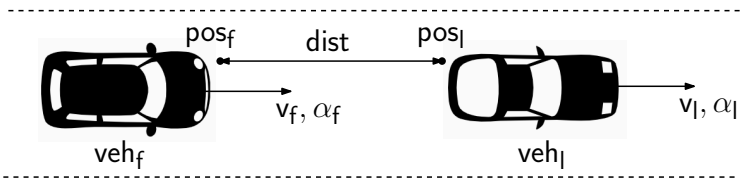

Fig. 1. Platooning Logical Scenario: The follower vehicle $\mathsf{veh_f}$ and $\mathsf{veh_l}$ are in a straight lane with respectively velocities and accelerations $\mathsf{v_f}, \alpha_\mathsf{f}$ and $\mathsf{v_l}, \alpha_\mathsf{l}$. $\mathsf{pos_f}$ is the position of front of $\mathsf{veh_f}$and $\mathsf{pos_l}$is the position of the back of the $\mathsf{veh_l}$. We consider vehicle positions to be only the x-component increasing with as one follows to the right direction of the road. The distance between the vehicles $\mathsf{dist} = \mathsf{pos_l} - \mathsf{pos_f}$.

tures the evolution of the system by accumulating constraints. The constraint-solver, on the other hand, generates proofs that a property is satisfied or that a property is unsatisfiable. We investigate in this paper the trade-offs between how much of verification is delegated to rewriting and how much to the constraint-solver. On the one hand, the more fine grained is the rewriting, e.g., searching with more constrained system evolutions, the greater is the number of states the search engine has to traverse leading to a greater number of calls to the SMT-solver, but the simpler are the problems that the solver has to solve. On the other hand, the more coarse is the rewriting, e.g., searching with less constrained system evolutions, the fewer are the calls to the SMT-solver, but the larger are problems that the constraint solver has to solve. Our experiments indicate that these trade-offs need to be considered in order to verify more challenging properties.

Plan. We start in Section 2 by describing a motivating example: a logical scenario from a vehicle platooning case study, which is used as running example. Section 3 introduces symbolic rewriting, then recalls the soft agents framework and its generalization to to symbolic rewriting with SMT solving. Section 4 presents key elements of the symbolic vehicle platooning logical scenario, including control decisions, safety properties, and search patterns for reachability analysis. Section 5 presents experiments evaluating trade-offs between size of search space and complexity of constraints to solve. We conclude by discussing related work in Section 6 and future work in Section 7.

2 Motivating Example

Our motivating example is a platooning scenario which is a typical Level 3 autonomy[5] use-case. This scenario takes place in a highway as illustrated by Figure 1. The vehicle $\mathsf{veh_f}$, called follower vehicle, follows autonomously, i.e.,

[5] For the levels of autonomy, see the SAE classification described in [11]

only with human supervision, vehicle $\mathsf{veh_l}$, called leader vehicle. The vehicles are driving in a highway lane and therefore are expected to have a speed within some given range of values normally obtained by considering legal speeds and the vehicle's capabilities, e.g., speeds between $60km/h$ and $130km/h$. Moreover, the acceleration (and deceleration) capabilities of the vehicles are also bounded, typically between $-8m/s^2$ and $2m/s^2$.

The goal of the follower vehicle is to maintain a safe distance to the leader vehicle, but still be close enough to profit from the wind shadow of the leader vehicle yielding upto 17% of fuel savings [27]. Since the speed of the vehicles may vary, it is not appropriate to define a safe distance as an absolute quantity, but in terms of *time to react*. That is, the distance will depend on the relative speeds of the vehicles.

As an example, building on ideas from [7], we define the following three properties for the platooning logical scenario:

$$P_{\mathsf{safer}} := \mathsf{dist} \geq \mathsf{v_f} \times (1[s] + \mathsf{gap_{safer}}) - \mathsf{v_l} \times 1[s], \tag{1a}$$

$$P_{\mathsf{safe}} := \mathsf{v_f} \times (1[s] + \mathsf{gap_{safer}}) - \mathsf{v_l} \times 1[s]) > \mathsf{dist} \geq \mathsf{v_f} \times (1[s] + \mathsf{gap_{safe}}) - \mathsf{v_l} \times 1[s], \tag{1b}$$

$$P_{\mathsf{unsafe}} := \mathsf{dist} < \mathsf{v_f} \times (1[s] + \mathsf{gap_{safe}}) - \mathsf{v_l} \times 1[s] \tag{1c}$$

Intuitively, their satisfaction is conditional on the distance (dist) between the vehicles; their speeds ($\mathsf{v_l}$ and $\mathsf{v_f}$); and the parameters $\mathsf{gap_{safer}}$ and $\mathsf{gap_{safe}}$ which are time to react parameters, typically a few seconds. Moreover, $\mathsf{gap_{safer}} > \mathsf{gap_{safe}}$, which means that the instance of a logical scenario satisfies P_{safer} (or simply safer) if the vehicles $\mathsf{veh_l}$ and $\mathsf{veh_f}$ have a greater distance between them. Finally, an instance of a logical scenario satisfies P_{unsafe} (or simply unsafe) if distance is to small to satisfy P_{safe} or P_{safer}.

A description of the function of a vehicle, such as platooning, using formal notations and ranges of parameters is called a *logical scenario* [15]. The objective is to prove that an implementation of a controller for the platooning function is safe, that is either P_{safer} or P_{safe} is satisfied for all concrete instances of this logical scenario. This is challenging as there are infinitely many such instances.

