Cross-Layer Interactions in CPS for Performance and Certification

Samarjit Chakraborty[1], James H. Anderson[2], Martin Becker[1], Helmut Graeb[1], Samiran Halder[3], Ravindra Metta[4]
Lothar Thiele[5], Stavros Tripakis[6], Anand Yeolekar[4]

[1]TU Munich, [2]UNC Chapel Hill, [3]TDK-Micronas GmbH, [4]TCS Research Pune
[5]ETH Zurich, [6]Northeastern University
Email: `samarjit@tum.de`

***Abstract*—A central challenge in designing embedded control systems or cyber-physical systems (CPS) is that of translating high-level models of control algorithms into efficient implementations, while ensuring that model-level semantics are preserved. While a large body of techniques for designing provably correct control strategies exist in the control theory literature, when it comes to transforming mathematical descriptions of these strategies to an efficient implementation, the available means are surprisingly ad hoc in nature. Among other reasons, this is because of (i) implementation platform details not sufficiently being accounted for in controller models, (ii) side effects introduced in the code generation process, (iii) various compiler optimizations whose impact on the dynamics of the plant being controlled not being properly understood, (iv) the presence of analog components on the implementation platform whose behavior is difficult to model, (v) computation and communication delays that exist in an implementation but were not accounted for in the model, and (vi) also the effects of image/video processing whose accuracy and timing behavior are difficult to model. As we move towards designing autonomous systems, these issues become biting problems on the path to certification, and striking a balance between performance and certification. In this position paper, we discuss some of these challenges – that we formulate as the need for modeling the interactions between various implementation layers in a CPS – and potential research directions to address them.**

I. Performance and Certification in CPS Design

Control algorithms lie at the heart of many safety-critical embedded systems ranging from cars to robots and medical devices. These systems have a tight interaction with physical processes, where the control inputs to these physical processes are computed by hardware/software (or cyber) platforms. The need for modeling and analyzing the cyber and the physical parts in a tightly coupled fashion is espoused in a cyber-physical systems or CPS-oriented design paradigm. However, traditionally *control theory* and *embedded systems* have been pursued separately, not only by distinct research communities, but also by different engineering groups when it comes to designing real-life products like cars.

In an usual development approach, control theorists are concerned with modeling the dynamics of the system or the "plant" to be controlled, and with designing a "controller" that implements a certain control strategy based on the state of the plant. Control theory offers a rich variety of techniques for designing such controllers and for analyzing the joint behavior of the plant and the controller to ensure that certain closed loop properties such as stability are being satisfied. Till this stage, both the plant models and the controllers are in the form of mathematical expressions that either exist on paper or as models in commonly used tools such as MATLAB/Simulink, which also allow the closed loop behavior of the system to be simulated. Such mathematical expressions of controllers serve as a specification for a (software) implementation, and at this stage an embedded systems engineer or an implementation team takes over. The first step in such an implementation is either manually writing code, or generating it from the mathematical expressions of the controller, and mostly a combination of the two. Automated code generation is supported by tools like the Simulink Coder that generates C or C++ code from different specifications such as Simulink models, Stateflow charts or MATLAB functions. The generated code is then cross compiled for a distributed architecture, on which the compiled code is mapped, scheduled, and ultimately run. This code then reads the states of the plant being controlled via sensors, computes control inputs using the read values, and applies these inputs via actuators. The implicit assumption in this development approach is that the closed loop behavior of the real plant and the implemented controller is identical to that of the modelled plant and mathematical description of the controller from the design stage.

Unfortunately, more often than not, this is not the case. In other words, while the plant and controller models meet all the higher-level requirements on the behavior of the joint system, the real plant and the implemented controller do not meet those requirements. In the more extreme case, while the model might be stable, the behavior of the real plant with the implemented version of the controller might be unstable, or might have different settling time or overshoot properties. These deviations in behavior can occur even if the plant model correctly captures all the properties of the real plant to be controlled, because the mathematical description of the controller is based on an idealistic behavior of the hardware/software implementation platform that does not reflect its real operation. This difference results in the need for extensive testing and debugging and is addressed in an iterative trial and error fashion. As implementation platforms continue to be more complex and distributed, this problem is becoming more acute.

