February 2002

Report on Verification of the MoBIES Vehicle-Vehicle Automotive OEP Problem

Franjo Ivancic
University of Pennsylvania

Follow this and additional works at: http://repository.upenn.edu/cis_reports

Recommended Citation
http://repository.upenn.edu/cis_reports/134


This paper is posted at ScholarlyCommons. http://repository.upenn.edu/cis_reports/134
For more information, please contact repository@pobox.upenn.edu.
Report on Verification of the MoBIES Vehicle-Vehicle Automotive OEP Problem

Abstract
The DARPA MoBIES Automotive Vehicle-Vehicle Open Experimental Platform [14] defines a longitudinal controller for the leader car of a platoon moving in an Intelligent Vehicle Highway System (IVHS) autonomously. The challenge is to verify that cars using this longitudinal controller provide a safe (that is, collision-free) ride. This report presents the process of verifying this particular controller using our CHARON [2] toolkit. In particular, it involves modeling and simulation of the system in CHARON and verifying the controller using our predicate abstraction technique for hybrid systems [3].

Comments

This technical report is available at ScholarlyCommons: http://repository.upenn.edu/cis_reports/134
Report on Verification of the MoBIES Vehicle-Vehicle Automotive OEP Problem

Franjo Ivancic
University of Pennsylvania,
Report on Verification of the MoBIES Vehicle-Vehicle Automotive OEP Problem

Franjo Ivančić
Department of Computer and Information Science
University of Pennsylvania
http://www.seas.upenn.edu/hybrid/
February 25, 2002

Abstract

The DARPA MoBIES Automotive Vehicle-Vehicle Open Experimental Platform [14] defines a longitudinal controller for the leader car of a platoon moving in an Intelligent Vehicle Highway System (IVHS) autonomously. The challenge is to verify that cars using this longitudinal controller provide a safe (that is, collision-free) ride. This report presents the process of verifying this particular controller using our CHARON [2] toolkit. In particular, it involves modeling and simulation of the system in CHARON, and verifying the controller using our predicate abstraction technique for hybrid systems [3].

1 Predicate Abstraction for Hybrid Systems

Inspired by the success of model checking in hardware verification and protocol analysis [11, 23], there has been increasing research on developing techniques for automated verification of hybrid (mixed discrete-continuous) models of embedded controllers [1, 4, 5, 20, 21]. The state-of-the-art computational tools for model checking of hybrid systems are of two kinds. Tools such as KRONOS [16], UPPAAL [26], and HyTech [22] limit the continuous dynamics to simple abstractions such as rectangular inclusions (e.g. $\dot{x} \in [1,2]$), and compute the set of reachable states exactly and effectively by symbolic manipulation of linear inequalities. On the other hand, emerging tools such as Checkmate [9], $d/dt$ [6], and level-sets method [19, 27], approximate the set of reachable states by polyhedra or ellipsoids [25] by optimization techniques. Even though these tools have been applied to
interesting real-world examples after appropriate abstractions, scalability remains a challenge.

In the world of program analysis, predicate abstraction has emerged to be a powerful and popular technique for extracting finite-state models from complex, potentially infinite state, discrete systems [10, 13, 15, 18]. A verifier based on this scheme requires three inputs, the (concrete) system to be analyzed, the property to be verified, and a finite set of boolean predicates over system variables to be used for abstraction. An abstract state is a valid combination of truth values to the boolean predicates, and thus, corresponds to a set of concrete states. There is an abstract transition from an abstract state \( A \) to an abstract state \( B \), if there is a concrete transition from some state corresponding to \( A \) to some state corresponding to \( B \). The job of the verifier is to compute the abstract transitions, and to search in the abstract graph for a violation of the property. If the abstract system satisfies the property, then so does the concrete system. If a violation is found in the abstract system, then the resulting counter-example can be analyzed to test if it is a feasible execution of the concrete system. This approach, of course, does not solve the verification problem by itself. The success crucially depends on the ability to identify the "interesting" predicates, and on the ability of the verifier to compute abstract transitions efficiently. Nevertheless, it has led to opportunities to bridge the gap between code and models and to combine automated search with user's intuition about interesting predicates. Tools such as Bandera [12], SLAM [7], and Feaver [24] have successfully applied predicate abstraction for analysis of C or Java programs.

Inspired by these two trends, we develop algorithms for invariant verification of hybrid systems using discrete approximations based on predicate abstractions. Consider a hybrid automaton with \( n \) continuous variables and a set \( L \) of locations. Then the continuous state-space is \( L \times \mathbb{R}^n \). For the sake of efficiency, we restrict our attention where all invariants, switching guards, and discrete updates of the hybrid automaton are specified by linear expressions, and the continuous dynamics is linear, possibly with bounded input. For the purpose of abstraction, the user supplies initial predicates \( p_1 \ldots p_k \), where each predicate is a polyhedral subset of \( \mathbb{R}^n \). In the abstract program, the \( n \) continuous variables are replaced by \( k \) discrete boolean variables. A combination of values to these \( k \) boolean variables represents an abstract state, and the abstract state space is \( L \times \mathbb{B}^k \). Our verifier performs an on-the-fly search of the abstract system by symbolic manipulation of polyhedra.

The core of the verifier is the computation of the transitions between abstract states that capture both discrete and continuous dynamics of the
original system. Computing discrete successors is relatively straightforward, and involves computing weakest preconditions, and checking non-emptiness of an intersection of polyhedral sets. For computing continuous successors of an abstract state $A$, we use a strategy inspired by the techniques used in CHECKMATE and $d/dt$. The basic strategy computes the polyhedral slices of states reachable from $A$ at fixed times $r, 2r, 3r, \ldots$ for a suitably chosen $r$, and then, takes the convex-hull of all these polyhedra to over-approximate the set of all states reachable from $A$. However, while tools such as CHECKMATE and $d/dt$ are designed to compute an approximation of the continuous successors of $A$, we are interested in testing if this set intersects with a new abstract state. Consequently, our implementation differs in many ways. For instance, it checks for nonempty intersection with other abstract states of each of the polyhedral slices, and omits steps involving approximations using orthogonal polyhedra and termination tests.