3 Symbolic Soft Agents Framework

We begin with an overview of challenges in modeling cyber-physical systems (CPSs), then recall the main features of soft agent specifications, and then briefly discuss the generalization to symbolic form.

3.1 Overview

A soft agent (SA) model of a CPS makes explicit both discrete changes (cyber actions, control settings) and continuous change (in the physical environment). Following ideas developed in Real Time Maude [18], soft agent models have instantaneous rules that specify agents decision processes that generate actions

such as communication or setting control parameters; and a timeStep rule that models the passage of some interval of time, updating the state according to a model of the time-dependent aspects of the state.

In contrast to the usual realtime specifications, soft agent CPS specifications involve variables, such as speed, distance, etc, that are dense and their evolutions over time are not discrete events. Moreover, system properties, such as safety properties, are expressed using these variables, e.g., keeping a given distance to the vehicle ahead rather than timing properties such as network delay or execution time. Verification of safety properties for CPS specifications involves reasoning about possibly infinitely many states and properties whose parameters may change continuously over time.

Two challenges for safety analysis of CPS specifications are (1) soundness of discrete time sampling execution; and (2) checking for reachability of unsafe states from a possibly infinite set of instances of a logical scenario. Challenge (1) includes choosing the timestep intervals small enough so that no unsafe situations are missed, while not being so fine grained that the state space becomes unmanageable. This is a design time concern, for example choosing the frequency with which sensors are read and control settings are updated. The latter challenge (2) involves the coverage and state space management with time properties.

Real Time Maude addresses (1) in [17], defining conditions on a timed rewrite theory that guarantee soundness and completeness of model checking based on maximal time elapsed discrete time sampling. Unfortunately, soft agent analysis problems generally do not meet these conditions. Narrowing is one approach to checking reachability from a possibly infinite initial set of system states. Maude supports narrowing modulo a rich collection of equational theories, but narrowing using conditional rules is not supported [4], and soft agent relies heavily on conditional rules.

New ideas are needed to address the verification challenges. We propose a form of symbolic rewriting that combines rewriting and constraint solving.

1. We represent *logical scenarios* as symbolic system states, representing a set of concrete states. A logical scenario consists of a pattern (a term with pattern variables called symbols) together with a set of constraints on values of the symbols.[6]
2. A symbolic rewrite rule introduces new symbols and additional constraints representing new values of the pattern variables. The resulting logical scenario represents the instances reachable from instances of the starting pattern using the rewrite rule.
3. Symbolic rule conditions use symbolic function evaluation to generate new symbols and their constraints.

The point of symbolic analysis is to check properties of concrete systems represented by concrete scenarios. Thus we want to connect symbolic executions to concrete executions. The concrete executions may be obtained from a concrete

[6] Mathematically, a logical scenario is a term with variables. To be able to rewrite logical scenarios in Maude, we replace variables by symbols, which formally are uninterpreted constants.

form of the rewrite rules, or simply using the symbolic rules with grounding constraints of the form, `sym == ground term`.

To describe the desired symbolic-concrete connection, we need a little notation. The basic idea is analogous to that presented in [20]. We assume a rewrite theory $\mathcal{T} = (\Sigma, B \cup E, R)$ with signature Σ, axioms B, equations E, and rules R. Assume further an equational subtheory $\mathcal{T}_0$ of $\mathcal{T}$ axiomatizing the theory in which the constraints are solved by the SMT solver. We use sS, sS_0, sS_1 .... to denote logical scenarios (symbolic states) and cS, cS_0, cS_1 ... to denote concrete states (ground states with no symbols). Let σ, σ_0, σ_1, ... denote substitutions mapping symbols to concrete terms (values). A logical scenario is structured as a pair $(\mathsf{sP}, \mathsf{sC})$ consisting of a pattern, sP, and a constraint, sC, on the symbols of sP. sC represents a quantifier free formula in the language of $\mathcal{T}_0$.

Application of a substitution, σ, to a logical scenario, $\mathsf{sS} = (\mathsf{sP}, \mathsf{sC})$ (written $(\sigma\mathsf{sP})$), gives an instance of sS if the domain of σ contains all the symbols of sS and σ satisfies sC ($\mathcal{T}_0 \models \mathsf{sC}\sigma$). We say σ_1 extends σ_0, written $\sigma_1 \gg \sigma_0$ if the domain of σ_1 contains the domain of σ_0 and $\sigma_0(v) = \sigma_1(v)$ (wrt. $\mathcal{T}$) for v in the domain of σ_0. Finally, we let $\longrightarrow_c$ denote the concrete rewrite relation induced by $\mathcal{T}$, and $\longrightarrow_s$ denote the symbolic rewrite relation induced by $\mathcal{T}$. Then the desired connection between the rewrite relations is give by the following *Soundness* and *Completeness* properties. These correspond to Theorems 1 and 2 of [20] and can be proved by analogous arguments.

Soundness. If $\mathsf{sS}_0 \longrightarrow_s \mathsf{sS}_1$ and σ_0 gives an instance of sS_0, then there exists $\sigma_1 \gg \sigma_0$ such that cS_1 is equivalent (in $\mathcal{T}$) to $\mathsf{sP}_1\sigma_1$ and and $\sigma_0(\mathsf{sP}_0) \longrightarrow_c \mathsf{cS}_1$.

Completeness. If σ_0 gives an instance of sS_0 and $\sigma_0(\mathsf{sP}_0) \longrightarrow_c \mathsf{cS}_1$ then there exists sS_1, and $\sigma_1 \gg \sigma_0$ such that σ_1 gives an instance of sS_1 with cS_1 equivalent to $\sigma_1(\mathsf{sP}_1)$ and $\mathsf{sS}_0 \longrightarrow_s \mathsf{sS}_1$ where σ_1 gives an instance of sS_1.

