Chapter 19
Verified Models & Verified Runtime Validation

Synopsis This chapter provides an important twist on cyber-physical systems analysis. 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 the model. Obtaining good enough models of physics is a nontrivial challenge in and of itself. This chapter provides a systematic way to transfer the safety guarantees about a CPS model 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.

19.1 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 ways in which 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 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-

A. Platzer, *Logical Foundations of Cyber-Physical Systems*,
https://doi.org/10.1007/978-3-319-63588-0_19

terminism helped simplify the bouncing-ball model in Chap. 4 and Sect. 11.12 by giving it more behavior than realistically possible but in ways that make it easier to describe. 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 the model. How can we establish that it really does? More generally, how can we have the model's safety result transfer to the safety of the actual implementation?

As we have already seen in Part I, we can 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 and the behavior of its environment.

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 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.

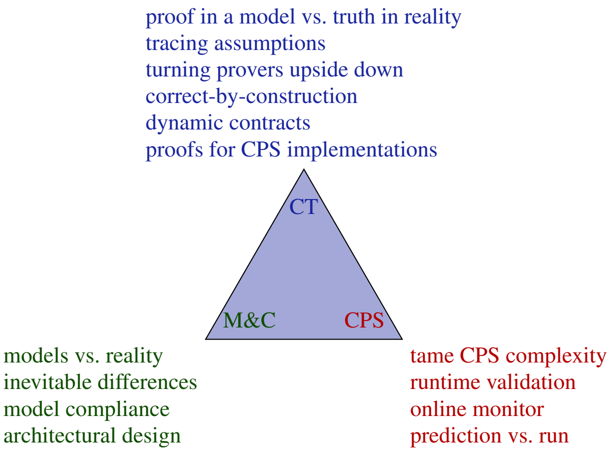
While a substantially more detailed technique can be found in previous work [5], this chapter emphasizes a simple intuitive approach to overcoming the challenges of model mismatches. The most important learning goals of this chapter are:

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 to make 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 reality. Another sideline 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 might 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 transfer proofs about models 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 disciplined 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 whenever potentially dangerous deviations are detected.



19.2 Fundamental Challenges with Inevitable Models

In differential dynamic logic [7–9], 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 an undeniable 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 the sake of illustration, consider a typical dL formula of this shape:

$$A \to [(ctrl;plant)^*]B \tag{19.1}$$

Asking René Descartes for help with his skepticism, what could still go wrong even after we have a proof of (19.1)? 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 are not interested in. It is, thus, of paramount importance that we review postcondition B carefully to make sure it really expresses all safety-critical properties of significance for the system.

What would happen if we used the wrong precondition A? Unlike in postcondition B, we cannot have forgotten a crucial condition in the precondition A, because the dL formula (19.1) would otherwise just not have a proof. What might 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 (19.1) is vacuously true, because its assumption is impossible. It is, thus, helpful to prove satisfiability of all preconditions as a sanity check.

Note 80 (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 (19.1)? 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 software implementation or low-level microcontroller really does. Differential refinement logic [3, 4] 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 would still need some way of establishing a guaranteed link between what was verified in (19.1) 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 (19.1). What would happen then? Well, then our CPS would be in trouble. If we accidentally wrote down the physical model of a train and prove the dL formula in (19.1), we cannot expect it to control a rocket satisfactorily, because their physical dynamics are so different on account of the noticeable 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 (19.1) would have limited predictive power for reality.

For the controller *ctrl*, the savior might 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 would help 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 [10] 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 close 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 [6] has been well-captured by George Box's [1] slogan.

Note 81 (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 models come with 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 can we even do without a model? We can 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 of 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 drives relevant for its operation? Probably not. Yet, does the slope of the road affect its behavior? Quite likely. Does the weather have any influence? Unless we at least make some such dependence and independence assumptions in a model, there is no way that we can ever conjecture with any amount of certainty that our car will need at least 16 m to brake to a standstill from 35 mph (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.

19.3 Runtime Monitors

To begin with, assume that we have put 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 illustrative shape:

$$A \rightarrow [(ctrl;plant)^*]B \qquad (19.1^*)$$

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 (19.1) fit 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 the CPS to be turned on if it is. That's still relatively straightforward (if every relevant quantity is physically measurable).

