André Platzer

Lecture Notes on Foundations of Cyber-Physical Systems

15-424/624/824 Foundations of Cyber-Physical Systems

Chapter 18
Verified Models & Verified Runtime Validation

Synposis This chapter provides an important twist on cyber-physical systems verification. Without any doubt, formal verification provides crucial safety information for CPSs with exhaustive coverage of all the infinitely many possible behaviors that no finite amount of testing could ever provide. The catch is that the safety result then covers all behavior of the verified CPS models, but only provides safety guarantees for the actual CPS implementation to the extent that this implementation fits to the model. Obtaining good enough models of physics is a nontrivial challenge in and of itself. This chapter provides a systematic way how the safety guarantees about a CPS model transfer to safety results for the actual implementation with the help of provably correct runtime compliance monitors. When run on the CPS implementation, these runtime monitors validate the actual execution in a verified way against the verified models that were proved safe previously.

18.1 Introduction

Since cyber-physical systems provide so many interesting control challenges of subtle interactions with the uncertainties and complexities of the physical world, it is quite nontrivial to get them right. Due to the large number of possibilities how the behavior of the relevant systems can interact, full coverage is best achieved with the support of formal verification and validation techniques. In order to have the benefit of a full coverage of safety for all possible behaviors, it is, of course, necessary to provide a model of the system under scrutiny, including a model not just of its controllers but also the relevant part of physics.

Models of reality come with certain inevitable challenges. The world is a complicated place, which implies that our models of the world will either also be exceedingly complex or else focus on certain simpler fragments. The trick is to focus exclusively on the relevant aspects of reality and devise a model of the physical dynamics that makes use of simplifying abstractions, including nondeterministic overapproximations, as much as possible. Just recall how hybridness and nonde-

terminism helped simplify the bouncing ball model in Chap. 4 and Sect. 11.11 by giving it more behavior than realistically possible but in ways that make it easier to describe. But what is relevant and how do we make sure that the model covers reality?

When we are done with a proof of safety, we have the most exhaustive guarantee for the particular question about the particular model in the differential dynamic logic formula that we proved. While this proves that model to be safe, it only verifies the actual CPS implementation to the extent that this implementation fits to the model. How could we establish that it really does? More generally, how could we have the model's safety result transfer to the safety of the actual implementation?

As we have already seen in Part I, we could hardly squeeze an actual self-driving car into a logical modality in a formula and expect to prove any meaningful properties about this mix of a syntactic expression and a physical object. Besides, any such attempt would still be missing out on the physical model of relevance for the car's motion.

Instead, we will take a more subtle route and produce a monitor program to be run on the CPS implementation that will check all the time whether the present behavior fits to what the model in the safety proof assumed, which ensures that the safety result about the CPS model applies to the present reality. But that monitor will be accompanied by a proof that it performs this checking in provably correct ways such that a successful response from the monitor program implies that the specific behavior of the concrete implementation is safe.

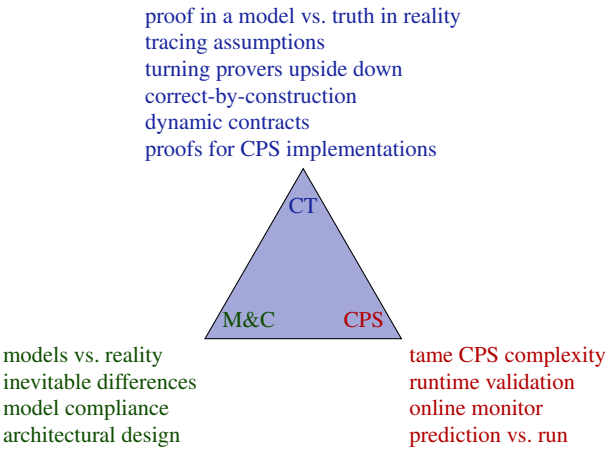
The most important learning goals of this chapter are:

Modeling and Control: The crucial lesson of this chapter is that there are inevitable differences of models compared to reality, because it is infeasible or even impossible to model all complexities of reality exactly. Fortunately, it also is not necessary to model all of reality for enabling predictions about a cyber-physical system affecting only a part of the world. But even then, there are nontrivial challenges in making sure that the models provide an adequate level of detail. This chapter investigates systematic ways of ensuring that the real system complies with the model, or, vice versa, the model fits to reality. Another side line will be a few insights about the impact that safety considerations have on architectural design, because clever system architectures simplify the safety argument by helping to reduce safety to a smaller subsystem.