Examples of implementation platform features that are typically abstracted away in controller models include (i) side effects introduced in the (C or C++) code generation process, (ii) optimizations introduced by a compiler or even manual implementation choices that might break the semantics of the original model, (iii) computation and communication delays (whereas the controller model might assume that control inputs can be computed instantaneously and are also immediately available at the actuators), (iv) computations using limited-precision representations of numbers, (v) image or video processing algorithms in the feedback control loop whose timing behavior might be highly variable and also whose accuracy might be difficult to analyze, (vi) the presence of analog components whose behavior is difficult to model, and

(vi) finally errors introduced by the hardware stemming from issues like processor aging that are increasingly becoming common. Efforts to incorporate some of these in a straightforward manner – such as worst-case delays involved in reading sensor values, computing control inputs and transmitting them – in the mathematical descriptions of a controller often lead to very pessimistic designs that might not be acceptable in many cost-sensitive domains like automotive. Hence, there is a difficult tradeoff between ensuring model-implementation compliance in controllers and their efficient implementation.

A. Need for cross-layer modeling

In order to address the above issues, in this paper we highlight the need for explicitly modeling the interactions between the different *layers* of a CPS – the model, code, architecture, its timing behavior and numerical accuracy, and the behavior of the hardware, including its analog components. There is also a need for accounting for more implementation platform behaviors in the controller model, rather than abstracting most of the details away as it is done today. In other words, the traditional controller design paradigm that assumes a single centralized controller with global and instantaneous access to all sensors and actuators, that could compute all control inputs accurately, synchronously, and instantaneously, worked in the past. It is no longer suitable for modern distributed implementation platforms, complex code generation techniques and compiler optimizations, complex video processing libraries in the control loop, and the need to ensure both certification and efficiency. Instead, the information processing and sharing architecture and different optimization options available on the implementation platform must be reflected in the controller design. As a result, new controller design techniques, *e.g.*, for designing distributed controllers, are necessary.

Similarly, on the implementation platform side, *i.e.*, in the embedded systems domain, optimizations have traditionally focused on metrics like improving execution time or memory footprint. Here, it might be necessary to step back and focus on "higher level" issues related to the dynamics of the plant being controlled (like stability and control performance measured in terms of settling time, peak overshoot, or the maximum voltage/current needed to ensure the desired control properties). We believe that the embedded systems design and optimization techniques required for this are also not readily available.

B. Combining metrics from computer science and control

While techniques from computer science, and in particular from formal methods, have been used in conjunction with control theory since a number of years (see, *e.g.*, [1], [2]) the two disciplines focus on two completely different classes of metrics. When designing controllers, the usual goals are to ensure some notion of stability, followed by optimizing certain control performance metrics like settling time. However, when implementing the code corresponding such controllers, the focus shifts on optimizing the usual computer science metrics like execution time, memory footprint, utilization of different computation and communication resources, and possibly also energy consumption. But at this stage, the "structure" of the controller and how it interacts with different sensors and actuators, *i.e.*, its communication requirements, have already been frozen from the controller design stage, thereby limiting the optimization of the computer science metrics that are now possible. The scope of the opportunity lost by separating these two classes of optimizations may be appreciated from the fact that there are several hundreds of millions of lines of software code in a modern car, most of which implement different kinds of controllers [3].

By incorporating the code-level optimization metrics in the controller design step, we believe that more efficient implementations could be obtained, while still ensuring the stability of these controllers and satisfying their control performance requirements. However, this could potentially result in completely different controllers compared to the ones that would be obtained by following a traditional control theoretic design that relies on a separation of concerns, and further, new control theoretic techniques would also be necessary for this.

C. Organization of this paper

In the next section we focus on a model to code generation example and argue on the need for more tightly modeling the interactions between these two layers, compared to the conventional approach of first finalizing the model-level design of the controller, followed by trying to generate code that accurately captures the behavior of the model. More importantly, such an approach is not suitable for more modern distributed and multicore architectures. After this, we briefly discuss some of the other implementation layers and related work that has been done, before concluding the paper.