Monitoring the postcondition B to see whether it is true would be equally straightforward. But that is not actually useful at all, because if B is ever false, then the system is already hopelessly unsafe by definition 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 have collided and all hope for a happy ending is lost. Well, maybe matters get better if we work with a postcondition B that has an extra margin such as a distance of 1 m at least? 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 is checked successfully. Proofs play a crucial rôle 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 (19.1) to see whether the real CPS fits 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 might have been implemented in a low-level language such as C, or might use preexisting legacy code, or might have been synthesized from a Stateflow/Simulink model, or it might be running low-level microcontroller machine code. More fundamentally, recall that the controller model *ctrl* includes a model of 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 can 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 real controller (let's call it γ_{ctrl}) should be monitored and checked for compatibility with what the controller model *ctrl* permits. 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* at least allows. If, in any circumstance, the real controller implementation γ_{ctrl} ever decides to assign values to control variables that the verified model *ctrl* does not allow, then these are potentially unsafe and should be rejected for safety reasons.

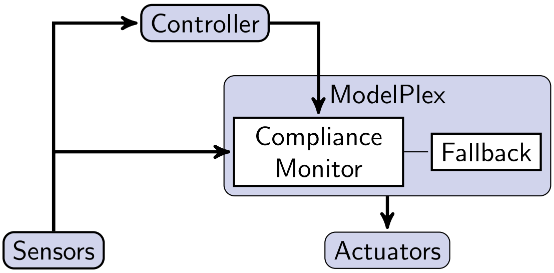


Fig. 19.1 ModelPlex monitors sit between controller and actuator to check the controller's decisions for compliance with the model based on sensor data with veto leading to a safe fallback action

Such a *controller monitor* inspects each and every resulting decision by the controller implementation for compliance with the verified controller model *ctrl* based on the current sensor data and vetoes the decision if *ctrl* does not allow it; see Fig. 19.1. 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 might consist of applying emergency brakes, cutting the engine's power, and asking the human to investigate. In an aircraft, it might be flying in a loitering circle until the problematic situation is resolved. In a quadrotor drone, it might be hovering in place.

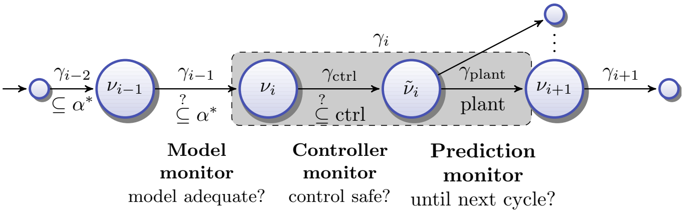


Fig. 19.2 Use of ModelPlex monitors along a system run

Let us assume that we have already found and proved safe such a safe fallback action, since that is an easier problem. That leaves open the question of how we can best monitor and determine whether an observed controller action of the real controller implementation γ_{ctrl} fits 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 (19.1) to see whether the real CPS physics fits 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 can run or read to try to find out whether it fits the model *plant*. Instead, the only chance is to try out the physical system and observe what it does to see whether it fits 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 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 *ctrl*;*plant*, however, then the safety proof of the CPS model (19.1) does not apply to the current execution, so the model monitor vetoes and initiates a safe fallback action.

[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 $(ctrl;plant)^*$.

19.4 Model Compliance

Based on these general runtime monitoring principles, the primary remaining question is how to actually check a concrete system execution for compliance with the verified model $(ctrl;plant)^*$ from (19.1). Of course, it is relatively easy to check whether the initial state satisfies the precondition A, or at least it is easy when all its relevant quantities can be measured appropriately. But the same is not true for the hybrid program $(ctrl;plant)^*$, because of all the nondeterminism and differential equations that it involves. It, thus, takes a more clever approach to determine whether the real system execution fits the hybrid program $(ctrl;plant)^*$. This section motivates and intuitively develops such an approach with the simple example of a bouncing ball.

Example 19.1 (Bouncing-ball monitors). As a simple guiding example, recall the familiar acrophobic bouncing ball Quantum that has been with us ever since Chap. 4:

$$0 \le x \land x = H \land v = 0 \land g > 0 \land 1 \ge c \ge 0 \rightarrow \\ \big[\big(\{x' = v, v' = -g \,\&\, x \ge 0\};\ (?x = 0; v := -cv \cup ?x \ne 0)\big)^*\big]\,(0 \le x \land x \le H) \tag{4.24}$$

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

You might already have noticed in earlier chapters that Quantum is easily startled. Even before reading Expedition 4.5 on p. 123, Quantum was already a natural born Cartesian skepticist. His level of scrutiny and skeptical doubt only increased after reading Chap. 19. Since Quantum really wants to get things right, he checks the initial condition of formula (4.24) and then figures out how to check whether the real execution fits the hybrid program model in (4.24).