Postulating the verification problem for hybrid systems as a search problem in the abstract system has many benefits compared to the traditional approach of computing approximations of reachable sets of hybrid systems. First, the expensive operation of computing continuous successors is applied only to abstract states, and not to intermediate polyhedra of unpredictable shapes and complexities. Second, we can prematurely terminate the computation of continuous successors whenever new abstract transitions are discovered. Finally, we can explore with different search strategies aimed at making progress in the abstract graph. For instance, our implementation always prefers computing discrete transitions over continuous ones. Our early experiments indicate that improvements in time and space requirements are significant compared to a tool such as $d/dt$.

Using the theory of reachability analysis of hybrid systems via predicate abstraction [3] we perform the verification of a longitudinal controller for the leader car of a platoon from the IVHS projects [14, 17, 28]. Our concrete model consists of 4 continuous variables, linear dynamics with one bounded input, and 11 initial predicates. The verifier could establish absence of collisions without using any significant computational resources. In the next section we will briefly introduce the longitudinal controller [17] that we consider. We will conclude by describing the whole process of verifying this particular controller using the CHARON toolkit. The source code used to verify the absence of collisions of the longitudinal controller is given in the Appendices A – E.
2 Vehicle Coordination

We have successfully applied our predicate abstraction technique to verify a longitudinal controller for the leader car of a platoon moving in an Intelligent Vehicle Highway System (IVHS) as it is defined in the DARPA MoBIES Vehicle-Vehicle Open Experimental Platform [14]. Let us briefly describe this system. In the leader mode all the vehicles inside a platoon follow the leader. We consider a platoon $i$ and its preceding platoon $(i-1)$. Let $v_i$ and $a_i$ denote respectively the velocity and acceleration of platoon $i$, and $d_i$ is its distance to platoon $(i-1)$. The most important task of a controller for the leader car of each platoon $i$ is to maintain the distance $d_i$ equal to a safety distance $D_i = \lambda_a a_i + \lambda_v v_i + \lambda_p$ (in the nominal operation $\lambda_a = 0s^2$, $\lambda_v = 1s$, and $\lambda_p = 10m$). Other tasks the controller should perform are to track an optimal velocity and trajectories for certain maneuvers. The dynamics of the system are as follows:

\[
\begin{align*}
\dot{d}_i &= v_{i-1} - v_i \\
\dot{v}_{i-1} &= a_{i-1} \\
\dot{a}_i &= a_i \\
\dot{u}_i &= u,
\end{align*}
\]

where $u$ is the control. Without going into details, the controller for the leader car of platoon $i$ proposed in the MoBIES Vehicle-Vehicle Open Experimental Platform consists of 4 control laws $u$ which are used in different regions of the state space. These regions are defined based on the values of the relative velocity $v_i^e = 100(v_{i-1} - v_i)/v_i$ and the error between the actual and the safe inter-platoon distances $e_i = d_i - D_i$. When the system changes from one region to another, the control law should change accordingly. The property we want to verify is that a collision between platoons never happens, that is, $d_i > 0m$. Here, we focus only on two regions which are critical from a safety point of view: “track optimal velocity” ($v_i^e \leq -10$ and $e_i \geq -1m - \epsilon$) and “track velocity of previous car” ($v_i^e \leq -10$ and $e_i \leq -1m$). We include a thickening parameter $\epsilon > 0m$ into the model to add non-determinism to it. The two regions under consideration overlap allowing the controller to either use the “track optimal velocity” controller or the “track velocity of previous car” controller in this $\epsilon$-thick region. Besides adding some non-determinism to the model, it also provides improved numerical stability to the simulation and reachability computation, as it is numerically hard to determine the exact time at which a switch occurs.
The respective control laws $u_1$ and $u_2$ are as follows:

\[
\begin{align*}
u_1 & = 0.125d_i + 0.75v_{i-1} - (0.75 + 0.125\lambda_v)v_i - 1.5a_i - 0.125\lambda_p \\
u_2 & = d_i + 3v_{i-1} - (3 + \lambda_v)v_i - 3a_i - \lambda_p.
\end{align*}
\]

Note that these regions correspond to situations where the platoon in front moves considerably slower and, moreover, the second region is particularly safety critical because the inter-platoon distance is smaller than desired.

3 Verification using CHARON

This section describes the several steps involved when verifying properties of hybrid systems using CHARON. These steps are modeling, assertion checking and testing using simulations, translation of the CHARON model into the verifier input language, specifying the properties and predicates to be used, and, finally, running the predicate abstraction model checker given these inputs.\(^1\)

3.1 Modeling the Longitudinal Controller in CHARON

A hybrid system is described in CHARON by a set of agents communicating over a set of shared variables in an asynchronous way. The agents may be grouped together in a hierarchical way into composite agents starting from the most primitive ones called atomic agents. Information flow inside a composite agent may be hidden to the outside world. The grouping of agents into composite agents gives the architecture of the hybrid system.

Atomic agents may be endowed with a set of parameters that can be instantiated in different ways. Thus an atomic or composite agent may also be understood as an architectural pattern that may be instantiated, i.e., reused in different contexts that match the pattern. For example, at a lower level, a robot may be understood as the composition of a sensing agent, a controller agent, and an actuator agent. At a higher level, one may consider a team of cooperating robots, communicating with each other in order to achieve a common goal.

The behavior of an atomic agent is given by a set of modes that are linked together by a set of transitions. Each mode represents a particular behavior

\(^1\)This report concentrates on the specific example of the longitudinal controller and its modeling and verification using CHARON. Though we try to include as much general information about the CHARON language and the CHARON toolkit as possible, this cannot be done in a complete manner. Please refer to the CHARON user manual [8] for general guidelines on how to use the CHARON toolkit.
of the agent and has an associated dynamics given by a set of algebraic and differential constraints. The dynamics may be further constrained by a set of invariants. Modes may also be grouped together in a hierarchical way to form composed modes starting from the most primitive ones called leaf modes. Moreover, each mode may declare its own set of local variables that is hidden outside the mode, but is accessible to its submodes.

There are two ways of specifying CHARON models. The user is allowed to specify the model either using the CHARON visual interface (see Figure 1) or the CHARON textual editor (see Figure 2). We will concentrate on the textual version in this paper.