3.2 The structure of Soft Agent Rewriting

In soft agents, a system state consists of a set of agent terms together with a unique environment term. Abstractly an agent term has the form `A(id,attrs)` where `id` is the agent identifier, and `attrs` is a set of named attributes including the agents local knowledge base (local KB), and a set of pending tasks and actions each labeled by the time until ready for execution. An environment term has the form `E(ekb)` where `ekb` is a knowledge base representing the physical state of the system and contextual information such as location of features or bounds on location.

There are two rewrite rules: `doTask` and `timeStep`. The `doTask` rule has the form

```
crl[doTask]: A(id,attrs) E(ekb) => A(id,attrs') E(ekb) if taskConds
```

where `taskConds` has clauses for reading sensors from the environment, evaluating possible actions, and updating the local KB, pending tasks, and actions. The `timeStep` rule has the form

```
crl[timeStep]: A(id1,attrs1) ... A(idk,attrsk) E(ekb)  =>
               A(id1,attrs1') ... A(idk,attrsk') E(ekb') if stepConds
```

where `stepConds` has a clause to execute ready actions (with time delay 0) and update time-dependent symbols to capture the passing of time. There are also clauses to update time parameters (clocks, delays...), transmit messages, and share knowledge amongst the agents. Executing actions affects parameters that control how the physical state evolves (change of acceleration, direction, on/off switches ...). Passing time lets the physical model run for the specified interval of time, updating the physical state (position, energy level, ...) according to laws parameterized by the control settings.

3.3 Symbolic Soft Agent Rewriting

To enable symbolic execution of soft agent specifications we abstract system states as terms of the form `SA[uu] SE[vv]` where `SA` is a pattern with symbols `uu` whose structure captures the state aspects that are not changed during execution, for example the number of agents, their ids, attribute names, and any persistent structure in attribute values. Similarly, `SE[vv]` is a pattern, with symbols `vv` capturing the persistent structure in the environment knowledge base. `uu` and `vv` are disjoint lists of symbols. For example, in a platooning scenario, symbols in `vv` would represent values including the position, acceleration, and velocity of each vehicle. Mathematically, we represent the symbolic constraint as a separate state component. In practice, we represent it as an element of the environment knowledge base.

Intuitively, the execution of a logical scenario constructs new constraints containing fresh symbols representing new values of the system's physical attributes. As for (concrete) soft agents, there are two rewrite rules for symbolic soft agents. At the framework level, the symbolic rules are obtained by replacing the clauses in the rule conditions of concrete rules by symbolic versions that refer to symbolic versions of the functions involved. It is the job of the specifier to define these symbolic functions and their symbolic evaluation equations. In the vehicle platooning case, symbolic functions were obtained by systematically transforming the original concrete versions. In the next section we give examples of key elements of the symbolic vehicle platooning system.

4 Vehicle Specifications

This section details how one can specify logical scenarios including safety properties by specifying the vehicle platooning example described in Section 2. While the specifications below are declarative, i.e., closely resemble textbook formulas, we do assume that the reader is familiar with the Maude syntax [4]. Our starting point is a concrete specification of the vehicle platooning described in [5]. It contains several features, such as vehicle controllers and communication protocol specifications, which have been ported to the symbolic machinery described

below. The complete code can be found at https://github.com/SRI-CSL/VCPublic.git in the folder `symbolic-platooning`. To execute this code you will need the Maude integration with Z3 which can be found at [14].

4.1 Basic Symbolic Sorts

`RealSym` is the sort of real values. It contains concrete values, i.e., real numbers, or symbols of the form `vv(i)` or `vv(i,str)` where `i` is a `Nat` uniquely identifying a symbol and `str` is a string describing the intuitive meaning of the symbol, used for improved readability. The term `mkNuVar(i,id,str)` evaluates to a (fresh) symbol with identifiers `id`,`str`, where `id` is an agent identifier and `str` is a string with a short description of the fresh symbol.

Example 1. The following symbols represent the initial conditions for the follower `ag1`, namely, its position, speed, maximum acceleration, maximum deceleration, and initial acceleration.

```
eq v1posx = vv(2,"ag1-positionX") . eq v1posy = vv(3,"ag1-positionY") .
eq v1vel = vv(5,"ag1-speed") .      eq maxacc1 = vv(9,"ag1-maxAcc") .
eq maxdec1 = vv(10,"ag1-maxDec") .  eq acc1 = vv(32,"ag1-acc") .
```

`SymTerm` is the sort of symbolic terms containing arithmetic expressions constructed inductively using basic arithmetic operators (e.g., addition, subtraction, division, multiplication) and elements of `RealSym`. They are used to specify constraints of sort `Boolean` involving symbols.

Example 2. The following constraint using the symbols in Example 1 specifies that `ag1`'s acceleration is bounded by the maximum acceleration and deceleration: `(acc1 <= maxacc1) and (acc1 >= maxdec1)`

4.2 Knowledge Specifications

Cyber-physical systems reason using knowledge about their locations, speeds, direction, and accelerations and of the surrounding objects. Such knowledge is represented using a sort `Info`. Knowledge base elements are of the form `info @ t` where `t` is a logical time, i.e., the number of time steps since the beginning.