Computational Thinking: Relationships between truth and proof are of fundamental significance in logic and are the backbone of soundness and completeness considerations. A unique twist in cyber-physical systems is the fundamental challenge that, despite all soundness in the proof calculus, there can still be discrepancies between proofs in a model and truth in reality. While a sound proof makes perfect guarantees about the system behavior in the model, these guarantees only apply to the real system *if* accurate models of the system can be obtained. For CPS, this includes the daunting task of finding models not only of the controllers but also of the physical dynamics. This problem is fundamental and cannot be overcome by shifting to more precise models. Seemingly, the problem could be sidestepped by working with CPS data and experiments, but

those have no predictive power without again assuming a corresponding model of reality, which leads to a vicious circle of assumptions either way.
This chapter gives a high-level account of a technique called *ModelPlex* [5], which provides a way of cutting through this Gordian knot of models and assumptions by combining offline proof with verifiably correct online monitoring. By turning a theorem prover upside down, ModelPlex generates provably correct monitor conditions that, if checked to hold at runtime, are provably guaranteed to imply that the offline safety verification results about the CPS model apply to the present run of the actual CPS implementation so that it is provably safe. This results in a correct-by-construction approach leveraging dynamic contracts to have proofs about models transfer to CPS implementations.

CPS Skills: This chapter provides ways of taming CPS complexity by making it possible to isolate safety-critical parts and by providing ways of working with simplified models without losing the connection to the real CPS. It introduces the important pragmatic concept of runtime validation with online monitors that check the behavior of the real CPS implementation at runtime. Their primary purpose is to check for deviations of predictions about the behavior of the system compared to the actual observed runs and stop the system safely if potentially dangerous deviations are detected.



18.2 Fundamental Challenges with Inevitable Models

In differential dynamic logic [6–8], models can be expressed directly as a hybrid program with its discrete controller actions and continuous differential equations that interact according to the program operators. After specifying the correctness

properties of interest in differential dynamic logic, this logic provides rigorous reasoning techniques for proving the correctness properties as we saw in Part I for elementary CPS and in Part II for advanced CPS with sophisticated continuous dynamics. Once we are done with such a proof, we have achieved a major advancement of our understanding of the system and have obtained a rigorous safety argument why the controllers keep the CPS safe according to the dL formula.

Indubitably, such a rigorous safety result provides a lot of confidence in the correct design of the system, especially since it is even accompanied by a proof as a safety certificate. The more nuanced subtlety, however, is that we need to make sure we have phrased the dL formula correctly. This formula contains the preconditions, controller, physical model, and postconditions. For example, consider a typical dL formula of the shape

$$A \rightarrow [(ctrl;plant)^*]B \tag{18.1}$$

Asking René Descartes for help with his skepticism, what could still have gone wrong even after we have a proof of (18.1)? What would happen if we wrote down the wrong postcondition B? If we ask the wrong question, we will get a perfect answer but for a question we were not interested in. It is, thus, of paramount importance that we review B carefully to make sure it really expresses all safety-critical properties of significance for the system.

What would happen if we use the wrong precondition A? Unlike in postcondition B, we cannot have forgotten a crucial condition in the precondition A, because the dL formula (18.1) would otherwise just not have a proof. What could happen, still, is that the precondition A on the initial state is overly conservative, so that we cannot turn the CPS on safely in as many circumstances as we would like. That is a pity but at least not unsafe. Except when A is never true, because it contains a contradiction, in which case (18.1) is vacuously true, because its assumption is impossible. It is, thus, helpful to prove satisfiability of all preconditions as a sanity check.

Note 77 (Impossible assumptions) *Whenever you make an assumption in your model or safety property, it is a good idea to check whether that assumption is possible. It is an even better idea to prove that it is at least satisfiable, so there is a state in which it is true.*

What else could have gone wrong in phrasing the safety conjecture (18.1)? We could have described the controller *ctrl* incorrectly. Or, rather, there could have been an important discrepancy between what our controller model *ctrl* says it does and what an actual code implementation or low-level microcontroller really runs. Differential refinement logic [3, 4], for example, is an extension of differential dynamic logic that provides a systematic approach for relating more abstract controllers with safety proofs to more concrete controllers with additional efficiency properties that inherit safety for free. As we saw in Part I, more abstract models are often easier to verify than models containing full detail. If we were to choose a very different implementation platform or, say, the additional semantic ambiguities of a low-level

C program, we still need some way of establishing a guaranteed link between what was verified in (18.1) and the code that is really running on the CPS in the end.