There are, of course, several ways of modeling the previously described longitudinal controller in CHARON. In order to present some features of the CHARON toolkit, we will present a model of the longitudinal controller as if it was designed in CHARON from scratch - in contrast to model the flattened system as described in [17]. The complete model of the longitudinal controller in CHARON is presented in Appendix A.\(^2\)

The architectural hierarchy of the longitudinal platoon control system is shown in Figure 3. The agent PLATON-i consists of two sub-agents,

\(^2\)There are a few assumptions made by the CHARON development environment that are often not followed by new users. One such assumption is that the name of the top level agent coincides with the name of the project. For example, the top level agent given in Appendix A is named platoon. Hence, the name of the corresponding project is also platoon, and the corresponding project file is platoon.prj. It's usually a good idea to name the corresponding main CHARON file platoon.cn. Please refer to the CHARON user manual [8] for more information about this and other requirements.
Figure 2: The editor frame on the right hand side of the CHARON desktop and the corresponding project frame on the left.

Figure 3: The architectural hierarchy of the system agent platoon
namely VELOCITY and CONTROLLER. The sub-agent CONTROLLER models the control laws and outputs the acceleration $a_i$ of the platoon $i$. The sub-agent VELOCITY takes as input the variable $acc$ and updates the variable vel of the platoon $i$. The agent PLATOON-$(i-1)$, whose role is to model all possible behaviors of the platoon in front, outputs its own velocity (variable vel) to the agent PLATOON-$i$. In other words, the velocity (or acceleration) of the platoon $(i-1)$ can be seen as uncertain input (or external disturbance) to the agent PLATOON-$i$.

Each agent has a well-defined interface which consists of its typed input and output variables, represented visually as blank and filled squares, respectively. The two variables vel of the agents PLATOON-$(i-1)$ and PLATOON-$i$ are inputs to the agent DISTANCE which outputs the variable dist representing the distance between the two platoons. The sub-agent CONTROLLER of PLATOON-$i$ computes the desired acceleration $a_i$ based on the inter-platoon distance and the velocity of the platoon in front.

Figure 3 illustrates the three operations defined on agents. Agents can be composed in parallel with each other. The parallel agents execute concurrently and communicate through shared variables. To enable communication between the two vehicles, global variables are renamed. For example, variables vel of agents PLATOON-$(i-1)$ and PLATOON-$i$ are renamed into velInFront and velBehind, respectively, so that the agent DISTANCE can read them without confusion. Finally, the communication between the vehicles can be hidden from the outside world. In our example, only the variable vel is the output of the PLATOON-$i$ agent. The variable acc, used internally by the agent PLATOON-$i$, cannot be accessed from outside the PLATOON-$i$ agent.

Modes represent behavioral hierarchy in the system design. The behavior of each atomic agent is described by a mode, which corresponds to a single thread of discrete control. Each mode has a well-defined data interface consisting of typed global variables used for sharing state information, and also a well-defined control interface consisting of entry and exit points, through which discrete control enters and exits the mode.3

3 To summarize, we put the CHARON model (see Appendix A) into perspective with respect to the model described in section 2: The variable dist in the CHARON model represents $d_i$, the variable acc, which will later also be referred to as follow.acc, represents $a_i$. The variable vel in the agent PLATOON-$(i-1)$ is also referred to as velLead in the composite system agent platoon and stands for $v_{i-1}$. Finally, the variable vel in the agent PLATOON-$i$ is also referred to as velFollow in the composite system agent platoon and represents $v_i$.
3.2 Assertion Checking and Testing using Simulation

A CHARON specification describes how a hybrid system behaves over the course of time. CHARON's simulator provides a means to visualize a possible behavior of the system. This information can be used for debugging or simply for understanding in detail the behavior of the given hybrid system description.

The simulation methodology used in the CHARON toolkit, which is depicted in Figure 4, resembles concepts in code generation from a specification. As CHARON allows to write external Java source code the simulator needs to be an executable Java program. CHARON has a set of Java files that represent a core simulator. Given a set of CHARON files, Java files are automatically generated which represent a Java interpretation of the CHARON specification of a hybrid system. They are used in conjunction with the predefined simulator core files and the external Java source code to produce a simulation trace.

The CHARON plotter allows the visualization of a simulation trace generated by the simulator. It draws the value of all selected variables using various colors with respect to time. It also highlights the time that selected transitions have been taken. The simulation results obtained in Figures 5-9 have been produced using the CHARON plotter.

In addition, the simulator checks assertions that are inserted at any place into the CHARON model by the user. Assertions can be added to any mode or agent in the model. They are state predicates over the variables of the mode or agent and are supposed to be true whenever the mode is active or, for agents, always. If an assertion is violated during a simulation, the simulator stops and the trace produced by the simulator can be used to find
We now consider simulation traces of the platoon controller under normal conditions. The Figures 5 – 7 show the simulation results for the CHARON model as given in Appendix A.\footnote{As the model of the platoon system is nondeterministic, the user should expect some differences between various simulation traces.} We initialize the system with the following values: $v_{i-1} = 15\frac{m}{s}$, $v_i = 25\frac{m}{s}$, $a_i = 0\frac{m}{s^2}$, and $d_i = 32m$.\footnote{These simulation traces have been generated using the following simulator options settings: 600 integration steps using 0.01s as the time-step. Please consult the CHARON manual [8] to learn how to set these values and how to change the visualization of simulation traces.} As can be seen in Appendix A, we use the \texttt{CHARON init} statement to initialize a system for simulation purposes. It should be noted though that these \texttt{init} statements do not affect the verification procedure. Later we describe how to initialize a system for verification purposes.

Initially, the platoon $i$ moves faster than its preceding platoon $(i - 1)$ and is too close to it given the current speeds. The controller being in the "track velocity of previous car" mode, decelerates considerably. After approximately 3s, the state space reaches the previously mentioned $\varepsilon$-thick region where both modes may be active. The system stays in this region for about 1s, and performs multiple switches between the two modes (see Figure 5). The simulation trace stops after approximately 4.3s, when the velocity of the preceding platoon $(i - 1)$ hits $0\frac{m}{s}$ and cannot be further reduced.\footnote{Notice, that the simulator picked one of the extreme values for $v_{i-1}$, as its is specified in the model in Appendix A. The velocity of the preceding platoon is decelerated by $3.5\frac{m}{s^2}$. The invariant of the agent representing the preceding platoon does not allow the simulation to continue, as the invariant $vel >= v\text{Min}$, with $v\text{Min} = 0.0\frac{m}{s}$, would be violated.}

Figure 8 shows a simulation trace, where the initial conditions are the same as before except for the initial distance, which is set to 5m instead of 32m. The simulation trace depicts the distance between the two platoons, and the zero-crossing of the distance at approximately 0.56s shows a crash of the two platoons.