Figuring that the differential equations in (4.24) are surely the best possible differential equations that could ever exist (since they are meant to describe bouncing balls, after all), Quantum first only worries about the discrete controller part of (4.24). Is there a logical formula characterizing that a controller implementation switching from position x and velocity v to new position x^+ and velocity v^+ fits the discrete controller in (4.24)?

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

A run of an actual discrete controller implementation changing the position from x to x^+ and the velocity from v to v^+ (leaving all other variables alone) only faithfully fits the controller in (4.24) if the following logical formula evaluates to *true*:

$$\big(x = 0 \land v^+ = -cv \ \lor\ x > 0 \land v^+ = v\big) \land x^+ = x \tag{19.2}$$

This formula represents a controller monitor if no physical motion ever happens. Certainly, the conjunct $x^+ = x$ needs to be *true* for an execution to faithfully run the discrete controller of (4.24), because the discrete dynamics of the bouncing ball does not affect the ball's altitude but merely its velocity. Furthermore, it is either the case that the ball runs the first control branch so is presently on the ground ($x = 0$) and the new velocity v^+ after the discrete ground controller is the damped version $-cv$ of the previous velocity v, or it is the case that the ball is still in the air ($x > 0$ because of the second control branch $?x \neq 0$ together with the evolution domain constraint $x \geq 0$) and then the velocity is unaltered ($v^+ = v$). Every run satisfying controller monitor (19.2) fits an execution of the discrete controller (4.24).

In retrospect, it is reasonably obvious how controller monitor (19.2) relates to the controller in HP (4.24), with a disjunction corresponding to the controller's choices and conjunctions accumulating the conditions of tests and effects of assignments. Of course, the particular discrete controller in (4.24) is deterministic and, hence, so is the monitor (19.2). But the same principle applies for controllers with ample nondeterminism, in which case the resulting monitor condition is more flexible.

To validate the controller model, Quantum takes out his favorite oscilloscope and plenty of other measuring devices and quickly evaluates controller monitor (4.24) for a number of trial bounces in a high-fidelity simulation environment (Fig. 19.3).

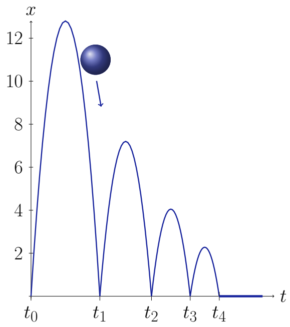


Fig. 19.3 Sample run of a bouncing ball (plotted as height over time) that ultimately lies down flat

The controller implementation holds up pretty well with respect to controller monitor (19.2) for quite a while, except that the ball ultimately dares to just lie flat on the ground. If the implemented controller turns $v = -3$ to $v^+ = 0$, then this is clearly a violation of (19.2) and flagged as such at time t_4. Quantum was initially tempted to dismiss t_4 as a measurement error but decided after retrying a few times that there must be an actual controller model mismatch. Indeed, looking back at (4.24), its HP does not yet include the refinements from Sect. 4.2.3 that enable bouncing balls to ultimately deflate and lie flat when their energy is insufficient to jump back up.

The controller implementation is not the only aspect of reality with the potential for a mismatch between model and reality. In fact, much more challenging than the controller's implementation is the implementation of physics that is also known as the real world. Despite dozens of centuries of progress, mankind is still sometimes at a loss when it comes to explaining physical phenomena, let alone the behavior of other agents acting in the environment. Consequently, there is even more that might go wrong if we start the analysis with a completely inadequate physical model. At the same time, even the best physical model is still just that: a model.

Example 19.2 (Bouncing-ball model monitors). When Quantum tries to devise a clever monitor to check the continuous physics behind HP (4.24), he discovers even more challenges than with the controllers. It is quite nontrivial to run a differential equation for an unspecified nondeterministic amount of time and check whether the resulting state coincides with the state that has been measured in a run of the real system implementation experimentally. Granted, the particular double integrator ODE in (4.24) is still of the relatively tame kind, but, for the most part, this only simplifies the illustration, not the question.

Is there a logical formula characterizing that a physical motion evolving from position x and velocity v to new position x^+ and velocity v^+ can be explained by the differential equation in (4.24)?