Finally, and most critically, we could have gotten the physical model *plant* wrong in (18.1). What happens then? Well, then our CPS is in trouble. If we accidentally wrote down the physical model of a train and prove the dL formula in (18.1), we cannot expect it to control a rocket satisfactorily, because their physical dynamics are so different on account of the dominant absence of rails in space. But even if we got the basic principles of the physical model right, yet missed some of its crucial aspects, then our proof of (18.1) will have limited predictive power for reality.

For the controller *ctrl*, the savior could be to move it closer to the implementation with increasingly fine-grained details, possibly absorbing the verification complexity shock with refinement proof techniques [4]. Maybe the same approach helps for the physical model *plant*? Well it would, and then again it would not. On the one hand, safety results about increasingly higher-fidelity models improve the range of behaviors where the safety results apply to reality. On the other hand, even higher-fidelity models are still just that: models of reality. Even a model with Schrödinger's equation of quantum mechanics [9] is still only a model of reality, and not even a very useful one for describing and predicting the motion of cars on the road, because quantum mechanics is more relevant for particles that are significantly smaller than cars. A model with Einstein's Field equations of general relativity [2] is also still a model of reality, and not any more useful for describing a car, because that is more relevant for fast objects closer to the speed of light, which is forbidden for cars on most highways.

Be that as it may, some models are pretty useful for making predictions about reality. This attitude [10] has been well-captured by George Box's [1] slogan:

Note 78 (George Box) *All models are wrong but some are useful.*

So, we will still continue to use models, because they enable predictions, but will from now on be more aware of the fact that there are certain tradeoffs of analyzability and accuracy.

Is there a way to sidestep the issue by just not using any models in the first place? In fact, what could we even do without a model? We could run experiments and gather sample data about the behavior of a system. While that is certainly quite useful, too, it is important to understand that we will still need models to enable any predictions at all. Generality comes from the use of models. Unless we fix a model how the behavior at the setup of the experiment relates to the behavior in other circumstances, the data alone will not provide any predictive power. Is the altitude of where your car is running relevant for its operation? Probably not. Yet, does the slope of the road affect its behavior? Quite likely. Unless we at least make such dependence and independence assumptions in a model, there is no way that we could ever conjecture with any amount of certainty that our car will need at least 16m to brake to a standstill from 35 mp/h (or roughly 50 km/h) next time, too, no matter where else we have already tried.

So it looks like, for better or for worse, we are stuck with the use of models at least for the physics. Is there anything at all that we can do to make sure our guarantees about the CPS models transfer to the actual CPS implementations? That is what this chapter investigates.

18.3 Runtime Monitors

To begin with, assume that we have taken the lessons from Parts I and II to good use by having come up with a proof of a dL formula, e.g., of the shape:

$$A \rightarrow [(ctrl;plant)^*]B \tag{18.1*}$$

Now all that remains is to figure out a way to make sure that this safety result about the CPS model $(ctrl;plant)^*$ transfers to the actual CPS implementation. If offline results alone do not check whether all parts of (18.1) fit to the real CPS, then let us investigate runtime monitors that address this question online when the CPS runs.

Checking that the initial condition A applies for the real CPS is straightforward as long as all its quantities can be measured. In that case, all it takes is to evaluate at runtime whether the formula A is true in the initial state and only permit turning the CPS on if it is.

Monitoring the postcondition B to see whether it is true would be equally straightforward. But that is not actually useful, because if B is ever false, then the system is already hopelessly unsafe and there is nothing that can be done about it anymore, since we cannot just go back in time and do things differently. Just think of a postcondition B expressing that the controlled car should have positive distance to other cars. Once that condition evaluates to *false*, the cars collided. Well, maybe matters get better if we work with a postcondition B that has an extra margin such as a distance of at least 1m, say? That does not really help either, because once that safety margin is violated, the car might already be going so fast, that a collision is unavoidable regardless.

So, the most challenging aspect in runtime monitoring is to find out what precise condition should be monitored, and to identify what exactly one knows about the system behavior if that monitor condition checks successfully. Proofs play a crucial role in finding out in identifying what needs to be monitored. And they are certainly fundamental in proving what correctness properties the resulting monitors come with, meaning what one knows about the CPS if the monitors all evaluate to *true*.

