POWVER

Technical Report 2021-15

Title: **Controller Verification meets Controller Code: A Case Study**

Authors: Felix Freiberger, Stefan Schupp, Holger Hermanns, Erika Ábrahám

Report Number: 2021-15

ERC Project: Power to the People. Verified.

ERC Project ID: 695614

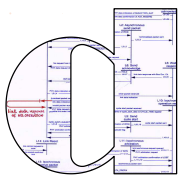
Funded Under: H2020-EU.1.1. – EXCELLENT SCIENCE

Host Institution: Universität des Saarlandes, Dependable Systems and Software
Saarland Informatics Campus

Published In: MEMOCODE '21

Please cite this publication as follows:

Felix Freiberger, Stefan Schupp, Holger Hermanns, Erika Ábrahám.
Controller Verification meets Controller Code: A Case Study.
19th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE '21), November 20–22, 2021, Beijing, China.



POWER TO THE PEOPLE.
VERIFIED.
erc

Controller Verification meets Controller Code: A Case Study

Felix Freiberger
Saarland University
Saarbrücken, Germany
Saarbrücken Graduate School of Computer Science
Saarbrücken, Germany

Stefan Schupp
TU Wien
Wien, Austria
RWTH Aachen University
Aachen, Germany

Holger Hermanns
Saarland University
Saarbrücken, Germany
Institute of Intelligent Software
Guangzhou, China

Erika Ábrahám
RWTH Aachen University
Aachen, Germany

ABSTRACT

Cyber-physical systems are notoriously hard to verify due to the complex interaction between continuous physical behavior and discrete control. A widespread and important class is formed by digital controllers that operate on fixed control cycles to interact with the physical environment they are embedded in. This paper presents a case study for integrating such controllers into a rigorous verification method for cyber-physical systems, using flowpipe-based verification methods to verify legally binding requirements for electrified vehicles to a custom bike design. The controller is integrated in the underlying model in a way that correctly represents the input discretization performed by any digital controller.

CCS CONCEPTS

• **Computer systems organization** → *Embedded and cyber-physical systems*; • **Theory of computation** → *Timed and hybrid models*; *Verification by model checking*; • **Software and its engineering** → *Software verification*; *Formal software verification*; • **Computing methodologies** → *Model verification and validation.*

KEYWORDS

Controller, hybrid automata, verification

ACM Reference Format:
Felix Freiberger, Stefan Schupp, Holger Hermanns, and Erika Ábrahám. 2021. Controller Verification meets Controller Code: A Case Study. In *19th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE '21), November 20–22, 2021, Beijing, China.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3487212.3487337

MEMOCODE '21, November 20–22, 2021, Beijing, China

ACM ISBN 978-1-4503-9127-6/21/11…$15.00
https://doi.org/10.1145/3487212.3487337

ACKNOWLEDGMENTS

This work was partially funded by the ERC Advanced Investigators Grant 695614 (POWVER) and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project number 389792660 – TRR 248 – CPEC, see https://perspicuous-computing.science).

1 INTRODUCTION

Cyber-physical systems are taking over ever more tasks. Their accumulated effect on our daily life and especially in mission- or safety-critical contexts grows massively. Despite an obvious need for establishing rigorous safety guarantees across many such contexts, cyber-physical systems *out there* have mostly eluded formal verification attempts thus far.

Many controllers act *periodically*, executing the same control logic repeatedly after a fixed time interval, mapping the sensor values and some controller-internal state to actuator commands and updates to the controller state. For instance, programmable logic controllers (PLCs) operate in this manner by default.

In this paper, we look at a real-world electrified dandy horse as an example of such a system. This is a custom-made, bicycle-like vehicle, but without pedals, so it is accelerated by the rider pushing her feet along the ground. In the actual prototype we are considering, an on-board controller senses the pushes and amplifies them by means of an electric motor, similar to how a traditional pedelec amplifies the force applied to the pedals.

In our case study, we verify legal requirements as well as practically relevant safety constraints for our dandy horse. Our approach is based on well-established verification techniques for hybrid automata, but is distinguished by its improved handling of the discrete controller. Here, the discrete behavior of the controller is extracted automatically from the controller code for direct inclusion in the model, eliminating the risk of errors introduced by modelling controllers manually.