Figure 9 shows the simulation result for another scenario. Initially, the distance between the two platoons is large, and the platoon $i$ is moving faster than the platoon in front $(i - 1)$ and is therefore closing the gap. We let the velocity of the platoon in front be a sinusoidal function of time starting at an initial value $20\frac{m}{s}$. To be able to generate this simulation trace, we needed to refine the model of the agent \texttt{PLATOON-}(i - 1) by adding a timer and a refined definition of $\hat{v}_{i-1}$. 
Figure 5: The acceleration $a_i$ of the platoon $i$. The dots represent the time that a transition between the controller modes occurred. The transitions are enabled only in the aforementioned $\varepsilon$-thick region. This plot shows that the system remained in this region between time $3.05s$ and $3.97s$.

Figure 6: The distance $d_i$ between the two platoons starting at the initial value of $32m$.

Figure 7: The velocity of the platoon $i$ and the preceding platoon $(i-1)$ (the platoon $i$ moves faster).
Figure 8: The distance $d_i$ between the two platoons starting at the initial value of 5m. A collision occurs approximately at time 0.56s.

Figure 9: A refined model for the agent $\text{PLATOON-}(i - 1)$ using a sinusoidal function. We show the velocity of the platoon $i$ and the preceding platoon $(i - 1)$ (the platoon $i$ moves faster). The simulation stops approximately at time 8.2s when the distance (not shown) equals the safety distance, and the two velocities are equal.
3.3 Verification of the Model

Our current prototype implementation of the predicate abstraction model checking tool is written in C++ using library functions of the hybrid systems reachability tool \texttt{d/dt}. Hence, we decided to reuse the \texttt{d/dt} input language for the predicate abstraction model checker. We implemented a translation routine from \textsc{Charon} source code to the \texttt{d/dt} input format. Therefore, it is now also possible to use the \texttt{d/dt} reachability tool on \textsc{Charon} source code. Before one can use the \texttt{d/dt} tool though, one may need to alter a parameter file (see Appendix D for the parameter file for this example). This parameter file contains information such as the size of the time-steps for computing the slices of the polyhedra or parameters for the visualization of the reachability analysis of the state space. The analysis tool \texttt{d/dt} also requires the user to specify a convex subset of the state-space $\mathbb{R}^n$ that is to be considered for the reachability.\footnote{The \texttt{d/dt} keyword for this is \texttt{limits}.} In addition, we need to specify the interesting regions of the state space, that is, we need to specify the initial region\footnote{The \texttt{d/dt} keyword for this is \texttt{initset}. We need to specify the region by defining a convex subset of the state space using a conjunction of linear predicates over the state variables. As mentioned earlier, the \textsc{Charon} \texttt{init} statement is used to specify the initial region for simulation purposes only. It is skipped in the translation from \textsc{Charon} source code to the \texttt{d/dt} input language. This is due to the fact that in simulation we want to specify exactly one initial state, whereas in verification we want to perform reachability analysis for a set of states.} as well as the region of the state space where the property is violated.\footnote{The \texttt{d/dt} keyword for this is \texttt{badset}.} The translation of the \textsc{Charon} source code given in Appendix A to the \texttt{d/dt} input language is provided in Appendix B. Appendix C shows the whole \texttt{d/dt} input language model after it has been enriched with the aforementioned necessary information.

As we mentioned before, currently we only have a prototype implementation of our predicate abstraction technique. Hence, we only perform reachability analysis for hybrid systems with linear continuous dynamics. The \textsc{Charon} model needs to be linear in order to be translated into the \texttt{d/dt} input language. This means that all the guards and invariants have to be linear relational expressions, the reset actions have to be linear expressions, and the dynamics of the modes need to be linear.

The translation of a \textsc{Charon} model can be initiated by choosing the menu item \texttt{Convert} in the menu \texttt{Reachability} in the top-level menu \texttt{Check}. This will create a new file in the project directory. The file will be named after the project, and will have the ending \texttt{.hyb}. For example, in our model
given in Appendix A, which is named the platoon project, a new file will be created in the same project directory with name `project.hyb`. This is the file that has been referred to as the translated model, and is shown in Appendix B. If the charon model does not conform to the rules governing the translation, such as linearity of the model, it cannot be translated, and the user is notified through an error message when trying to translate the model.

Certain information that is needed by \( d/dt \) for the reachability analysis by \( d/dt \) or the predicate abstraction model checker has to be added to the translated `.hyb`-file by the user. This includes the initial region of the continuous state space for the reachability analysis. We are considering the following initial region:\(^{10}\)

\[
20 \leq d_i \leq 100, -1 \leq a_i \leq i, 15 \leq v_{i-1} \leq 18, 20 \leq v_i \leq 25. \tag{4}
\]

As the ordering of the variables in the `.hyb`-file is `follow.acc, dist, velFollow, velLead` (see second line in Appendix B), the relational expression

\[
c_0a_i + c_1d_i + c_2v_i + c_3v_{i-1} + c_4 \sim 0,
\]

with \( \sim \in \{\leq, \geq\} \) is written as:

\[
\sim \ c_0 \ c_1 \ c_2 \ c_3 \ c_4.
\]

The constraint \( d_i \leq 100 \) is hence written as \( \leq 0.0 \ 1.0 \ 0.0 \ 0.0 \ -100.0 \).