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

For just the differential equations $x' = v, v' = -g$, a corresponding monitor can be read off from the invariants (7.10) that we used to prove the bouncing ball:

$$2g(x^+ - x) = v^2 - (v^+)^2 \tag{19.3}$$

The change $v^2 - (v^+)^2$ in the squares of the velocities equals $-2g$ times the change $x - x^+$ in positions. This monitor condition can also easily be read off quite directly from the solutions of the differential equation (expressing the new position x^+ and velocity v^+ as a function of the old) by eliminating time t, which is unobservable in the HP of (4.24):

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

Indeed, the resulting equation (19.3) is the crucial invariant of the dynamics, but it does not characterize the system behavior sufficiently, because equation (19.3) also holds for physically impossible situations such as when the new velocity v^+ is chosen to be bigger than the previous velocity v, which is impossible under gravity $v' = -g$. The reason is that (19.3) is an invariant equation that is always true during the differential equation but neglects the fact that the differential equation must evolve forward. So (19.3) holds whether the differential equation evolves forward or backward in time, because it is unaware of the directionality in the system. Since $g > 0$, this directionality is easily expressed by an additional conjunct saying that the velocity never increases:

$$2g(x^+ - x) = v^2 - (v^+)^2 \land v^+ \leq v \tag{19.4}$$

The only challenge remaining to adequately represent all assumptions of the plant model in the bouncing ball is to add the domain constraint for the initial $x \geq 0$ and final position $x^+ \geq 0$:

$$2g(x^+ - x) = v^2 - (v^+)^2 \land v^+ \leq v \land x \geq 0 \land x^+ \geq 0 \tag{19.5}$$

Quantum can now validate trial runs of the physical motion in gravity for compatibility with the plant model by testing whether monitor (19.5) evaluates to *true*. In particular, if the observed change in the ball's position and change in velocity while falling according to gravity are always compatible with (19.5), then this supports the hypothesis that the differential equation and evolution domain constraints in HP (4.24) describe the reality of falling balls well.[2] The combination into an overall model monitor respecting the discrete controller and continuous motion is natural [5], essentially by substituting the plant monitor (19.5) into the controller monitor (19.2).

19.5 Provably Correct Monitor Synthesis

Reading off the appropriate monitor conditions from the models by an educated guess as in Sect. 19.4 is one thing. But justifying their correctness rigorously and making sure that no subtle but critical conditions are left out is another. While the relationship of controller monitor (19.2) to the discrete control model in (4.24) is reasonably transparent after suitable transformations of the evolution domains and tests, it is non-obvious whether (19.5) correctly checks *all* correctness-critical conditions of the plant model.

Since correctness in CPS is so important, it is also important to get the correctness monitors themselves correct. After all, the controller and model monitors are supposed to be last-resort mechanisms that, when things do not go according to plan, avert potential catastrophes by interfering before it is too late. It would not help if the monitor conditions monitored the wrong expressions and ended up missing critical safety hazards that they were meant to spot.

The question is: what makes a runtime monitor condition correct? How do we tell a correct runtime monitor apart from a well-intended but flawed monitor? These questions are not unlike the ones we asked out for cyber-physical systems themselves in Chap. 4. But the big difference is that it is a more narrow problem to ask whether a runtime monitor adequately represents the conditions of a model, because we already have both the runtime monitor as well as the model it is supposed to check compatibility with. Contrast this with Chap. 4 where we were given a model and were still trying to identify its appropriate safety condition. So when is a runtime monitor correct?

[2] Of course, falling balls cannot deny a certain resemblance to the apples that were falling from trees in Isaac Newton's time. So one model might describe two related scenarios.

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

The specification that correct runtime monitors need to obey is that they imply that there, indeed, is a run of the model that explains the observed transition of the previous state to the present state. If the monitor condition evaluates to *false*, then an alarm about a possible model violation is raised and a safe fallback action is initiated (such as applying emergency brakes and cutting power). But if it evaluates to *true*, then the monitor had better be right about the fact that the observed behavior fits to the model. In other words, if the monitor condition is *true* for the old position x and new position x^+, then there really needs to be a run of the corresponding model that, indeed, leads from position x to x^+ as observed (and accordingly for previous velocity v to new velocity v^+ and other variables).

Solve Exercise 19.1 now to demonstrate that the runtime monitors are correct.

19.5.1 Logical State Relations

Let us call the hybrid program of the relevant model α and let $\chi(x,x^+)$ be a runtime monitor formula. The runtime monitor formula $\chi(x,x^+)$ is a formula in which the variable (vector) x is meant to refer to the previous position (and velocity or other relevant state variables) while variable (vector) x^+ is meant to refer to the new position (and velocity).