Continuing down the list, how can we monitor the controller *ctrl* in (18.1) to see if the real CPS fits to it? For various reasons, the controller implementation in the CPS may have slight discrepancies compared to its higher-level control model *ctrl*. For example, the controller could have been implemented in a low-level language like C, or could use preexisting legacy code, or could have been synthesized from a Stateflow/Simulink model, or it might be running a low-level microcontroller machine code. Also, for more fundamental reasons, recall that the controller model *ctrl*

includes a model for the discrete actions of agents in the environment, such as the nondeterministic choice of acceleration or braking for the car in front, where we may not have access to the actual implementation. All these factors contribute to potential deviations of the actual controllers compared to the controller model *ctrl*. What could we monitor to check whether the real CPS fits to the model *ctrl*?

Before you read on, see if you can find the answer for yourself.

The most important impact of the controller *ctrl* is to decide how to set control variables for the physical dynamics *plant* after suitable computations based on certain measurements of sensor inputs. All these final control variable decisions in the the real controller (let's call it γ_{ctrl}) can be monitored and checked for compatibility with what the controller model *ctrl* would have permitted. Of course, due to nondeterminism, the real controller implementation γ_{ctrl} can reach different decisions than the model *ctrl*, but it should only ever reach decisions that the verified controller model *ctrl* would at least have allowed. If, in any circumstance, the real controller implementation γ_{ctrl} ever decides to assign values to control variables that the verified model *ctrl* would not have allowed, then these are potentially unsafe and should be rejected for safety reasons.

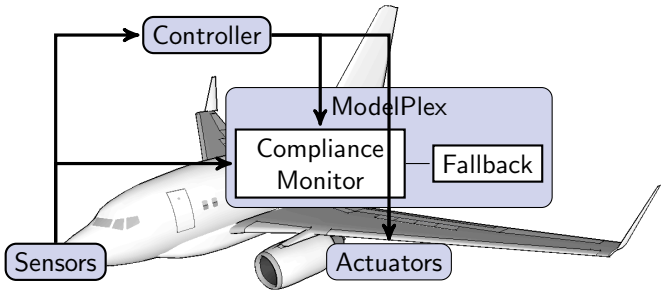


Fig. 18.1 ModelPlex monitors sit in between controller and actuator to check controller's decisions for compliance with the model based on sensor data with veto leading to a safe fallback action

Such a *controller monitor* would inspect each and every resulting decision by the controller implementation for compliance with the verified controller model *ctrl* based on the current sensor data and veto the decision if *ctrl* would not allow it; see Fig. 18.1. Of course, the resulting controller monitor cannot just reject a control decision, but must also override it with a safe fallback action to execute on *plant* instead. Figuring out such a safe fallback action is not always obvious either. But at least it is an easier problem, because that safe fallback action just needs to keep the system in safe stasis without doing anything particularly useful. In a car, for example, this last resort action could consist in applying emergency brakes and asking the human to investigate. In an aircraft, it could be flying in a loitering circle until the problematic situation resolves.

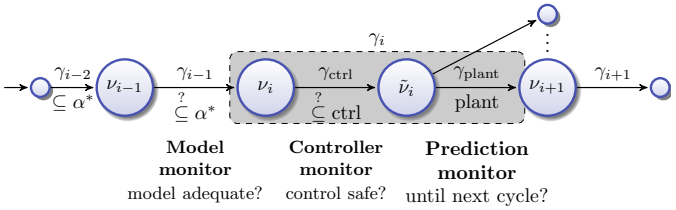


Fig. 18.2 Use of ModelPlex monitors along a system run

Let us assume that we have already found and proved safe such a safe fallback action, after all that is an easier problem. That leaves open the question how we could best monitor and determine whether an observed controller action of the real controller implementation γ_{ctrl} fits to the verified model *ctrl*. While you are invited to think about this challenge already, we will postpone it and first consider another challenge.

Proceeding further, how can we monitor the physical model *plant* in (18.1) to see if the real CPS physics fits to it? That is even more subtle, because, no matter what we try, the real physical world does not come with any source code that we could run or read to try to find out whether it fits to the model *plant*. Instead, the only chance is to try out the physical system and observe what it does to see if it fits to the model *plant*. Interestingly, that is already quite well-aligned with the approach we settled on for the real controller γ_{ctrl}, just for completely different reasons.