II. Model-Code Interactions

There have been various efforts in the past to ensure that the code generated from a model accurately captures the behavior encoded in the model. This is primary prerequisite in any model-based design. One line of work attempts to preserve the timing semantics of the model – *e.g.*, a synchronous behavior – on an implementation platform that does not adhere to the same (synchronous) timing paradigm [4], [5], [6], [7], [8], [9]. While one possibility is to formally verify the code generator, most commercially available code generators are black boxes that are not amenable to any formal verification. However, there have been various instances of bugs reported in popular code generators [10], [11], [12] which have prompted work on verifying instances of the generated code with respect to the model from which the code has been generated [13]. In addition, there has also been work on analyzing the software implementation of control algorithms independently of the specification, to verify properties like stability [14]. The reasons for ignoring the specification are that sometimes commonly used tools like MATLAB do not have well-defined semantics, whereas lower level languages like C are better defined and verification at a level closer to the implementation is considered to be more useful.

Further, in addition to the code automatically generated from a specification, a considerable amount of code is also handwritten. What they implement can have a considerable influence on the performance of the final implementation and how much deviation it has from the model. Consider the following implementation:

```
loop
  apply_control(u)
  x <- get_plant_state()
  u <- controller_step(x)
  t_next += 10 ms
  sleep until t_next
end loop
```

Here, the plant state is read at the start of the sampling period of 10 ms and the control input is applied at the start of the next period. The controller gain values can be determined assuming this one sample delay. As long as the execution time of the code of the function `controller_step(x)` is bounded by this 10 ms there would be conformance between the model and the implementation. Note that here only the code of the function `controller_step(x)` is obtained from the code generator and the surrounding code is handwritten. With the same `controller_step(x)`, a different implementation might be:

```
loop
  x <- get_plant_state()
  u <- controller_step(x)
  apply_control(u)
  t_next += 10 ms
  sleep until t_next
end loop
```

In this implementation, because of the variability in the execution time of the function `controller_step(x)`, the control input might get applied at different points in time within each sampling period. This is difficult to account for at the model level, resulting in a deviation between the behavior of the model and the implementation. However, since the application of the control input is not delayed till the start of the next period, the performance of this implementation might be better than the former implementation. In order to see the difference between such implementations, we consider a more detailed example in the next section.

A. Control inputs: Hold or apply?

Our example plant to be controlled is a jet transport aircraft in cruise flight, given by its state-space model

$$\dot{x}(t) = Ax(t) + Bu(t) \tag{1}$$
$$y(t) = Cx(t), \tag{2}$$

where x is the four-dimensional state vector (slip angle, yaw rate, roll rate, bank angle), A is the state transition matrix, u the two-dimensional control input vector (rudder, ailerons), and B the input matrix. Finally, vector y is the observable plant state, which will be a subset of x. This model is based on the example included in the Matlab control toolbox[1]. The values of matrices A, B and C are as follows:

$$A = \begin{pmatrix} -0.0558 & -0.9968 & 0.0802 & 0.0415 \\ 0.5980 & -0.1150 & -0.0318 & 0 \\ -3.0500 & 0.3880 & -0.4650 & 0 \\ 0 & 0.0805 & 1.0000 & 0 \end{pmatrix}, \tag{3}$$

$$B = \begin{pmatrix} 0.0073 & 0 \\ -0.4750 & 0.0077 \\ 0.1530 & 0.1430 \\ 0 & 0 \end{pmatrix}, \quad C = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{4}$$

The goal is to stabilize the jet aircraft's attitude in level flight, such that all angles and rates approach zero, *i.e.*, $\dot{x} = x \approx (0,0,0,0)^T$. We always assume the initial state $x(0) = (0,0,0.6109,0.6109)^T$, and initial control input $u(0) = (0,0)^T$.

[1]https://de.mathworks.com/help/control/ug/mimo-state-space-models.html

1) The controller: We consider a discrete-time feedback controller, which provides control inputs u periodically every T_s time units. Towards this, the plant had to be converted from continuous time to discrete time with sampling period T_s. The plant model now becomes:

$$x[k] = \hat{A}x[k-1] + \hat{B}u[k-1] \tag{5}$$
$$y[k] = Cx[k], \tag{6}$$

where k is shorthand for $t = kT_s$ with $k \in \mathbb{Z}$. The feedback controller was designed using a standard pole placement technique on this discrete model, taking the form

$$u[k] = Ky[k], \tag{7}$$

where K are the controller gains.

2) Two implementation variants: We now propose two different ways to implement the controller. The first follows the conventional approach of applying the control input after a delay of one sampling period. On the other hand, the second follows the principle of applying the different components of the control input as and when they are ready, *i.e.*, not wait for *all* the components to be computed before their application.