Definition 19.1 (Correctness of runtime monitors). The *runtime monitor* formula $\chi(x,x^+)$ is called *correct* for the hybrid program model α with bound variables $\mathrm{BV}(\alpha) \subseteq \{x\}$ iff the following dL formula is valid:

$$\chi(x,x^+) \rightarrow \langle\alpha\rangle x = x^+$$

If x is a vector of variables $(x_1,\ldots,x_n)$ and x^+, thus, is the vector of variables $(x_1^+,\ldots,x_n^+)$, then the vectorial equation $x = x^+$ means the same as the conjunction

$$\bigwedge_{i=1}^{n} x_i = x_i^+$$

Example 19.3 (Correctness of controller monitor). Continuing Example 19.1, let $\chi(x,v,x^+,v^+)$ denote the controller monitor formula (19.2) for the discrete controller of (4.24).

$$\chi(x,v,x^+,v^+) \overset{\text{def}}{\equiv} \big(x = 0 \land v^+ = -cv \lor x > 0 \land v^+ = v\big) \land x^+ = x \qquad (19.2^*)$$

The correctness of controller monitor (19.2) can be proved in the dL calculus:

$$
\begin{array}{rl}
 & * \\ \cline{2-2}
 & \chi(x,v,x^+,v^+) \vdash (x=0 \to x{=}x^+ \land -cv{=}v^+) \lor (x \neq 0 \to x{=}x^+ \land v{=}v^+) \\ \cline{2-2}
\langle ? \rangle & \chi(x,v,x^+,v^+) \vdash \langle ?x=0 \rangle (x{=}x^+ \land -cv{=}v^+) \lor \langle ?x \neq 0 \rangle (x{=}x^+ \land v{=}v^+) \\ \cline{2-2}
\langle := \rangle & \chi(x,v,x^+,v^+) \vdash \langle ?x=0 \rangle \langle v:=-cv \rangle (x{=}x^+ \land v{=}v^+) \lor \langle ?x \neq 0 \rangle (x{=}x^+ \land v{=}v^+) \\ \cline{2-2}
\langle ; \rangle & \chi(x,v,x^+,v^+) \vdash \langle ?x=0; v:=-cv \rangle (x{=}x^+ \land v{=}v^+) \lor \langle ?x \neq 0 \rangle (x{=}x^+ \land v{=}v^+) \\ \cline{2-2}
\langle \cup \rangle & \chi(x,v,x^+,v^+) \vdash \langle ?x=0; v:=-cv \cup ?x \neq 0 \rangle (x{=}x^+ \land v{=}v^+)
\end{array}
$$

While this simple notion of correctness of runtime monitors is particularly easy to understand, there are also improved ways of establishing the correctness of controller monitors that additionally guarantee that the controller will never violate the evolution domain constraints of the plant [5].

Example 19.4 (Correctness of plant monitor). Continuing Example 19.2, this time, let $\chi(x,v,x^+,v^+)$ denote the differential equation monitor formula (19.4) for the differential equation of (4.24) without its evolution domain constraint. Because $g > 0$ holds, this monitor $\chi(x,v,x^+,v^+)$ is correct for the differential equation:

$$
\begin{array}{rl}
 & * \\ \cline{2-2}
\mathbb{R} & g>0, \chi(x,v,x^+,v^+) \vdash \exists t {\geq} 0 \, (-\tfrac{g}{2}t^2 + vt + x = x^+ \land v - gt = v^+) \\ \cline{2-2}
\langle := \rangle & g>0, \chi(x,v,x^+,v^+) \vdash \exists t {\geq} 0 \, \langle x := -\tfrac{g}{2}t^2 + vt + x \rangle \langle v := v - gt \rangle (x{=}x^+ \land v{=}v^+) \\ \cline{2-2}
\langle ' \rangle & g>0, \chi(x,v,x^+,v^+) \vdash \langle x' = v, v' = -g \rangle (x{=}x^+ \land v{=}v^+)
\end{array}
$$

In fact, it can even be shown that the differential equation monitor condition (19.4) is perfect, because it is *true* if and only if the differential equation can reach position x^+ and velocity v^+:

$$
g>0 \to \big(\langle x' = v, v' = -g \rangle (x = x^+ \land v = v^+) \leftrightarrow 2g(x^+ - x) = v^2 - (v^+)^2 \land v^+ \leq v \big)
$$

Because this equivalence is provable for $g > 0$, the differential equation monitor (19.4) will never raise false alarms. In a similar way (19.5) is a provably correct runtime monitor for the plant of (4.24) including the evolution domain constraint. That is, the following dL formula is provable:

$$
\begin{aligned}
g>0 \land 2g(x^+ - x) = v^2 - (v^+)^2 \land v^+ \leq v \land x \geq 0 \land x^+ \geq 0 \to \\
\langle x' = v, v' = -g \,\&\, x \geq 0 \rangle (x = x^+ \land v = v^+)
\end{aligned}
$$