The initial region given in (4) is hence written as the following code fragment, which is taken from Appendix C:

```plaintext
initset:
>= 0.0 1.0 0.0 0.0 -20.0 , /* dist -20 >= 0 */
<= 0.0 1.0 0.0 0.0 -100.0 , /* dist -100 <= 0 */
>= 1.0 0.0 0.0 0.0 1.0 , /* follow.acc +1 >= 0 */
<= 1.0 0.0 0.0 0.0 -1.0 , /* follow.acc -1 <= 0 */
>= 0.0 0.0 0.0 1.0 -15.0 , /* velLead -15 >= 0 */
<= 0.0 0.0 0.0 1.0 -18.0 , /* velLead -18 <= 0 */
>= 0.0 0.0 1.0 0.0 -20.0 , /* velFollow -20 >= 0 */
<= 0.0 0.0 1.0 0.0 -25.0 , ; /* velFollow -25 <= 0 */
```

\(^{10}\)Please note, that the reachability analysis does not necessarily start with this initial region. Depending on the predicates used, the tool performs a reachability analysis from all abstract states that intersect with the specified initial region. Hence, we over-approximate the reachable set.
Additionally, we want to specify the region of the state space that is considered a violation of the property to be verified – we call this region the “bad region” of the state space. In our longitudinal controller example, the safety property is violated, if the distance between the two platoons decreases to zero. The relational expression $d_i \leq 0$ is written in the $\text{d/dt}$ input file as follows (see Appendix C):

```
badset:
  <= 0.0 1.0 0.0 0.0 0.0 , /* dist <= 0 */
```

Finally, $\text{d/dt}$ requires the user to specify a subset of the continuous universe $\mathbb{R}^n$ that is to be considered for the reachability analysis. These “limits” of the universe have to be specific with respect to the example being verified. In our example, we use the following limits on the state space (see Appendix C):

```
limits:
  x[0] <= 10.0 and
  x[0] >= -20.0 and /* -20 <= follow.acc <= 10 */
  x[1] <= 150 and
  x[1] >= -5.0 and /* -5 <= dist <= 150 */
  x[2] <= 30.0 and
  x[2] >= 0.0 and /* 0 <= velFollow <= 30 */ /* as invariant */
  x[3] <= 50.0 and
  x[3] >= 0.0 /* 0 <= velLead <= 50 */ /* as invariant */
```

As was mentioned earlier, $\text{d/dt}$ also requires certain parameter information. These are stored in a separate parameter file. The parameter file used for our running example, the longitudinal controller, can be found in Appendix D. The parameter file needs to have the same name as the project name, but using the ending .par. The parameter file for the longitudinal controller is hence called platoon.par.

To be able to use our predicate abstraction model checker tool, the user needs to specify linear predicates in addition to the input needed for $\text{d/dt}$.

---

**Footnotes:**

11. Please note the specific syntax used for describing the limits in $\text{d/dt}$. The ordering of the variables remains the same as before (see line 2 in Appendix B).

12. The user may start a new parameter file by copying the one given in Appendix D. The user may need to change the parameter file to reflect the correct dimensionality of the corresponding system, and the number of states in the model (see the comments included in Appendix D). The parameter file may need other alterations as well though to fit the model being verified.

13. The keyword used for this in our predicate abstraction model checker is predicateset.
The predicates could, for example, include all the guards and invariants of the system. Additionally, the user may specify other predicates that may be important for the verification of the current property. We provide the whole input model needed for our predicate abstraction tool in Appendix E.

The current default implementation adds all invariants and guards automatically into the set of predicates to be used for the reachability analysis via predicate abstraction.\(^{14}\) Hence, we only need to add other predicates that we deem important for the reachability analysis. For example, we will (for obvious reasons) include the predicates that are specified in the \texttt{badset}, which is in our case \(d_i \leq 0\). We additionally add more predicates over the distance variable to be able to separate the bad region from the reachable set: \(d_i - 2 \geq 0, d_i - 10 \geq 0, d_i - 20 \geq 0\). This translates to the following additions to the \texttt{.hyb-file} (see Appendix E):

\begin{verbatim}
predicateset:
>= 0.0 1.0 0.0 0.0 -20.0 ,  /* dist - 20 >= 0 */
>= 0.0 1.0 0.0 0.0 -10.0 ,  /* dist - 10 >= 0 */
>= 0.0 1.0 0.0 0.0 -2.0 ,  /* dist - 2 >= 0 */
<= 0.0 1.0 0.0 0.0 0.0 ,  /* dist <= 0 */
;
\end{verbatim}

As it was shown in [3], if the predicate abstraction model checker finds that the property holds in the abstract state space, then we know that the property also holds in the concrete state space. On the other hand, if the tool finds a counter-example in the abstract state space, we need to check whether the found counter-example corresponds to a real counter-example in the concrete state space. We are currently working on an implementation of the counter-example checking capability.

To conclude, we will briefly review the generated \texttt{d/dt} input format from the \texttt{CHARON} model, and the verification of the model using our predicate abstraction tool. The model of this system in the \texttt{d/dt} input format consists of a hybrid automaton with 4 continuous variables \((d_i, v_{i-1}, v_i, a_i)\) and two locations corresponding to the two regions. The continuous dynamics of each location is linear as specified above, with \(u\) specified by (2) and (3). To prove that the controller of the leader car of platoon \(i\) can guarantee that no collision happens regardless of the behavior of platoon \((i-1)\), \(a_{i-1}\) is treated as \textit{uncertain input} with values in the interval \([a_{\min}, a_{\max}]\) where \(a_{\min}\) and \(a_{\max}\) are the maximal deceleration and acceleration.\(^{15}\) The invariants of the

\(^{14}\) Adding them again into the \texttt{predicateset} section of the \texttt{.hyb-file} is possible though, as we perform a check of the set of predicates prior to starting the reachability analysis.