The verification algorithm we use is rooted in state-of-the art methods for hybrid systems, namely reachability analysis based on flowpipe construction [1, 2, 4, 6, 10]. It is tailored to models combining continuous physical behavior and discrete controller behavior applied at fixed, recurring time points.

In contrast to related approaches, our method is automated and does not require explicit modelling of the controller. The controller

Figure 1: Draisine 200.0, a wooden dandy horse

can be treated as a black box, merely requiring knowledge of the inputs, outputs, and state variables which are preserved across control cycles. This considerably reduces modelling effort and, more importantly, sets the basis for verifying almost arbitrary controllers in the loop. This approach also opens a path towards analysis across intellectual property boundaries, requiring only the interface and controller state to be handled explicitly, without requiring disclosure of controller source code.

2 DRAISINE 200.0

In this paper, we focus on Draisine 200.0[1], a prototype human-powered and engine-assisted vehicle similar to a bicycle, shown in Figure 1. It is made of wood and has been built as a demonstrator for various scientific projects on the occasion of the 200$^{\text{th}}$ birthday of the original dandy horse.

The vehicle features a bicycle frame but is not equipped with pedals. Instead, a rider accelerates it by pushing herself forward with her feet on the ground. Braking force can be applied either by a traditional friction brake or by the rider's feet directly.

The vehicle is equipped with an electric motor in the rear wheel for the purpose of providing assistance to the rider, similar to a pedelec (an electrified bicycle that provides assistance when the rider pedals), which is powered by a battery mounted in the frame. A freewheel assembly ensures the wheel can turn faster than the motor, avoiding resistance while the motor is not powered. The low-level control of the motor (including all high-wattage power lines) is handled by a Vedder VESC 4.12, an off-the-shelf driver. The motor can be actuated via pulse width modulation: It expects to be given a target *duty signal*, i.e., a signal specifying the percentage of time during which the motor should be powered.

Determining the proper duty signal is the task of a dedicated controller. The overall goal of the controller system is to provide an experience analogous to a pedelec to the rider: She should be able to use the dandy horse normally, i.e., accelerate it by pushing, but receive assistance proportional to the pushes, such that higher speeds and larger distances are possible, without impacting the handling of the vehicle too much. Similar to a PLC, the controller is executed periodically after passage of Δ_t time units.

On the demonstrator, for flexibility, the controller is run on a Raspberry Pi computer powered by an additional battery (in a mass-produced vehicle, a dedicated microprocessor, powered by the main battery, would be used instead).

The controller is attached to a motion processing unit containing, amongst others, an accelerometer. This accelerometer can sense the longitudinal acceleration caused by the rider pushing or braking the vehicle, and will provide this information as input to the controller. Then, the controller has the task of briefly powering the engine to amplify that push.

In addition, the dandy horse contains a Hall-effect sensor which is mounted close to the rear wheel and wired to GPIO ports on the Raspberry Pi and four magnets mounted equidistantly along the perimeter of the rear wheel, allowing the controller to approximate the speed of the wheel with high accuracy. This enables the controller to determine the vehicle speed, so as to prevent accidental acceleration when the speed is near zero, and to ensure that the vehicle does adhere to regulations limiting the maximum speed up to which electrical assistance is allowed (25 km/h for pedelecs).

The objective is to verify the digital controller responsible for converting sensor readings into a duty signal. In this work, the driver is out of scope for verification and will instead be modelled as part of the physical system. Similarly, as a simplification, the speed estimation logic is also not verified (although our verification method could in general be used for that, at the cost of increased verification execution time). To this end, in the model, we assume the vehicle is equipped with a perfect speed sensor, while in practice, estimating the speed is handled by an external component that receives the Hall sensor events and passes an estimated speed to the controller core. The same component also handles communication with the hardware through the GPIO pins.

The to-be-verified core controller logic is implemented in C++. It is a stateful algorithm that uses the accelerometer input, speed, and a limited history of previous values of the accelerometer and duty signal stored in the controller state to compute the duty signal. The historic values of the accelerometer help to smooth out noise (caused by vibrations of the vehicle or uneven ground), providing an estimate of the current amount of acceleration caused by the user pushing the vehicle. The estimated speed is then used to shape this signal into a target duty signal, providing no motor support at very slow speeds (to prevent unwanted acceleration of a standing vehicle) or above the legal speed limit. At the same time, high motor support should be given at intermediate speeds, with smooth transitions between the different modes for increased rider comfort. The previous duty signal is used to limit the speed with which the actual duty signal can change. This not only serves to smooth out uncomfortable spikes in acceleration, but also ensures that a strong but short push by the user yields a slightly longer push by the motor, providing an acceptable support level overall.