Again, we can prove that this plant monitor is perfect because, given $g > 0$, it is equivalent to the reachability of x^+ and v^+ along the continuous plant:

$$
\begin{aligned}
g>0 \to \big(& 2g(x^+ - x) = v^2 - (v^+)^2 \land v^+ \leq v \land x \geq 0 \land x^+ \geq 0 \\
& \leftrightarrow \langle x' = v, v' = -g \,\&\, x \geq 0 \rangle (x = x^+ \land v = v^+) \big)
\end{aligned}
$$

19.5.2 Model Monitors

The above runtime monitors (19.2) for the discrete controller, (19.4) for the differential equation and (19.5) for the full plant are all useful, but do not cover all details of the HP model (4.24) of the bouncing ball yet. We can, nevertheless, follow essentially the same approach again for a runtime monitor of the full HP model. The only tricky part is the need to deal with the loop. The most natural way of monitoring the correct execution of a control loop, however, is to separately check each round of the control loop. Consequently, all we need to do is unwind the loop once with the iteration axiom $\langle ^* \rangle$ and just find a runtime monitor for the loop body α instead of for the full loop α^*. *Then this runtime monitor can be used for each round of the control loop in the controller implementation.* Such a runtime monitor for just the loop body is, in fact, also much more useful than a runtime monitor for the full loop α^*, because it can take quite a while before we finally know whether the entire loop execution fits the model. We would much prefer to already check during each run of the loop body whether everything still operates according to the model.

Example 19.5 (Correctness of model monitor). A runtime monitor for the HP's loop body can be constructed by substituting the respective monitors into one another:

$$\begin{aligned} &x^+ > 0 \land 2g(x^+ - x) = v^2 - (v^+)^2 \land v^+ \leq v \land x \geq 0 \\ \lor\, &x^+ = 0 \land c^2 2g(x^+ - x) = c^2 v^2 - (v^+)^2 \land v^+ \geq -cv \land x \geq 0 \end{aligned} \tag{19.6}$$

Correctness of the model monitor (19.6) for one iteration of the full loop body of the HP model (4.24) can be proved in dL.

Proposition 19.1 (Correct bouncing-ball model monitor). *Formula* (19.6) *is a correct model monitor for* (4.24). *That is this* dL *formula is valid:*

$$\begin{aligned} &g > 0 \land 1 \geq c \geq 0 \rightarrow \\ &\quad \big(x^+ > 0 \land 2g(x^+ - x) = v^2 - (v^+)^2 \land v^+ \leq v \land x \geq 0 \\ &\quad \lor x^+ = 0 \land c^2 2g(x^+ - x) = c^2 v^2 - (v^+)^2 \land v^+ \geq -cv \land x \geq 0 \\ &\quad \rightarrow \langle \{x' = v, v' = -g \,\&\, x \geq 0\}; (?x = 0; v := -cv \cup ?x \neq 0) \rangle (x = x^+ \land v = v^+)\big) \end{aligned}$$

In fact, equivalence instead of implication can again be proved as well, which demonstrates that the model monitor is exact and does not produce any false alarms.

19.5.3 Correct-by-Construction Synthesis

So far, we have settled for educated guesses to produce runtime monitor formulas and subsequently proved them correct in the dL proof calculus. That is perfectly acceptable except that we then always have to get creative to produce the monitor formulas in the first place.

It would be better if we could construct the runtime monitor formulas systematically. In fact, it would be even more helpful if we had a way of constructing the runtime monitor formulas in a correct-by-construction approach such that we simultaneously generate the runtime monitor and a proof of its correctness. Surprisingly, this is perfectly possible as well. All we need to do is to exploit the rigorous reasoning principles of differential dynamic logic for a purpose other than safety verification. The starting point is one crucial observation. What is the easiest formula satisfying correctness criterion Definition 19.1?

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

By far the easiest and most obviously correct formula satisfying runtime monitoring correctness (Definition 19.1) for model α is the dL formula $\langle\alpha\rangle\, x = x^+$ itself. Of course, when choosing $\chi(x,x^+) \overset{\text{def}}{\equiv} \langle\alpha\rangle\, x = x^+$, then the correctness condition is trivially valid because every formula implies itself, even $\langle\alpha\rangle\, x = x^+$ does:

$$\chi(x,x^+) \rightarrow \langle\alpha\rangle\, x = x^+$$

True beyond any doubt. What could be wrong with that?