Vehicle locations are two-dimensional, speeds are real values, and directions are vectors specified using two locations and a magnitude:

```
op loc : SymTerm SymTerm -> Loc .
op speed : Id RealSym -> Info .
op dir : Id Loc Loc SymTerm -> Info .
```

Example 3. The agent `ag1`'s initial knowledge base, that is, at logical tick 0, contains the following terms, specifying its initial position, speed, acceleration and direction:

```
(at(ag1,loc(v1posx,v1posy)) @ 0) (speed(ag1,v1vel) @ 0)
(accel(ag1,acc1) @ 0)
(dir(v(1),loc(v1ix,v1iy),loc(v1tx,v1ty),v1mag) @ 0)
```

Based on the above notation, we can specify symbolically typical definitions, such as the distance between two locations:

```
op ldist : Nat Loc Loc -> NatSymTermBoolean .
eq ldist(i,loc(x0,y0),loc(x1,y1))
 = {s(i),vv(i,"dist"), (vv(i,"dist") >= 0/1) and
    vv(i,"dist") * vv(i,"dist") === ((y1 - y0) * (y1 - y0) +
                                     (x1 - x0) * (x1 - x0)) } .
```

This definition creates a fresh symbol, `vv(i,"dist")` together with the constraint specifying the Euclidean distance. Notice that we need to specify that the distance is a non-negative value. Similar specifications can be made for other distance measures, such as, Manhattan distance.

The following operator specifies how an agent's location, `loc(x,y)`, is updated to `loc(nuVarX,nuVarY)` given an (average) speed, `spd`, and a direction.

```
op upVLoc : Nat Id Loc SymTerm Info -> NatLocBoolean .
ceq upVLoc(i,id,loc(x,y),spd,dir(id,loc(x0,y0),loc(x1,y1),mag))
 = {i + 2,loc(nuVarX,nuVarY),cond}
 if nuVarX := mkNuVar(i,id,"-positionX")
 /\ nuVarY := mkNuVar(i + 1,id,"-positionY")
 /\ cond1 := (x0 === x1) and (not (y0 === y1)) and
             (nuVarX === x) and (nuVarY === y + spd)
 /\ cond2 := (not (x0 === x1)) and (y0 === y1) and
             (nuVarX === x + spd) and (nuVarY === y)
 /\ cond3 := (not (x0 === x1)) and (not (y0 === y1)) and
             (nuVarX === (x + spd * (x1 - x0) / mag)) and
             (nuVarY === (y + spd * (y1 - y0) / mag))
 /\ cond := cond1 or cond2 or cond3 .
```

We made some design choices in this definition. The first design choice is to split it into three different cases. The first case (`cond1`) is when the agent is moving vertically, the second case (`cond2`) horizontally, and the third case (`cond3`) when it is moving in the quadrant. In this way we help the constraint solver to avoid to solve the harder non-linear constraint involved in the third case whenever the agent is moving only along the x-axis and only along the y-axis. The second design choice was to include the magnitude in the definition of `dir` which may seem redundant as it can be specified from the two associated locations. However, by doing so, we avoid the need to generate fresh symbols and new constraints whenever the magnitude is needed as in the third case of `upVLoc`.

Finally, we also capture symbolically the fact that the physical system is continuous while the cyber part of the system works in logical ticks. The size of the tick is specified by the term `tickSize(dt)`, where `dt` is symbol denoting the size of the tick. Typically it is fixed during the whole execution by using a constraints, e.g., `dt === 1/10`, specifying a tick duration of $100ms$. We assume here for simplicity that all agents use the same tick duration. However, agents with different tick duration can also be specified. When the soft agent machinery updates the agent's positions using `upVLoc` it scales accordingly the speed to the tick size by multiplying the speed with `dt`.

4.3 Soft-Constraint Controller

Agents decide which action to take based on their local knowledge base, which is updated by reading their sensors, and taking into account different concerns, such as safety and efficiency. For vehicle platooning, as described in detail in [5], there are two main concerns, *safety*, i.e., maintaining a safe distance between vehicles, and *fuel-efficiency*, i.e., maintaining a distance between vehicles that is not too great.

The controller is specified in a similar way to the knowledge functions described above by using existing symbols, creating new symbols, and using constraints to determine its possible values.

The following equation specifies the controller evaluation to rank the possible actions that the vehicle can take from a safety perspective. In particular, it takes as input `i`, for creating fresh symbols, `vmin,vmax`, respectively, the minimum and maximum speeds that the vehicle is allowed to use, `vminD,vmaxD`, the minimum and maximum desired speeds according to the safety parameters ($\mathsf{gap}_{\mathsf{safe}}$, $\mathsf{gap}_{\mathsf{safer}}$), and the constraints `cond` on the existing symbols. It then returns a range of speeds that are safe specified by the interval between the fresh symbols `vv(i)`and `vv(i + 1)`. However, the concrete values for `vv(i),vv(i + 1)` depend on the relation between the possible speeds (`vmin,vmax`) and the desired speeds `vminD,vmaxD` as detailed by the constraints `cond11,cond21,...,cond61`.

```
ceq symValSpeedRed(i,str,vmin,vmax,vminD,vmaxD,cond) =
{i + 2, [vv(i),vv(i + 1),nuCond and cond]}
if cond1 := vmin >= vmaxD
/\ cond11 := vv(i) === vmin and
             vv(i + 1) === ((vmin + vmax) / 2/1) and cond1
/\ cond2 := vmax <= vminD
/\ cond21 := vv(i) === ((vmin + vmax) / 2/1)
        and vv(i + 1)  === vmax and cond2
...
/\ cond6 := vmin >= vminD and vmax < vmaxD
/\ cond61 := vv(i) === vmin and vv(i + 1)  === max and cond6
/\ nuCond := (cond11 or cond21 or cond31 or cond41 or cond51 or cond61) .
```

In the definition above, the effort of determining which condition applies is delegated to the constraint solver. As we will investigate in Section 5, this will lead to great performance penalties.