[1] see https://www.powver.org/draisine-200-0/

Listing 1: The control algorithm of Draisine 200.0. Some sub-functions and declarations have been removed to save space.

```
double Controller::sig( double x ) const {
    return 1.0 / ( 1.0 + exp( -x ) );
}

double Controller::sig1( double x ) const {
    return sig( 2 * x * exp( 2 ) );
}

ControllerOutput control( const ControllerState state, const ControllerInput input ) const {
    constexpr int precision = 10;

    constexpr double threshold = 0.3;
    constexpr double accMax = 3.5;

    constexpr double maxDutyIncrease = 20.0 / precision;
    constexpr double maxDutyDecrease = 1.0 / precision;

    double speed = decodeSpeed( input.speedValue );
    double oldDuty = ( (double)state.dutySignal ) / 31;

    double lowPassedAcceleration = lowPassFilter(state.accelerometerHistory, input.accelerometerValue); // details
     ↳ skipped here

    double dutyPercentage = std::min( 1.0, ( 1.0 / ( 1.0 - threshold ) ) * std::max( 0.0, lowPassedAcceleration / accMax
     ↳  - threshold ) );

    // modulate duty signal to speed using constants v1 through v4
    const double modulationFactor = ( speed <= ( v2 + v3 ) / 2 )
                                  ? sig1( (  speed - ( v1 + v2 ) / 2 ) / ( v2 - v1 ) )
                                  : sig1( ( -speed + ( v3 + v4 ) / 2 ) / ( v4 - v3 ) );
    dutyPercentage *= modulationFactor;

    // limit changes in duty signal
    dutyPercentage = std::min( std::max( dutyPercentage, oldDuty - maxDutyDecrease )
                               oldDuty + maxDutyIncrease );
    DutySignalType duty = std::max( 0, std::min( 31, int( 31 * dutyPercentage ) ) );

    ControllerState newState( duty, accelerometerHistory );
    shiftAccelerometerHistory(accelerometerHistory, input.accelerometerValue); // details skipped here

    return ControllerOutput( newState, duty );
}
```

Note that the accelerometer not only measures the acceleration caused by the user, but also the acceleration caused by the motor.[2] This creates a potential for run-off acceleration that is countered by carefully tuning the parameters of the controller (a property which we will verify later on). The controller's source code is shown in Listing 1.

2.1 Properties

For this case study, we focus on properties of the form "When φ (*scenario*) is true continuously for t_{grace} (*grace period*), then ψ (*requirement*)".

This form of property is very relevant in practice, especially in legal safety requirements. For example, in many countries, pedelecs are only allowed to provide electric assistance up to a certain speed limit, often 25 km/h [3]. Normally, such rules do not require the device to cut off assistance instantaneously after the speed limit is hit, but provide some leeway (i.e., a grace period), allowing devices to detect the event and react smoothly.

[2] In contrast, acceleration caused by gravity while riding downhill (on a smooth surface) is not measured by the longitudinal accelerometer.

In this case study, we have identified three properties relevant for operation of the dandy horse, derived from the current regulations for pedelecs which we aim to verify:

- **No duty above speed limit.** When the speed of the vehicle has been higher than a certain threshold (here: 25 km/h) for one second, the duty signal must be zero.
- **No duty without rider input.** When the rider has not accelerated the vehicle beyond a certain threshold within the past second, the duty signal must be zero.
- **Braking stops the vehicle.** When the rider does not accelerate and engages the brakes for a certain amount of time, the speed must be zero.

Whether these properties hold is less obvious than it may appear. For instance, as the controller limits the rate with which the duty signal decreases, the former two properties cannot be verified without considering the controller state (as high speeds or lack of acceleration will only cause the controller to return a duty signal that is lower than before, but not necessarily zero).