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

Well, nobody can argue against the validity of $\langle\alpha\rangle\, x = x^+ \rightarrow \langle\alpha\rangle\, x = x^+$. But we hardly learn anything at all about what constitutes a faithful execution of the controller model α if this is how we choose $\chi(x,x^+)$. Indeed, the HP α itself is a pretty perfect model of what it means to fit to its model. Yet, the problem is that if α is quite a complicated HP with ample nondeterminism, then it can be rather time-consuming to exhaustively execute all its different choices just to find out whether there is one combination that explains the present state transition in the concrete controller implementation. For that reason, it is better to find a simpler formula that also implies $\langle\alpha\rangle\, x = x^+$ but is easier to evaluate at runtime, for example, a formula that is pure real arithmetic. How else might we construct such a simpler monitor formula $\chi(x,x^+)$ that provably implies $\langle\alpha\rangle\, x = x^+$ as well?

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

The idea to construct such a monitor $\chi(x,x^+)$ implying $\langle\alpha\rangle\, x = x^+$ is about as easy as it is far-reaching. The clou is that we already have a pretty powerful and rigorously correct transformation technique at our disposal: the axioms and proof rules of differential dynamic logic from Parts I and II. All we need to do is to apply them to $\langle\alpha\rangle\, x = x^+$ and find out how that simplifies the formula.

Example 19.6 (Synthesis of controller monitor). The controller monitor (19.2) from Example 19.1 for the bouncing-ball controller (4.24) can also be synthesized systematically with a correct-by-construction approach. We simply start with a hopeless attempt to prove $\langle ctrl\rangle(x{=}x^+ \wedge v{=}v^+)$ for the bouncing-ball controller *ctrl*:

$$
\begin{array}{rl}
 & \vdash (x=0 \to x{=}x^+ \land -cv{=}v^+) \lor (x \neq 0 \to x{=}x^+ \land v{=}v^+) \\ \hline
\langle ?\rangle & \vdash \langle ?x=0\rangle(x{=}x^+ \land -cv{=}v^+) \lor \langle ?x \neq 0\rangle(x{=}x^+ \land v{=}v^+) \\ \hline
\langle :=\rangle & \vdash \langle ?x=0\rangle\langle v:=-cv\rangle(x{=}x^+ \land v{=}v^+) \lor \langle ?x \neq 0\rangle(x{=}x^+ \land v{=}v^+) \\ \hline
\langle ;\rangle & \vdash \langle ?x=0; v:=-cv\rangle(x{=}x^+ \land v{=}v^+) \lor \langle ?x \neq 0\rangle(x{=}x^+ \land v{=}v^+) \\ \hline
\langle \cup\rangle & \vdash \langle ?x=0; v:=-cv \cup ?x \neq 0\rangle(x{=}x^+ \land v{=}v^+)
\end{array}
$$

This formula cannot possibly be proved because not every value of new positions x^+ and velocities v^+ is reachable from every initial position x and velocity v in the bouncing-ball controller. Fortunately! But the purpose is not to prove the runtime monitor formula offline once and for all, but instead to check whether it evaluates to *true* at runtime and, thereby, to complete the above offline proof at runtime. The resulting monitor condition is the remaining premise at the top of the proof:

$$(x=0 \to x{=}x^+ \land -cv{=}v^+) \lor (x \neq 0 \to x{=}x^+ \land v{=}v^+) \tag{19.7}$$

While this is syntactically different than the manually constructed controller monitor (19.2), the two are provably equivalent and can be obtained by minor simplification when taking into account that the evolution domain constraint guarantees $x \geq 0$. But monitor (19.7) has been systematically constructed and is already accompanied by a correctness proof, because it implies $\langle ctrl\rangle(x{=}x^+ \land v{=}v^+)$ by the above dL proof.

All dL-generated runtime monitors are correct-by-construction. How conservative they are can be read off from an inspection of their proofs. If only equivalence axioms and proof rules have been used, then the runtime monitors are exact. This is the case in the proof of Example 19.6. Otherwise, when implication axioms or proof rules are used, then the monitor may be conservative and might cause unnecessary false alarms, but at least its positive answers are perfectly reliable by proof.

The logical transformation generating correct-by-construction model monitors or other models involving differential equations and/or loops is slightly more involved but follows very similar principles [5]. The basic idea is to unroll loops once by axiom $\langle ^*\rangle$ and either skip or follow differential equations in a proof for one control cycle of the appropriate runtime to monitor reaction time.