An alternative way to expressing the same controller is to return six possibilities as specified by the following equation, rather than the single disjunction `nuCond`:

```
ceq symValSpeedRed-Split(i,str,vmin,vmax,vminD,vmaxD,cond) =
  {i + 2, [vv(i),vv(i + 1),cond11 and cond]}
  {i + 2, [vv(i),vv(i + 1),cond21 and cond]}
  ...
  {i + 2, [vv(i),vv(i + 1),cond61 and cond]}
 if cond1 := vmin >= vmaxD
 ...
 /\ cond61 := vv(i) === vmin and vv(i + 1)  === vmax and cond6.
```

With this new definition the choice of which condition is applicable is left to the search engine, i.e., Maude.

A similar choice occurs when specifying how the time advancement affects agent's speeds. Several cases occur due to the fact that logical scenarios assume that vehicle's speeds are bounded. For example, depending on the tick duration, current speed and maximum acceleration, an agent's speed may reach the maximum speed or not before completing a logical tick. For analyzing the impact of delegating such enumeration of cases to the SMT-solver or to the search engine, we implemented two versions of time advancement: `timestep` that returns one output with a constraint with a disjunct for each case, as in `symValSpeedRed`; and `timestep-split` that returns several outputs, one for each possible case as `symValSpeedRed-split`.

4.4 System Configurations

As described in Section 3, a system configuration of sort `ASystem` is a collection of agent configurations and an environment configuration.

An agent configuration has the form `[id : class | attrs ]`, where `id` is the agent's unique identifier, `class` is its class, e.g., vehicle, and `attrs` are its attributes which include its local knowledge base written `lkb : kb`, where `lkb` is a label and `kb` is the local knowledge base contents.

An environment configuration has the form `[eId | ekb]` where `ekb` is the environment knowledge base which specifies state of the world. The environment knowledge base contains the knowledge item `constraints(i,cond)` where `i` is the current index of fresh variables, and `cond` is the constraints (accumulated) on the existing symbols.

Example 4. The initial configuration of a platooning scenario described in Section 2 is as follows:

```
asysI = { [eid | (kb constraint(i,condI))]
          [v(0) : veh | lkb : kb0 ] [v(1) : veh | lkb : kb1 ] }
```

where `kb` is the environment knowledge base specifying among other things, the vehicles's actual locations and speeds, while `kb0` and `kb1` are the vehicle `v(0)` and `v(1)`'s local knowledge bases. The constraint `condI` contains the constraints on these values as per the logical scenario. It contains for example constraints on the acceleration of vehicles (see Example 2) and the following constraints:

```
(v1vel >= vellb1) and (v1vel <= velub1) and (v0posy > v1posy)
```

which specify that the follower vehicle's speed is bound within the bounds `vellb1` and `velub1`. Moreover, the following vehicle `v(1)` is behind the leader `v(0)`.

Notice that such a symbolic system configuration may correspond to infinitely many concrete system configuration, i.e., concrete instances of the specified platooning scenario.

4.5 Safety Properties

We are interested in generating proofs regarding the safety of logical scenarios, such as the one specified in Example 4. The specification of safety property is formalized using the operator:

```
op mkSPCond : SP ASystem -> SPSpec .
```

This function takes a property (an identifier in `SP`) and a system configuration, and returns a safety property of sort `SPSpec` of the form:

```
op {_,_,_,_} : Nat SymTerms Boolean Boolean -> SPSpec .
```

The first element is the new symbol index, the second is the new (auxiliary) symbols created for specifying the property, which are then constrained by the third element. The last element specifies the safety property based on the auxiliary symbols and the previously existing symbols in the given system configuration.

For example, the first safety property in Eq 1b is specified as follows:

```
ceq mkSPCond(saferSP, { conf env }) = {k + 1,dis,cond00,nucond}
if [id0 | kb] := env
/\ (atloc(v(0),l0) @ t0) (atloc(v(1),l1) @ t1)
   (speed(v(0),v0) @ t2) (speed(v(1),v1) @ t3)
   (gapSafety(v(1),gapSafer,gapSafe)) (constraint(n,cond)) kb1 := kb
/\ {k,dis,cond00} := ldist(n,l1,l0)
/\ nucond := (dis >= ((1/1 + gapSafer) * v1) - v0) .
```

Notice the use of the function `ldist` that creates the auxiliary fresh symbol `dis`.

Using `mkSPCond`, we specify an operator (definition elided)

```
op enforceSP : SP ASystem -> ASystem .
```

For example, `enforce(saferSP,asysI)` returns a configuration in which the conditions (`cond00` and `nucond` from `mkSPCond`) are added to the set of constraints. This means that the resulting configuration will only have instances `asysI` that satisfy the `saferSP`. The term `isSatModel(enforce(saferSP,asysI))` calls the SMT-Solver and returns an assignment for `asysI` symbols:

```
ag0-positionX |-> (0/1).Real,  ag0-positionY |-> (1/1).Real
ag1-positionX |-> (0/1).Real,  ag1-positionY |-> (0/1).Real,
ag0-speed |-> (7/1).Real,      ag1-speed |-> (2/1).Real,
ag1-safer |-> (3/1).Real
```

This state satisfies the `saferSP` property for a $\mathsf{gap}_{\mathsf{safer}}$ of value 3.