\(^{15}\) In the here presented example in Appendix A we use \(a_{\min} = -3.5 \text{ m/s}^2\) and \(a_{\max} = 2 \text{ m/s}^2\).
locations are defined by the constraints on $e_i$ and $v_i^c$ and the bounds on the velocity and acceleration. The bad set is specified as $d_i \leq 0$. To construct the discrete abstract system, we use 11 predicates in total. For the initial set specified as $20 \leq d_i \leq 100, -1 \leq a_i \leq i, 15 \leq v_{i-1} \leq 18, 20 \leq v_i \leq 25$, the tool found 14 reachable abstract states and reported that the system is safe.\footnote{To start the predicate abstraction tool, please execute the executable \texttt{boolreach} using the \texttt{.hyb-file} as its argument. For example, the verification of the longitudinal controller would be started by typing \texttt{boolreach platoon/platoon.hyb}. The predicate abstraction tool will report its results as textual output. This could either be the fact, that the property holds, or if it does not hold, a counter-example in the abstract state space.}

A The CHARON Model of the Longitudinal Controller

agent DISTANCE ( real initialDistance )
{
    write analog real dist ; // dist is the output variable of the agent DISTANCE
    read analog real velInFront, velBehind ; // these are input variables

    init { dist = initialDistance ; } // initializing the variable dist

    mode top = DistanceTopMode () ; // the behavior is defined in DistanceTopMode
}

agent PLATOON_i_MINUS_1 ( real aMin, real aMax, real vMin, real vMax, real vInit )
{
    write analog real vel ;

    init { vel = vInit ; }

    mode top = LeaderTopMode ( aMin, aMax, vMin, vMax ) ;
}

agent VELOCITY ( real vMin, real vMax, real vInit )
{

\[16]
write analog real vel;
read analog real acc;

init { vel = vInit; }

mode top = VelocityTopMode (vMin, vMax);
}

agent CONTROLLER (real lambdaP, real lambdaV, real epsilon, real aInit) {
write analog real acc;
read analog real vel, dist, velInFront;

init { acc = aInit; }

mode top = ControllerTopMode (lambdaP, lambdaV, epsilon);
}

/* The agent PLATOON_i consists of two sub-agents, namely CONTROLLER and
* VELOCITY. The variable acc is used only between these two sub-agents, hence
* it is declared to be local (private). Other agents in the system do not
* know of the existence of acc.
*/
agent PLATOON_i (real vMin, real vMax, real vInit, real lambdaP, 
    real lambdaV, real epsilon, real aInit) {
private analog real acc;
write analog real vel;
read analog real velInFront, dist;

agent ctrl = CONTROLLER (lambdaP, lambdaV, epsilon, aInit);
agent car = VELOCITY (vMin, vMax, vInit);
}

/* The agent platoon is the agent that represents the combined system. To
* facilitate communication between the various agents, we add so called
* variable renamings to the agent definitions. For example, the renaming
* [ velInFront, velBehind := velLead, velFollow ] for the DISTANCE agent
* means, that the variable velInFront, which is used in the DISTANCE agent,
* will be called velLead in this agent. Similarly, velBehind in DISTANCE
* will be referred to as velFollow in the platoon agent.
*
agent platoon ()
{
  private analog real dist, velLead, velFollow;

agent lead = PLATOON_i_MINUS_1 ( -3.5, 2, 0, 50, 15 )
  [ vel := velLead ];

agent distance = DISTANCE ( 32 )
  [ velInFront, velBehind := velLead, velFollow ];

agent follow = PLATOON_i ( 0, 30, 25, 10, 1, 0.2, 0.0 )
  [ vel, velInFront := velFollow, velLead ];
}

//---w--------------------------------------------------------
// Mode definitions follow
//---w--------------------------------------------------------

mode VelocityTopMode ( real vMin, real vMax )
{
  write analog real vel;  // this mode writes continuously to vel of type real
  read analog real acc;   // and reads the continuously changing variable acc

  diff { d(vel) == acc; }  // this would be written in TeX as: \dot{vel} = acc

  inv { vel >= vMin && vel <= vMax }  // this is an invariant of the mode
}
mode LeaderTopMode ( real aMin, real aMax, real vMin, real vMax )
{
    write analog real vel;
    diff { d(vel) <= aMax ; d(vel) >= aMin ; } // \dot{vel} \in [aMin, aMax]
    inv { vel >= vMin && vel <= vMax }
}

/* The mode ControllerTopMode is a hierarchical mode, and has two sub-modes. */
mode ControllerTopMode ( real lambdaP, real lambdaV, real epsilon )
{
    write analog real acc;
    read analog real vel, dist, velInFront;

    mode prev = TrackVelocityOfPreviousCarMode ( lambdaP, lambdaV );
    mode opt = TrackOptimalVelocityMode ( lambdaP, lambdaV, epsilon );

    trans from default to prev // initially, go to sub-mode prev
    when true
    do {} // the guard of this transition: true
    do {} // no variable resets

    trans from prev to opt
    when dist - lambdaV*vel >= lambdaP - 1.0 - epsilon
    do {}

    trans from opt to prev
    when dist - lambdaV*vel <= lambdaP - 1.0
    do {}
}

mode TrackVelocityOfPreviousCarMode ( real lambdaP, real lambdaV )
{
    write analog real acc;
    read analog real vel, dist, velInFront;

    diff { d(acc) == dist - 3.0*acc + 3.0*velInFront -
          (3.0+lambdaV)*vel - lambdaP; }

20
inv { dist - lambdaV * vel <= lambdaP - 1.0 &&
   10.0*velInFront - 9.0*vel <= 0.0 }
}

mode TrackOptimalVelocityMode ( real lambdaP , real lambdaV , real epsilon )
{
    write analog real acc ;
    read analog real vel , dist , velInFront ;

    diff { d(acc) == -1.5*acc - (0.75+0.125*lambdaV)*vel + 0.75*velInFront +
           0.125*dist - 0.125*lambdaP ; }

    inv { dist - lambdaV*vel >= lambdaP - 1.0 - epsilon &&
          10.0*velInFront - 9.0*vel <= 0.0 }
}

mode DistanceTopMode ()
{
    write analog real dist ;
    read analog real velInFront , velBehind ;

    diff { d(dist) == velInFront - velBehind ; }
}

B The d/dt Model of the Longitudinal Controller
translated from the CHARON Model

dimension: 4;
/* [follow.acc, dist, velFollow, velLead] */
contdim: 4;
discdim: 0;
paramdim: 0;
initloc: 0;
/* platoon.lead.top+platoon.distance.top+platoon.follow.ctrl.top.prev+platoon.fc
state : 0;
matrixA:
-3.0 1.0 -4.0 3.0,
0.0 0.0 -1.0 1.0,
1.0 0.0 0.0 0.0,
0.0 0.0 0.0 0.0;
matrixB: 1;
inputset: type convex_vert
-10.0 0.0 0.0 2.0 ,
-10.0 0.0 0.0 -3.5;
invariant:
>= 0.0 0.0 0.0 1.0 0.0 ,
<= 0.0 0.0 0.0 1.0 -50.0 ,
<= 0.0 1.0 -1.0 0.0 -9.0 ,
<= 0.0 0.0 -9.0 10.0 0.0 ,
>= 0.0 0.0 1.0 0.0 0.0 ,
<= 0.0 0.0 1.0 0.0 -30.0 , ;
transition:
/* label goplatoon.follow.ctrl.top.prev+platoon.follow.car.topplatoon.follow.ctrl.
label go5:
if in
guard:
  >= 0.0 1.0 -1.0 0.0 -8.8 , ;
goto 1
reset
1 0 0 0 0,
0 1 0 0 0,
0 0 1 0 0,
0 0 0 1 0
;