19.6 Summary

Even if this chapter merely scratched the surface of the technical aspects of synthesizing provably correct runtime monitors [5], it, nevertheless, held particularly valuable lessons in store for the mindful CPS enthusiast. Meddling with models is an inevitable part of working out the design of a cyber-physical system. But, as the name suggests, such models have to include a sufficiently adequate model of the relevant part of the physical world, which is quite a nontrivial challenge in and of itself.

Fortunately, the logical foundations of safe model transfer, nevertheless, provide a way of exploiting differential dynamic logic to generate runtime monitors that are accompanied by correctness proofs implying that if they evaluate to *true*, then the actual system run fits a provably safe CPS model and is, thus, safe itself. Beyond the high-level ideas behind the ModelPlex approach making this idea reality [5], one of the most important take-home lessons is that the combination of offline verification with runtime monitoring concludes proofs about true CPS runs at runtime. This approach enables tradeoffs that analyze simpler models offline while safeguarding them for suitability with ModelPlex monitors at runtime. Except for the fact that the runtime monitors of overly simplistic models may cause more alarms for discrepancies, such a combination leads to better analysis results for simpler models without paying the full price that we pay when entirely ignoring important effects in pure offline models. Of course, even runtime monitors can only provide limited safety recovery when starting out with models with unbounded errors. When using a bouncing-ball model to describe the flight of an airplane, one should not be surprised to find a significant discrepancy when trying it out for real. At the very least, the bounce on the ground won't proceed as planned.

Exercises

19.1 (Correct bouncing-ball monitors). Prove correctness of the monitors for the bouncing ball from Sect. 19.4 in dL's sequent calculus. That is, prove that truth of the controller monitor (19.2) implies existence of a corresponding execution of the controller. Show that truth of the plant monitor (19.5) implies existence of a run of the plant.

19.2 (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 for the time-triggered design in Chap. 9. Convince yourself that they are correct, i.e., can lead to a corresponding dL proof. Discuss the pragmatic difference between the monitors resulting from the event-triggered models and those from the time-triggered designs. Is there a discrepancy that one monitor discovers that the other one does not?

19.3 (Model monitor correctness). Prove that truth of the model monitor (19.6) for the bouncing ball implies existence of a corresponding run of its model (4.24).

19.4 (Controller monitor generation). Extract the respective controller monitor and prove it correct for the models from Exercises 3.9, 4.22, 9.14, and 12.5.

19.5 (*Monitor synthesis). Describe a proof strategy that synthesizes a runtime monitor for an HP that it receives as input. Is the resulting monitor correct-by-construction? Is it possible to change the approach such that the proof strategy synthesizes both a runtime monitor together with a proof of the correctness of the monitor?

References

[1] George E. P. Box. Science and statistics. *Journal of the American Statistical Association* **71**(356) (1976), 791–799. DOI: 10.1080/01621459.1976.10480949.

[2] Albert Einstein. Die Feldgleichungen der Gravitation. *Sitzungsberichte der Preussischen Akademie der Wissenschaften zu Berlin* (1915), 844–847.

[3] Sarah M. Loos. Differential Refinement Logic. PhD thesis. Computer Science Department, School of Computer Science, Carnegie Mellon University, 2016.

[4] Sarah M. Loos and André Platzer. Differential refinement logic. In: *LICS*. Ed. by Martin Grohe, Eric Koskinen, and Natarajan Shankar. New York: ACM, 2016, 505–514. DOI: 10.1145/2933575.2934555.

[5] Stefan Mitsch and André Platzer. ModelPlex: verified runtime validation of verified cyber-physical system models. *Form. Methods Syst. Des.* **49**(1-2) (2016). Special issue of selected papers from RV'14, 33–74. DOI: 10.1007/s10703-016-0241-z.

[6] John von Neumann. The mathematician. In: *Works of the Mind*. Ed. by R. B Haywood. Vol. 1. 1. Chicago: University of Chicago Press, 1947, 186–196.

[7] André Platzer. Differential dynamic logic for hybrid systems. *J. Autom. Reas.* **41**(2) (2008), 143–189. DOI: 10.1007/s10817-008-9103-8.

[8] André Platzer. Logics of dynamical systems. In: *LICS*. Los Alamitos: IEEE, 2012, 13–24. DOI: 10.1109/LICS.2012.13.

[9] André Platzer. A complete uniform substitution calculus for differential dynamic logic. *J. Autom. Reas.* **59**(2) (2017), 219–265. DOI: 10.1007/s10817-016-9385-1.

[10] Erwin Schrödinger. An undulatory theory of the mechanics of atoms and molecules. *Phys. Rev.* **28** (1926), 1049–1070. DOI: 10.1103/PhysRev.28.1049.