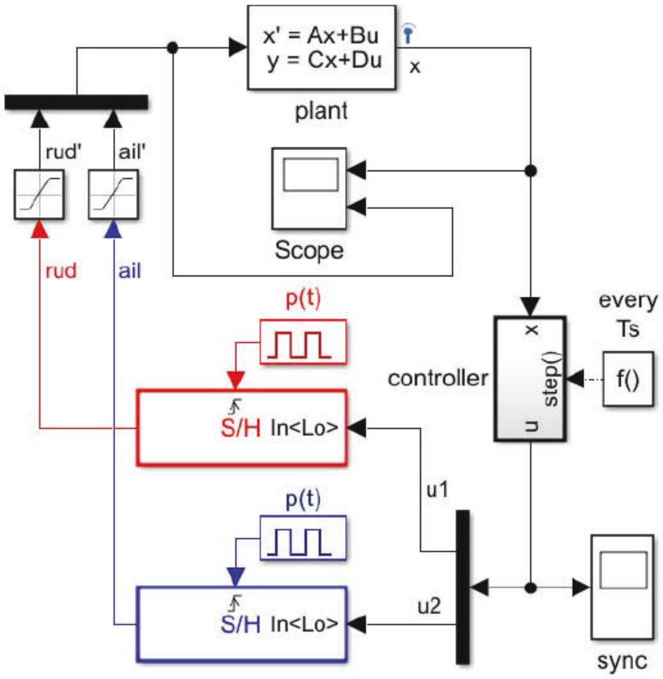

Fig. 1. Conventional implementation: Hold till all control inputs are ready

The conventional implementation follows Equations (5) and (7). That is, $u[k]$ is computed based on the current observed plant state[2], but only applied after a delay of one sampling period T_s. The complete, closed-loop model is depicted in Fig.1. In this model, the controller block is triggered every T_s time units and computes its outputs $u[k]$ instantaneously as in Equation (7). However, u is passed through *sample and hold* blocks ("S/H"), which are triggered once every sampling period and latch the previous cycle's values. The resulting signals ("ail", "rud") are the two components of u, but both delayed by T_s, and thus following Equation (5).

[2]we let $x = y$ in the simulation for vizualization purposes

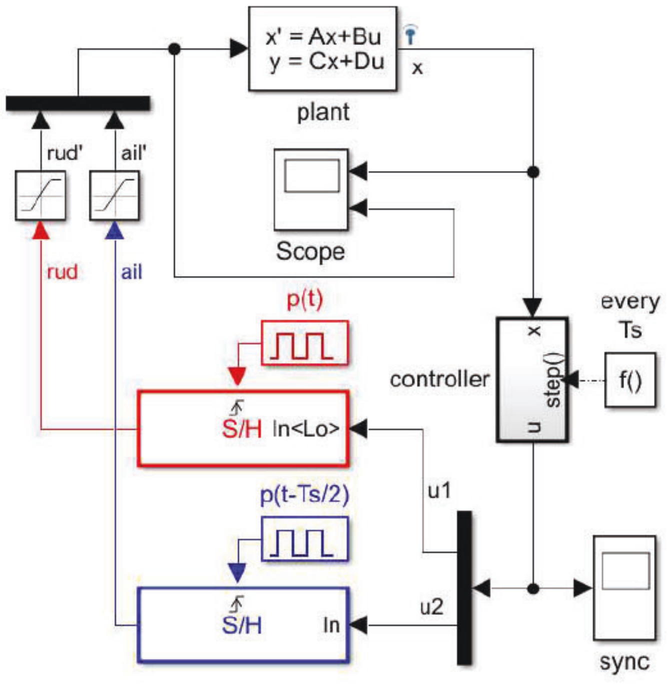

Fig. 2. Deviation from convention: Apply control inputs as and when ready

A different implementation is depicted in Fig.2. Instead of delaying both components of u (*i.e.*, $u1$ and $u2$) by a full period T_s, we assume that the computation of the more critical aileron control component is scheduled first. It can then be applied earlier at time $t = kT_s + Ts/2$, whereas the rudder component remains unchanged. It is computed later and is only updated at time $t = (k+1)T_s$. Towards this, the lower *sample and hold* block is triggered half a period earlier and no longer latches the signal, and everything else remains unchanged.