/* platoon.lead.top+platoon.distance.top+platoon.follow.ctrl.top.opt+platoon.fc
state : 1;
matrixA:
-1.5 0.125 -0.875 0.75,
0.0 0.0 -1.0 1.0,
1.0 0.0 0.0 0.0,
0.0 0.0 0.0 0.0;
matrixB: 1;
inputset: type convex_vert
-1.25 0.0 0.0 2.0 ,
-1.25 0.0 0.0 -3.5;
invariant:
  >= 0.0 0.0 0.0 1.0 0.0 ,
  <= 0.0 0.0 0.0 1.0 -50.0 ,
  >= 0.0 1.0 -1.0 0.0 -8.8 ,
  <= 0.0 0.0 -9.0 10.0 0.0 ,
  >= 0.0 0.0 1.0 0.0 0.0 ,
  <= 0.0 0.0 1.0 0.0 -30.0 , ;
transition:
  /* label goplatoon.follow.ctrl.top.opt+platoon.follow.car.topplatoon.follow.ctrl.t
label go6:
if in
guard:
  <= 0.0 1.0 -1.0 0.0 -9.0 , ;
goto 0
reset
  1 0 0 0 0 ,
  0 1 0 0 0 ,
  0 0 1 0 0 ,
  0 0 0 1 0
; 

C The whole d/dt Model of the Longitudinal Controller

dimension: 4;
  /* [follow.acc, dist, velFollow, velLead] */
contdim: 4;
discdim: 0;
paramdim: 0;
initloc: 0;

/* platoon.lead.top+platoon.distance.top+platoon.follow.ctrl.top.prev+platoon.fc
state : 0;
matrixA:
-3.0  1.0  -4.0  3.0,
0.0  0.0  -1.0  1.0,
1.0  0.0  0.0  0.0,
0.0  0.0  0.0  0.0;
matrixB: 1;
inputset: type convex_vert
-10.0 0.0 0.0 2.0 ,
-10.0 0.0 0.0 -3.5;
invariant:
  >= 0.0 0.0 0.0 1.0 0.0 ,
   <= 0.0 0.0 0.0 1.0 -50.0 ,
   <= 0.0 1.0 -1.0 0.0 -9.0 ,
   <= 0.0 0.0 -9.0 10.0 0.0 ,
   >= 0.0 0.0 1.0 0.0 0.0 ,
   <= 0.0 0.0 1.0 0.0 -30.0 ,
transition:
/* label goplatoon.follow.ctrl.top.prev+platoon.follow.car.topplatoon.follow.ctrl.
label go5:
if in
  guard:
    >= 0.0 1.0 -1.0 0.0 -8.8 , ;
goto 1
reset
  1 0 0 0 0 ,
  0 1 0 0 0 ,
  0 0 1 0 0 ,
  0 0 0 1 0
  ;

/* platoon.lead.top+platoon.distance.top+platoon.follow.ctrl.top.opt+platoon.fc
state : 1;
matrixA:
-1.5  0.125 -0.875  0.75,
0.0  0.0  -1.0  1.0,
1.0 0.0 0.0 0.0,
0.0 0.0 0.0 0.0;
matrixB: 1;
inputset: type convex_vert
-1.25 0.0 0.0 2.0 ,
-1.25 0.0 0.0 -3.5;
invariant:
  >= 0.0 0.0 0.0 1.0 0.0 ,
  <= 0.0 0.0 0.0 1.0 -50.0 ,
  >= 0.0 1.0 -1.0 0.0 -8.8 ,
  <= 0.0 0.0 -9.0 10.0 0.0 ,
  >= 0.0 0.0 1.0 0.0 0.0 ,
  <= 0.0 0.0 1.0 0.0 -30.0 , ;
transition:
  label goplatoon.follow.ctrl.top.opt+platoon.follow.car.topplatoon.follow.ctrl.t
  label go6:
  if in
  guard:
    <= 0.0 1.0 -1.0 0.0 -9.0 , ;
goto 0
reset
  1 0 0 0 0 ,
  0 1 0 0 0 ,
  0 0 1 0 0 ,
  0 0 0 1 0
; ;

/***********************************************************************************/
/*** Additions to the generated file are the following: ***/  
/***********************************************************************************/
/* now define the initial region of the continuous state space */
initset:
  >= 0.0 1.0 0.0 0.0 -20.0 , /* dist - 20 >= 0 */
  <= 0.0 1.0 0.0 0.0 -100.0 , /* dist - 100 <= 0 */
  >= 1.0 0.0 0.0 1.0 , /* follow.acc + 1 >= 0 */
  <= 1.0 0.0 0.0 0.0 -1.0 , /* follow.acc - 1 <= 0 */
  >= 0.0 0.0 0.0 1.0 -15.0 , /* velLead - 15 >= 0 */
badset: <= 0.0 1.0 0.0 0.0 0.0 , ; /* dist <= 0 */

limits:

D The d/dt Parameter File for the Longitudinal Controller

dimension: 4; /* change this value to correspond to first line in .hyb-file */

state: 0, /* add more states in the same notation to correspond to .hyb-file */
time_step 0.5
abs_tol 12 12 12 12
rel_tol 10
tmax 6
itermax 500
method sim

state: 1,
time_step 0.5
abs_tol 12 12 12 12
rel_tol 10
tmax 15
itermax 500
method sim
;

display 0 /* ddt_viewer */
iterreach 2
mult_win 0
polylib_priority qhull
verbose w
precision 1E-7
offset 1E-6
;

angle 60
colour 2
zoom 2 1.4 0
projection 0 1 2
rotation 0 0 0
viewing_mode 3
refresh 0
xmin -50
xmax 500
ymin -50
ymax 100
zmin -100
zmax 100
width 600
height 600
The whole Model for the Predicate Abstraction Model Checker

dimension: 4;
/* [follow.acc, dist, velFollow, velLead] */
contdim: 4;
discdim: 0;
paramdim: 0;

initloc: 0;