2.2 Model

2.2.1 Plant. We model Draisine 200.0 with a hybrid automaton [5]

$$\mathcal{H} = (Loc, Var, Flow, Inv, Edge, Init)$$

with a single location $Loc = \{\texttt{running}\}$ and Var being:

- v: (longitudinal) velocity of the vehicle,
- a_{rider}: acceleration applied by the rider's feet,
- j_{rider}: jerk (change of acceleration) from the rider's feet,
- a_{terrain}: acceleration resulting from uneven terrain,
- j_{terrain}: jerk resulting from uneven terrain,
- p_{brake}: percentage of the available braking force currently being applied,
- Δp_{brake}: change rate of braking force,
- ds: last duty signal output from the controller,
- a_{history}: variable used by the controller to store previous measurements from the accelerometer, and
- t: time passed since the last control cycle.

For the single location `running`, the dynamics are set to

$$\dot{v} = ds \cdot c_{\text{motor}} + a_{\text{rider}} + a_{\text{terrain}} + p_{\text{brake}} \cdot c_{\text{brake}}$$

(where c_{motor} and c_{brake} are constants describing the power of the motor and brake, respectively), $\dot{a_{\text{rider}}} = j_{\text{rider}}$, $\dot{a_{\text{terrain}}} = j_{\text{terrain}}$, and $\dot{p_{\text{brake}}} = \Delta p_{\text{brake}}$.

Inv assigns the single location `running` a predicate constraining a_{rider}, j_{rider}, a_{terrain}, j_{terrain}, and Δp_{brake} to specific reasonable ranges describing our assumptions on the user's behavior and terrain, p_{brake} to the interval $[0, 1]$ and t to the interval $[0, \Delta_t]$. For practical reasons, we also restrict v to a reasonable range. The set of initial states *Init* is set depending on the property that needs to be verified. For instance, if a property requires the initial velocity to be larger than a certain value (here for instance 25 km/h), the initial condition for this property considers any state with velocity larger than the threshold which also satisfies the invariant.

2.2.2 Controller. Draisine 200.0 features a digital controllers that operates periodically: at equidistant time points (Δ_t), a *controller execution* reads sensor inputs, computes the implemented control function, and passes the computed values to the actuators.

The real-world implementation of such a controller implies two types of inherent discretization: The periodic execution causes the controller to act at discrete time points, and the interaction with the plant through digital actuators and sensors uses discrete values.

When it comes to verification, there are two methods to embed a controller into a hybrid automaton:

- The controller's influence can be integrated into the continuous behavior, i.e., the plant is directly and continuously influenced by the controller output, which in turn is directly and continuously influenced by the plant (see, e.g., [8, 9]). However, this method requires a model of the controller and for a digital controller, the continuous version in the model can only approximate the real system.
- Alternatively, the control function of the digital controller can be expressed through discrete jumps (see, e.g., [7]).

In our case study, we follow the second approach, and set *Edge* to be the set of edges that *exactly* describe our digital controller: For every combination of values the controller can read from its input and state variable, *Edge* contains exactly one jump (guarded to the appropriate ranges of input and controller state variables) describing the controller behavior for this input. The resulting structure of the hybrid automaton is sketched in Figure 3.

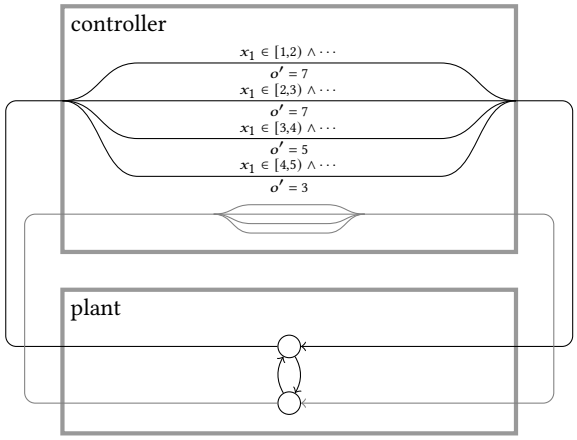


Figure 3: Structure of a hybrid automaton for a cyber-physical system with a discrete controller

In practice, the set *Edge* is not specified manually, but derived automatically from the controller source code by executing the controller with every possible input with respect to the controller-resolution.