Somewhat similar to the controller monitor, which models just the responsibilities of the controller for compliance with *ctrl*, the *model monitor* models the whole system for compliance with the model *ctrl*;*plant* that includes the physical model.[1] The model monitor will inspect the data from the current state ν_i and the data from the previous state ν_{i-1} when it ran last to check whether the transition from ν_{i-1} to ν_i can be explained by the model *ctrl*;*plant*. If that transition fits to the model *ctrl*;*plant*, then since all repetitions of said model from a safe initial state satisfying *A* (which we checked at runtime initially) are safe (satisfying *B* by the offline proof), then the concrete run of the real CPS implementation ending in ν_i is also safe. If the transition ν_{i-1} to ν_i does not fit to *ctrl*;*plant*, however, then the safety proof of (18.1) does not apply to the current execution, so the model monitor vetoes and initiates a safe fallback action.

18.4 Model Compliance

Based on these general runtime monitoring principles, the primary remaining question is how to check a concrete system execution for compliance with the verified

[1] The reason for monitoring the whole control loop body *ctrl*;*plant* instead of just separately the physics *plant* is that this provides better guarantees [5] and also works if the HP is of any arbitrary form other than *ctrl*;*plant**.

model $(ctrl;plant)^*$ from (18.1). Of course, it is comparably easy to check whether the initial state satisfies the precondition A, at least when all its relevant quantities can be measured appropriately. But the same is not true for the hybrid program $(ctrl;plant)^*$, because of the nondeterminism and differential equations that it involves.

Example 18.1 (Bouncing ball monitors). As a simple guiding example, recall the familiar acrophobic bouncing ball Quantum that has been with us ever since Chap. 4:

$$0 \leq x \land x = H \land v = 0 \land g > 0 \land 1 \geq c \geq 0 \rightarrow \\ \left[\left(\{x' = v, v' = -g \,\&\, x \geq 0\};\ (?x = 0; v := -cv \cup ?x \neq 0)\right)^*\right](0 \leq x \land x \leq H) \tag{4.24}$$

This formula has been proved in Proposition 7.1 with the additional assumption $c = 1$ that Exercise 7.5 showed can be removed again. That led to a perfect proof of safety for Quantum, if only Quantum actually fits to the hybrid model in the formula (4.24) that was proved in dL's sequent calculus.

You might have noticed in earlier chapters that Quantum is easily startled. Even before reading Expedition 4.4 on p. 109, Quantum was already a natural born Cartesian skepticist. His level of scrutiny and skeptical doubt only increased after reading this chapter. Since Quantum really wants to get things right, he checks the initial condition of (4.24) and then figures out how to check whether the real execution fits to the hybrid program model in (4.24).

$$\left(x = 0 \land v^+ = -cv \lor x > 0 \land v^+ = v\right) \land x^+ = x \tag{18.2}$$

This represents a controller monitor if no physical motion ever happens. For just the differential equations $x' = v, v' = -g$, a corresponding monitor read off from the loop invariants (7.6) of the system is:

$$2g(x^+ - x) = v^2 - (v^+)^2$$

In fact, this monitor condition can also easily be read off from the solutions of the differential equation by eliminating time t, which is unobservable in (4.24):

$$\begin{aligned} v^+ = v - gt \qquad & \overset{g\neq 0}{\equiv} t = \frac{v - v^+}{g} \\ x^+ = x + vt - \frac{g}{2}t^2 \qquad & \overset{\text{above}}{\equiv} x^+ = x + v\frac{v - v^+}{g} - \frac{g}{2}\frac{v^2 - 2vv^+ - (v^+)^2}{g^2} \\ & \equiv\ 2g(x^+ - x) = 2v^2 - 2vv^+ - v^2 + 2vv^+ - (v^+)^2 \end{aligned}$$

This invariant is always true during the differential equation but it neglects the fact that the differential equation must have evolved forward ($t \geq 0$), which, since $g > 0$ is easily expressed by an additional conjunct:

$$2g(x^+ - x) = v^2 - (v^+)^2 \land v^+ \leq v \tag{18.3}$$

Let us call the hybrid program of interest α.

18.5 Provably Correct Monitor Synthesis

18.5.1 Logical State Relations

18.5.2 Model Monitors

18.5.3 Correct-by-Construction Synthesis

18.6 Summary

Exercises

18.1 (Safe bouncing ball monitors). Prove correctness of the monitors for the bouncing ball in dL's sequent calculus.

18.2 (Ping pong monitors). Create the controller monitor and model monitor for the verified ping pong models of the event-triggered design in Chap. 8 and of the time-triggered design in Chap. 9. Convince yourself that they are correct, i.e. could have led to a corresponding dL proof. Discuss the pragmatic different of the monitors resulting from the event-triggered models compared to the time-triggered designs. Is there a discrepancy that one monitor discovers that the other one does not?