/* platoon.lead.top+platoon.distance.top+platoon.follow.ctrl.top.prev+platoon.fc state : 0;
matrixA:
-3.0 1.0 -4.0 3.0,
0.0 0.0 -1.0 1.0,
1.0 0.0 0.0 0.0,
0.0 0.0 0.0 0.0;
matrixB: 1;
inputset: type convex_vert
-10.0 0.0 0.0 2.0 ,
-10.0 0.0 0.0 -3.5;

invariant:
>= 0.0 0.0 0.0 1.0 0.0 ,
<= 0.0 0.0 0.0 1.0 -50.0 ,
<= 0.0 1.0 -1.0 0.0 -9.0 ,
<= 0.0 0.0 -9.0 10.0 0.0 ,
>= 0.0 0.0 1.0 0.0 0.0 ,
<= 0.0 0.0 1.0 0.0 -30.0 , ;

transition:
/* label goplatoon.follow.ctrl.top.prev+platoon.follow.car.top+platoon.follow.ctrl.car label go5:
if in
guard:
>= 0.0 1.0 -1.0 0.0 -8.8 , ;
goto 1
reset
1 0 0 0 0,
0 1 0 0 0,
0 0 1 0 0,
0 0 0 1 0
;

/*  platoon.lead.top+platoon.distance.top+platoon.follow.ctrl.top.opt+platoon.fol
state : 1;
matrixA:
-1.5 0.125 -0.875 0.75,
0.0 0.0 -1.0 1.0,
1.0 0.0 0.0 0.0,
0.0 0.0 0.0 0.0;
matrixB: 1;
inputset: type convex_vert
-1.25 0.0 0.0 2.0 ,
-1.25 0.0 0.0 -3.5;
invariant:
>= 0.0 0.0 0.0 1.0 0.0 ,
<= 0.0 0.0 0.0 1.0 -50.0 ,
>= 0.0 1.0 -1.0 0.0 -8.8 ,
<= 0.0 0.0 -9.0 10.0 0.0 ,
>= 0.0 0.0 1.0 0.0 0.0 ,
<= 0.0 0.0 1.0 0.0 -30.0 , ;
transition:
/* label goplatoon.follow.ctrl.top.opt+platoon.follow.car.topplatoon.follow.ctrl.t
label go6:
if in
guard:
<= 0.0 1.0 -1.0 0.0 -9.0 , ;
goto 0
reset
1 0 0 0 0 ,
0 1 0 0 0 ,
0 0 1 0 0 ,
0 0 0 1 0
;

;
initset:

\[ \begin{align*}
& \geq 0.0 \quad 1.0 \quad 0.0 \quad 0.0 \quad -20.0 \ , \quad \text{/* dist - 20 } \geq 0 */ \\
& \leq 0.0 \quad 1.0 \quad 0.0 \quad 0.0 \quad -100.0 \ , \quad \text{/* dist - 100 } \leq 0 */ \\
& \geq 1.0 \quad 0.0 \quad 0.0 \quad 0.0 \quad 1.0 \ , \quad \text{/* follow.acc + 1 } \geq 0 */ \\
& \leq 1.0 \quad 0.0 \quad 0.0 \quad 0.0 \quad -1.0 \ , \quad \text{/* follow.acc - 1 } \leq 0 */ \\
& \geq 0.0 \quad 0.0 \quad 0.0 \quad 1.0 \quad -15.0 \ , \quad \text{/* velLead - 15 } \geq 0 */ \\
& \leq 0.0 \quad 0.0 \quad 0.0 \quad 1.0 \quad -18.0 \ , \quad \text{/* velLead - 18 } \leq 0 */ \\
& \geq 0.0 \quad 0.0 \quad 1.0 \quad 0.0 \quad -20.0 \ , \quad \text{/* velFollow - 20 } \geq 0 */ \\
& \leq 0.0 \quad 0.0 \quad 1.0 \quad 0.0 \quad -25.0 \ , \quad \text{/* velFollow - 25 } \leq 0 */ \\
\end{align*} \]

badset:

\[ \begin{align*}
& \leq 0.0 \quad 1.0 \quad 0.0 \quad 0.0 \quad 0.0 \ , \quad \text{/* dist } \leq 0 */ \\
\end{align*} \]

/**************************
/* We need to add the predicates to our system */
/**************************
/* that should be used during the reachability. */
/**************************

predicateset:

\[ \begin{align*}
& \geq 0.0 \quad 1.0 \quad 0.0 \quad 0.0 \quad -20.0 \ , \quad \text{/* dist - 20 } \geq 0 */ \\
& \geq 0.0 \quad 1.0 \quad 0.0 \quad 0.0 \quad -10.0 \ , \quad \text{/* dist - 10 } \geq 0 */ \\
& \geq 0.0 \quad 1.0 \quad 0.0 \quad 0.0 \quad -2.0 \ , \quad \text{/* dist - 2 } \geq 0 */ \\
& \leq 0.0 \quad 1.0 \quad 0.0 \quad 0.0 \quad 0.0 \ , \quad \text{/* dist } \leq 0 */ \\
\end{align*} \]

limits:

\[ \begin{align*}
x[0] & \leq 10.0 \ \text{and} \\
x[0] & \geq -20.0 \ \text{and} \quad \text{/* -20 } \leq \text{follow.acc } \leq 10 */ \\
x[1] & \leq 150 \ \text{and} \\
x[1] & \geq -5.0 \ \text{and} \quad \text{/* -5 } \leq \text{dist } \leq 150 */ \\
x[2] & \leq 30.0 \ \text{and} \\
x[2] & \geq 0.0 \ \text{and} \quad \text{/* 0 } \leq \text{velFollow } \leq 30 */ \quad \text{/* as invariant */} \\
x[3] & \leq 50.0 \ \text{and} \\
x[3] & \geq 0.0 \ \text{and} \quad \text{/* 0 } \leq \text{velLead } \leq 50 */ \quad \text{/* as invariant */} \\
\end{align*} \]
References