2.3 Code Simplifications

To keep verification feasible within reasonable time bounds, it is crucial to reduce the resolution of the controller input, state, and output variables, i.e., to use a coarse discretization (corresponding to variables with few bits) and to keep as few state variables as possible. The controller used in Draisine 200.0 was designed to follow this principle from the start, empirically evaluating different resolutions and number of variables. The controller chosen for deployment to the vehicle was the one having the lowest resolution and variable count while still producing both sufficient support (maxing out the available motor power briefly when the rider accelerates quickly) and a subjectively smooth experience (avoiding sudden or unexpected changes in acceleration). This results in a controller with 5-bit unsigned speed values (input), 5-bit signed accelerometer values (input & state), and a 5-bit unsigned duty signal (output & input).

Similarly to minimizing variable resolution, the time resolution has also been minimized by running only 10 control cycles per second ($\Delta_t = 0.1$). This helps in two ways: Not only does it reduce the number of controller applications the verifier has to reason about, but it also means that the controller has to store fewer previous accelerometer values to reason about the same time frame. With this low cycle rate, keeping a single previous accelerometer reading has been empirically shown to reliably prevent accelerometer noise from causing unintended accelerations.

With these optimizations, the controller is a function that maps 984064 distinct pairs of inputs and states to the corresponding outputs and new states, i.e. the number of jumps required to represent the controller is $|Edge| = 984064$.

3 VERIFICATION

To verify our properties, we use a verification algorithm based on flowpipe construction [1, 2, 4, 6, 10]. Intuitively, the approach over-approximates the set of reachable states of a given hybrid system by sets of geometric shapes based on time discretization for the continuous behavior. Note that different types of shapes yield different properties with respect to execution time and precision. Commonly used shapes include boxes, convex polyhedra, support functions, and zonotopes.

The structure of our model and properties pose two main problems for verification:

(1) Despite the optimizations described in Section 2.3, our method of embedding the controller into the plant, while preserving the real-world value discretization, created a high number of jumps.
As processing the jumps requires computing an intersection with the guard of every single jump, this severely affects the execution time of a naïve verifier implementation.
(2) As the scenario condition may become satisfied at any point in time, including at any point *in between* controller applications, the controller applications are not "aligned" relative to the start of the grace period t_{grace}.
This means that a naïve verification algorithm cannot merely process the controller jumps $\frac{t_{\text{grace}}}{\Delta_t}$ times, but needs to consider them for every flowpipe segment.

Solving these problems is crucial to keep verification times feasible. We counter the first problem by computing an optimized internal representation of the jumps to use during verification. To represent controller jumps based on the discretization described in Section 2.2.2 while keeping execution time reasonable, we preprocess the discretization obtained from a full sampling of the controller and group neighboring controller inputs which yield the same output into so-called *chunks*. This effectively reduces the number of transitions which are required to model the controller. These chunks also lay the groundwork for a state set representation method based on sets of boxes (each of which is a sub-box of a chunk) that allows high efficiency while keeping the representation-induced overapproximations within boundaries with limited impact on the overall verification result.

The second problem is solved by an overapproximation based on the observation that independently of the initial state of the system (and the internal clock t triggering a controller application), a controller application is hit after passage of *at most* Δ_t time units. After this first controller application, the value of the clock is known, and state space exploration needs to consider the jumps only once per time segment of Δ_t. This is valid as the behavior of the draisine itself is purely dynamical, i.e. no other discrete jumps may happen in-between two controller applications. Therefore, the verification algorithm can be structured to only apply the controller at these intervals by overapproximating the time spent in the first control cycle to align controller applications. Due to the added overapproximation on the initial clock valuation, the last control cycle needs to be overapproximated, too, to compensate for the missing information on the actual clock valuation (and therefore missing information on when the grace period expires), which can range up to Δ_t. Consequently, the intersection with bad states must be checked during the whole last control cycle, not just at the time horizon.

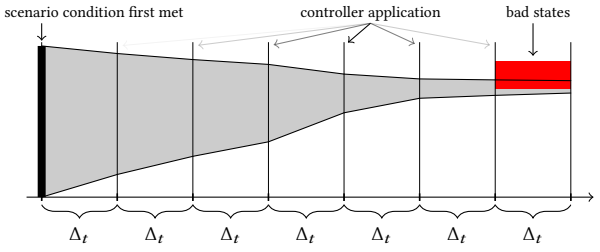


Figure 4: Overview of state space during verification. Analysis begins when the scenario condition is first met, starting the grace period. Overapproximations ensure the controller only needs to be applied at discrete time points (thin vertical lines). The behavior of the plant may change at these points due to changes to actuator values. Verification is successful if no bad states are encountered after the grace period has passed (which, due to overapproximation, must be assumed to be possible at any point within the last cycle).