4.6 Verifying Logical Scenarios

We can now use Rewriting Modulo SMT [21] to verify and effectively generate safety proofs of the specifications above in an automated fashion. Consider the following search:

```
  search enforceSP(safeSP,setStopTime(asysI,2)) =>*
    asys such that checkSP(unsafeSP,asys) .
No solution. states: 63  rewrites: 394686 in 20134ms
```

It attempts to find any instance of system configuration `asys` that satisfies `unsafeSP` (see Eq. 1c) starting from any instance of `asysI` that satisfies property `safeSP`. Moreover, the term `setStopTime(asysI,2)` specifies that the search is bound to two logical ticks, i.e., search stops after two tick rules. The search engine combined with the SMT-solver can generate proofs that no instance of reachable states are unsafe. However, as shown Section 5, the complexity of the problem greatly increases when considering larger logical tick bounds.

5 Trade-offs Between Rewriting and Constraint Solving

The verification of logical scenario involves rewriting and constraint solving. Rewriting enumerates possible system states while the constraint solver attempts to check the satisfiability of constraints. As demonstrated in Section 4.3, how much of verification is delegated to rewriting and how much to the constraint solver can be adjusted by leaving the non-determinism in the constraints, e.g., by placing disjunctions in the constraints, or to the rewriting, e.g., returning instead for each disjunct an output, a rewriting choice.

Delegating verification to the rewriting engine means that the search tree is larger leading to more calls to the SMT-solver, but each call involves simpler constraints to solve, i.e., with less disjunctions and therefore less cases to consider. Delegating verification to the constraint solver, on the other hand, means a smaller search space traversed by the rewriting engine leading to less calls to the constraint solver, but with more complex constraints.

To demonstrate this, we considered three cases according to the specifications described in Section 4.3:

- **More SMT Less Search:** This case uses `symValSpeedRed` for the controller and `timestep` for the time step evolution. This means that all cases are specified as disjunctions in the constraint that will need to be solved by the solver.
- **Less SMT More Search:** This case uses `symValSpeedRed-split` for the controller and `timestep-split` for the time step evolution. This means that all cases are specified as different outputs that need to be traversed by the rewriting engine.
- **Balanced:** This case uses `symValSpeedRed` for the controller and the specification `timestep-split` for the time step evolution. This means that some cases are specified as constraints and others as outputs.

To evaluate the different cases, we executed the command:

```
search enforceSP(safeSP,setStopTime(asysI,Bound)) =>! asys
      such that isSat(asys) .
```

which enumerates all the reachable symbolic configurations that are satisfiable exactly in `Bound` time ticks, i.e., number of applications of the `timeStep` rule.

Time Bound	Pruning	More SMT Less Search	Balanced	Less SMT More Search
2	No	19/20.4s	71/2.5s	1427/29.7s
	Tick	19/32.4s	63/8.3s	497/47.4s
	All	19/56.0s	63/11.6s	296/52.7s
3	No	DNF	DNF	42827/3054s
	Tick	DNF	DNF	2484/3412s
	All	DNF	DNF	1976/5238s

Table 1. Experiments with the Platooning Logical Scenario Verification. DNF denotes that the experiment was aborted after 5 hours. The experiment results are expressed as *states*/*time*, where *states* is the total number of states in the search tree and *time* is the time needed to traverse all states. The experiments were carried out in a 2.2 GHz 6-Core Intel Core i7 machine with 16 GB memory.

A second dimension that we investigated was on the way we can prune the search tree. We considered the following cases:

- **All Pruning:** At each rewrite rule for `doTask`, which evaluates an agent's actions, and `tick`, which applies the agent's actions, we placed a check whether the resulting configuration is satisfiable. This means that the search tree has only satisfiable configurations with the price of calling the SMT-Solver at each step.
- **No Pruning:** As opposed to the `All Pruning` case, rewrites `doTask` nor `tick` did not check the satisfiability of the resulting configuration. The check was made only at the configuration resulting from applying the number to ticks specified by the bound. This means that the search tree is not pruned, and therefore, more states are traversed.
- **Tick Pruning:** The third case does a check on the configuration resulting from `timeStep` rewrites, but not on `doTask`. In this way, we still prune the search tree without calling the SMT-solver at each rewriting step.

Table 1 summarizes our experiments with these scenarios using bounds of two and three cs. The best case was not pruning the tree and delegating verification to the search tree when considering greater time bounds. The balanced case had better results when considering lower time bounds.

Interestingly, pruning the tree, while had a great effect on number of states, it did not improve the time required to traverse the tree. We believe that this can be further improved if the search engine uses the SMT-solver in a more clever way, in particular, using its incremental solving features. This would allow the solver to re-use work done in previous calls.

6 Related Work

Existing work for the verification of autonomous cyber-physical systems can be divided into three different approaches.

The first approach [9] is to use simulation-based methods that run a sufficiently large number of simulations using simulators [8]. A main advantage of this approach is that it can be used to verify the actual artifacts, e.g., machine learning artifacts, used in applications and rely on vehicle simulators to generate very complicated and high-fidelity scenarios. However, as already mentioned, as each simulation is run using a concrete instance of a logical scenario, a limitation of this approach is that possibly a large number of simulations need to be generated for each logical scenario. Our work complements this work by enabling the specification and verification of vehicle behavior using symbolic methods covering all instances of a logical senario, and enables early verification of designs before expensive artifacts are built.