The evolution of the sets of reachable states in the resulting verification algorithm is sketched in Figure 4.

4 EVALUATION

To verify the properties for Draisine 200.0 discussed in Section 2, we implemented a verification approach in C++ as described in Section 3. For the reachability analysis and state set representation, we used HyPro [10], a C++ programming library for hybrid systems safety verification via flowpipe-construction-based reachability analysis. All experiments were performed on an Intel® Core™ i7-4790 CPU (3.60 GHz) (with 4 physical / 8 logical threads) running Ubuntu 20.04 with 32 GB of memory.

4.1 Experimental Results

Our goal was to analyze the three properties in Section 2.1.

As described in Section 3, our verification algorithm uses a preprocessed representation of the set *Edge*. This preprocessing step only needs to be run once when the controller code is changed and is not considered in the following discussion. This process takes about 30 s on our test machine. As the controller outputs are precomputed and stored in this step, all execution times discussed in the following no longer depend on the complexity of the control algorithm itself, only on the chunks describing it.

In addition to measuring the influence of the number of chunks during our evaluation, we evaluate the influence of the time step size for the time discretization and the used state set representation during flowpipe construction with respect to overall execution time and precision of the results. While the effect of the cycle time was negligible in terms of precision and execution time when using boxes as a state set representation, using support functions instead resulted in infeasible execution times. The presented results thus show execution times using a box-based representation and a time discretization of $\delta = 0.01 = \frac{\Delta_t}{10}$ for the flowpipe construction. In

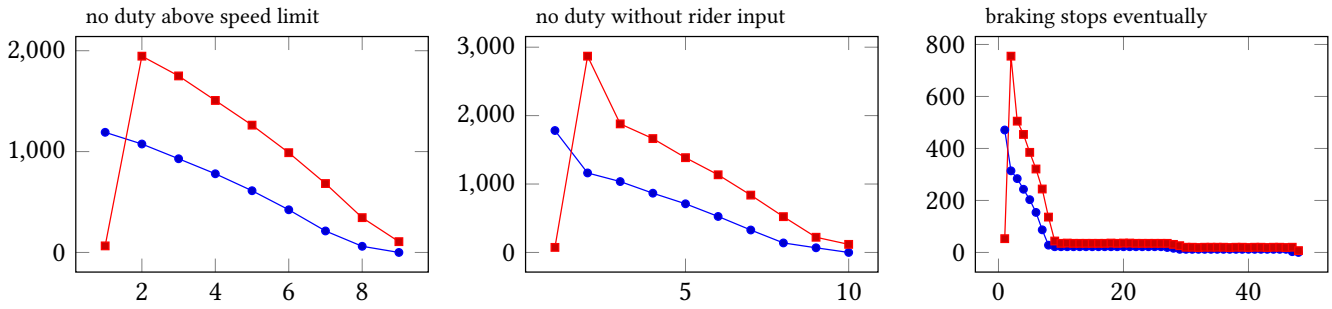


Figure 5: Execution time (—■—, in milliseconds, y axis) and number of chunks produced (—●—, y axis) per cycle (x axis). The time in the first cycle is always low because the initial state is represented by a single box; the number of boxes generated influences the number of intersections needed in the *next* cycle.

Table 1: Verification execution times for different properties with different grace periods (GP).

Property	**GP** $[\Delta_t]$	**runtime** [s]
no duty above speed limit	9	8.7
no duty without rider input	10	10.7
braking stops eventually	50	4.0

this setting, all properties take less than 15 s to verify, with exact execution times shown in Table 1.

For our verifier, execution time is dominated by processing the jumps induced by the controller. Consequently, the majority of the execution time is spent in the first control loop cycles, and as the scenario condition and the resulting behavior of the controller cut off otherwise reachable states (cf. Figure 4), analysis speeds up, shown in Figure 5. This also explains that properties with stricter scenario conditions are analyzed faster as larger parts of the sets of reachable states are discarded quickly during analysis as they do no longer satisfy the scenario conditions.

4.1.1 Parallelization. Because execution time is dominated by processing the jumps, a problem that lends itself to parallelization, a parallelized verifier can achieve a significant speedup. In our experiments, verification reached an overall speedup of 4.24, measured by comparing execution time between single-threaded and multi-threaded runs. This speedup is higher than 4, the number of physical cores of the test machine. (Restricting the number of worker threads to 4 still yielded an overall speedup of 3.71.)

5 CONCLUSION AND FUTURE WORK

In this paper, we presented Draisine 200.0, a real-world cyberphysical system with a digital controller. In our case study, we have shown that while the interaction between the digital controller and the plant makes verification difficult, a carefully crafted verifier can yield feasible verification running times. As such, the Draisine 200.0 case presented here can be considered a valuable benchmark for the verification of real-world cyberphysical systems.

Acknowledgments

We would like to thank Gereon Fox and Florian Schießl for their work on the soft- and hardware of Draisine 200.0 and Michaela Klauck and Gilles Nies for their feedback on a draft of this paper.

REFERENCES

[1] Matthias Althoff. 2015. An Introduction to CORA 2015. In *1st and 2nd International Workshop on Applied veRification for Continuous and Hybrid Systems, ARCH@CPSWeek 2014, Berlin, Germany, April 14, 2014 / ARCH@CPSWeek 2015, Seattle, WA, USA, April 13, 2015 (EPiC Series in Computing, Vol. 34)*, Goran Frehse and Matthias Althoff (Eds.). EasyChair, 120–151. https://easychair.org/publications/paper/xMm

[2] Sergiy Bogomolov, Marcelo Forets, Goran Frehse, Kostiantyn Potomkin, and Christian Schilling. 2019. JuliaReach: a toolbox for set-based reachability. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control, HSCC 2019, Montreal, QC, Canada, April 16-18, 2019*, Necmiye Ozay and Pavithra Prabhakar (Eds.). ACM, 39–44. https://doi.org/10.1145/3302504.3311804

[3] European Committee for Standardization. 2017. European Standard EN 15194.

[4] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. 2011. SpaceEx: Scalable Verification of Hybrid Systems. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6806)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, 379–395. https://doi.org/10.1007/978-3-642-22110-1_30

[5] Thomas A. Henzinger. 1996. The Theory of Hybrid Automata. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996*. IEEE Computer Society, 278–292. https://doi.org/10.1109/LICS.1996.561342

[6] Colas Le Guernic and Antoine Girard. 2010. Reachability analysis of linear systems using support functions. *Nonlinear Analysis: Hybrid Systems* 4, 2 (2010), 250–262. https://doi.org/10.1016/j.nahs.2009.03.002 IFAC World Congress 2008.

[7] Magnus Lindahl, Paul Pettersson, and Wang Yi. 2001. Formal design and analysis of a gear controller. *International Journal on Software Tools for Technology Transfer* 3 (2001), 353–368. Issue 3. https://doi.org/10.1007/s100090100048

[8] Ibtissem Ben Makhlouf and Stefan Kowalewski. 2014. Networked Cooperative Platoon of Vehicles for Testing Methods and Verification Tools. In *1st and 2nd International Workshop on Applied veRification for Continuous and Hybrid Systems, ARCH@CPSWeek 2014, Berlin, Germany, April 14, 2014 / ARCH@CPSWeek 2015, Seattle, WA, USA, April 13, 2015 (EPiC Series in Computing, Vol. 34)*, Goran Frehse and Matthias Althoff (Eds.). EasyChair, 37–42. https://easychair.org/publications/paper/3QLs

[9] Hafiz Muhammad Yasir Naeem and A Mahmood. 2016. Autonomous cruise control of car using LQR and H2 control algorithm. In *2016 International Conference on Intelligent Systems Engineering (ICISE)*. 123–128. https://doi.org/10.1109/INTELSE.2016.7475173

[10] Stefan Schupp, Erika Ábrahám, Ibtissem Ben Makhlouf, and Stefan Kowalewski. 2017. HyPro: A C++ Library of State Set Representations for Hybrid Systems Reachability Analysis. In *NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10227)*, Clark W. Barrett, Misty Davies, and Temesghen Kahsai (Eds.). 288–294. https://doi.org/10.1007/978-3-319-57288-8_20