The second approach is to use safe controllers [23,1] that are guaranteed to generate safe trajectories under the assumption that the remaining agents behave correctly. A limitation of this type of work is that it focuses only on individual functions, typically control algorithms without taking into account other functions needed for AVs, e.g., sensing, knowledge bases, and communication channels. As shown in [7], safe controllers can be integrated with advanced (high-performance, but not safe) controllers as fall-back options whenever safety assurance is low. In particular, a formal framework for Run Time Assurance (RTA) is presented, and conditions are given that, if satisfied by a safe controller and associated monitor, guarantee that integration with an untrusted control maintains safe operation. The paper leaves open methods to verify that a controller satisfies its RTA requirements. Our work has been greatly inspired by [7] and the result is complimentary. Symbolic rewriting combined with SMT solving provides automated methods to verify correctness of time sampling mechanisms and safety requirements.

The third approach [16,26,18], similar to the non-symbolic Soft Agents, are formal frameworks that enable the specification and verification of other functions, besides trajectory planning [10,5]. However, as with the first approach, the evidence that can be produced by these frameworks is based on running simulations or model checking concrete scenario instances. Therefore, it also suffers the limitation that a large number of simulations need to be carried out, or a large sample of senario instances must be model checked.

The Soft Agent execution strategy is based on the Real Time Maude maximal time elapse (MTE) execution strategy for real time theories [18]. In [17] two conditions for soundness and completeness of model checking Real Time Maude specifications based on the MTE execution strategy are given. The first condition, time robustness, is a property of the rewrite theory. It requires that timesteps of any duration are allowed, and a timestep can be subdivided without changing the end result. The second condition requires that atomic propositions are stable with respect to time: at most one change during a time step. These conditions hold for a wide range of Real Time Maude specifications, timing of protocols, network performance, or discrete events used for defining system behavior of, e.g., manufacturing plants. SA specifications are concerned with physical properties of a system such as bounds on distance, change of position,

use of resources to express both safety and goal satisfaction properties. SA specifications are time robust, but the properties of interest are generally not stable with respect to time. Thus, we can not directly use the Real Time Maude results. Work is in progress to define an analog to stability for system properties that evolve over time.

A formal mathematical foundation for symbolic rewriting modulo SMT is presented in [20]. Our work is essentially a mapping of these ideas to be executable in Maude with an integrated SMT solver. The soft agents `doTask` rule is not technically topmost, but could easily be modified to be topmost without changing any behavior in our examples. Also, the theory $\mathcal{T}$ has non-axiom equations that are not in $\mathcal{T}_0$ These equations define functions is a straight forward way, so they do not cause a problem for our symbolic rewriting but may challenge narrowing. Our logical scenarios are ground terms from Maude's perspective and correspond to terms whose only variables have builtin sorts (in $\mathcal{T}_0$). On the other hand, search starts with terms that possibly have non builtin variables in [20]. Generating new symbols to update values plays a similar role to the *fresh* substitution used in the symbolic rewrite relation of [20]. Important future work is to better understand criteria for allowing equations over non-builtin sorts, to make symbolic rewriting modulo SMT more generally applicable.

A notion of guarded term is introduced in [2] as a method to reduce the search state space in symbolic rewriting modulo SMT. A guarded term is a pair consisting of a term and a constraint, or the disjunction of a set of guarded terms. The paper develops the formal theory of rewriting with guarded terms and presents experiments based on the CASH protocol showing state space reduction for various forms of guard. Although the paper motivates guards by a need to also reduce complexity of constraints sent to the SMT solver, no results on constraint size are reported. The results in the present paper seem to suggest that not only the size of state space matters for automation, but also the size of constraints that are sent to the SMT-Solver. It will be interesting to see if guards can be used to control the tradeoffs between search space size and constraint size explored in the present paper.

7 Conclusions

This paper proposes an extension of Soft Agents frameworks with Rewriting Modulo SMT to enable the automated generation of safety proofs of CPS. We demonstrate its expressiveness with a vehicle platoon scenario which is a common feature of autonomous vehicles. We carry out a collection of experiments demonstrating that delagating verification to rewriting has a positive impact in verification performance.

We are planning to use this framework in several directions that complement related work. We are currently automating the verification conditions for RTA [7]. We also believe that our framework is applicable to problems other than vehicle safety, for example it could be used to enable symbolic security verification by extending our previous work [5].

Inspired by the presentation at WRLA 2022 on the Python bindings for Maude [22], we adapted our implementation to use the Python bindings instead of MaudeSE [14]. This enables full access to SMT-solver interface, including to new SMT-solvers such as CVC5 [3]. In the future, we plan to implement Python libraries based on these Python bindings for Maude to improve usability of the Soft Agents framework and quick integration to other tools/methods.

Acknowledgments. Talcott was partially supported by the U. S. Office of Naval Research under award numbers N00014-15-1-2202 and N00014-20-1-2644, and NRL grant N0017317-1-G002.

References

1. M. Althoff and J. M. Dolan. Online verification of automated road vehicles using reachability analysis. *IEEE Trans. Robotics*, 30(4):903–918, 2014.
2. K. Bae and C. Rocha. Symbolic state space reduction with guarded terms for rewriting modulo SMT. *Science of Computer Programming*, pages 20–42, 2019.
3. H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar. cvc5: A versatile and industrial-strength SMT solver. In D. Fisman and G. Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.
4. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude: A High-Performance Logical Framework*, volume 4350 of *LNCS*. Springer, 2007.
5. Y. G. Dantas, V. Nigam, and C. L. Talcott. A formal security assessment framework for cooperative adaptive cruise control. In *IEEE Vehicular Networking Conference, VNC 2020, New York, NY, USA, December 16-18, 2020*, pages 1–8. IEEE, 2020.
6. L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.
7. A. Desai, S. Ghosh, S. A. Seshia, N. Shankar, and A. Tiwari. SOTER: A runtime assurance framework for programming safe robotics systems. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019, Portland, OR, USA, June 24-27, 2019*, pages 138–150. IEEE, 2019.
8. A. Dosovitskiy, G. Ros, F. Codevilla, A. M. López, and V. Koltun. CARLA: an open urban driving simulator. In *1st Annual Conference on Robot Learning, CoRL 2017, Mountain View, California, USA, November 13-15, 2017, Proceedings*, volume 78 of *Proceedings of Machine Learning Research*, pages 1–16. PMLR, 2017.
9. D. J. Fremont, T. Dreossi, S. Ghosh, X. Yue, A. L. Sangiovanni-Vincentelli, and S. A. Seshia. Scenic: a language for scenario specification and scene generation. In K. S. McKinley and K. Fisher, editors, *Proceedings of the 40th ACM SIGPLAN*

Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019, pages 63–78. ACM, 2019.

10. I. i. Mason, V. Nigam, C. L. Talcott, and A. V. D. Brito. A framework for analyzing adaptive autonomous aerial vehicles. In A. Cerone and M. Roveri, editors, *Software Engineering and Formal Methods - SEFM 2017 Collocated Workshops: DataMod, FAACS, MSE, CoSim-CPS, and FOCLASA, Trento, Italy, September 4-5, 2017, Revised Selected Papers*, volume 10729 of *Lecture Notes in Computer Science*, pages 406–422. Springer, 2017.
11. S. J3016. https://www.sae.org/news/2019/01/sae-updates-j3016-automated-driving-graphic. 2021.
12. S. Jha, J. Rushby, and N. Shankar. Model-centered assurance for autonomous systems. In A. Casimiro, F. Ortmeier, F. Bitsch, and P. Ferreira, editors, *Computer Safety, Reliability, and Security - 39th International Conference, SAFECOMP 2020, Lisbon, Portugal, September 16-18, 2020, Proceedings*, volume 12234 of *Lecture Notes in Computer Science*, pages 228–243. Springer, 2020.
13. N. Kalra and S. M. Paddock. Driving to safety – https://www.rand.org/content/dam/rand/pubs/research_reports/RR1400/RR1478/RAND_RR1478.pdf. 2021.
14. MaudeSE. https://github.com/maude-se/maude-se.github.io. 2021.
15. T. Menzel, G. Bagschik, and M. Maurer. Scenarios for development, test and validation of automated vehicles. In *2018 IEEE Intelligent Vehicles Symposium, IV 2018, Changshu, Suzhou, China, June 26-30, 2018*, pages 1821–1827. IEEE, 2018.
16. F. Moradi, S. A. Asadollah, A. Sedaghatbaf, A. Causevic, M. Sirjani, and C. L. Talcott. An actor-based approach for security analysis of cyber-physical systems. In M. H. ter Beek and D. Nickovic, editors, *Formal Methods for Industrial Critical Systems - 25th International Conference, FMICS 2020, Vienna, Austria, September 2-3, 2020, Proceedings*, volume 12327 of *Lecture Notes in Computer Science*, pages 130–147. Springer, 2020.
17. P. C. Ölveczky and J. Meseguer. Abstraction and completeness for real-time maude. In G. Denker and C. L. Talcott, editors, *Proceedings of the 6th International Workshop on Rewriting Logic and its Applications, WRLA 2006, Vienna, Austria, April 1-2, 2006*, volume 174 of *Electronic Notes in Theoretical Computer Science*, pages 5–27. Elsevier, 2006.
18. P. C. Ölveczky and J. Meseguer. The real-time maude tool. In *TACAS 2008*, pages 332–336, 2008.
19. S. Riedmaier, T. Ponn, D. Ludwig, B. Schick, and F. Diermeyer. Survey on scenario-based safety assessment of automated vehicles. *IEEE Access*, 8:87456–87477, 2020.
20. C. Rocha, J. Meseguer, and C. Muñoz. Rewriting modulo SMT and open system analysis. *Journal of Logical and Algebraic Methods in Programming*, pages 269–297, 2017.
21. C. Rocha, J. Meseguer, and C. A. Muñoz. Rewriting modulo SMT and open system analysis. *J. Log. Algebraic Methods Program.*, 86(1):269–297, 2017.
22. R. Rubio. Maude as a library: an efficient all-purpose programming interface. In *Rewriting Logic and its Applications (WRLA)*, 2022.
23. S. Shalev-Shwartz, S. Shammah, and A. Shashua. On a formal model of safe and scalable self-driving cars. *CoRR*, abs/1708.06374, 2017.
24. J. Sifakis. Autonomous systems - an architectural characterization. *CoRR*, abs/1811.10277, 2018.

25. C. Talcott, V. Nigam, F. Arbab, and T. Kappé. Formal specification and analysis of robust adaptive distributed cyber-physical systems. In M. Bernardo, R. D. Nicola, and J. Hillston, editors, *Formal Methods for the Quantitative Evaluation of Collective Adaptive Systems*, LNCS. Springer, 2016. 16th edition in the series of Schools on Formal Methods (SFM), Bertinoro (Italy), 20-24 June 2016.
26. C. L. Talcott, F. Arbab, and M. Yadav. Soft agents: Exploring soft constraints to model robust adaptive distributed cyber-physical agent systems. In *Software, Services, and Systems - Essays Dedicated to Martin Wirsing*, pages 273–290, 2015.
27. S. van de Hoef, K. H. Johansson, and D. V. Dimarogonas. Fuel-efficient en route formation of truck platoons. *IEEE Trans. Intell. Transp. Syst.*, 19(1):102–112, 2018.