Note that this now deviates from Equation (5), since x is now updated twice every T_s time units. However, Equation (7) is still followed, ensuring that the plant is sampled at the same rate as earlier.

3) Results - How do the two variants compare: We have evaluated the two implementation variants for two different cases. They show that it is unclear which variant is best, and therefore the second non-conventional variant could be considered useful in some situations.

Case 1: Here, we use a short sampling period of $T_s = 0.01s$. The controller gains have been chosen as

$$K = \begin{pmatrix} -51.8397 & 20.3237 & -5.8464 & -20.3044 \\ -2.1609 & -10.0233 & -50.2046 & -92.6389 \end{pmatrix}.$$

The result can be seen in Figure 3. The closed-loop behavior of both implementation variants is indistinguishably similar, both stabilizing and levelling the aircraft quickly.

Case 2: Now we use a longer sampling period of $T_s = 0.25s$. Note that $\hat{A}$ and $\hat{B}$ carry different values than in the previous case, since the system has now been discretized for a different sampling period. The controller gains for this system have been chosen as

$$K = \begin{pmatrix} -5.1386 & 6.6989 & -0.6119 & -0.6751 \\ 21.0567 & -9.9973 & -22.6222 & -26.1402 \end{pmatrix}.$$

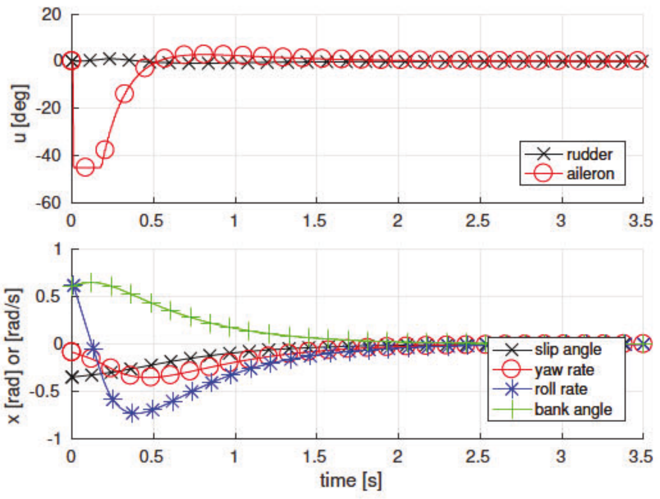

Fig. 3. At high sampling rate both implementations are stable.

The results can be seen in Figures 4 and 5. Whereas the conventional implementation leads to an oscillating aircraft attitude, the second implementation that prioritizes the computation of the aileron control component and applies it earlier is still able to regulate the attitude to a level flight.

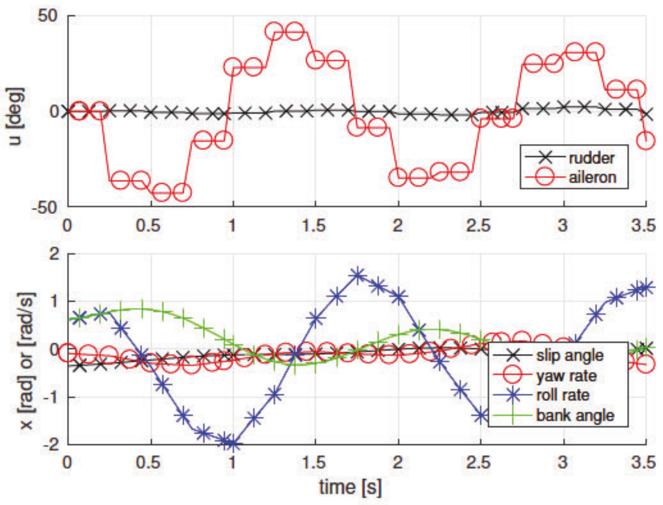

Fig. 4. Conventional implementation becomes unstable at low sampling rate.

B. Lessons learnt: Code generation for modern architectures

Current controller design and implementation paradigms rely on the principle of separation of concerns. The controller model assumes that all the components of the control input are computed and applied synchronously. In order to account for the fact that computations might take non-zero time, Equation 5 is followed, *i.e.*, it is assumed that even in the worst case the computation is bounded by one sampling period. In order to conform to this model in an implementation, even if some components of the control input are ready before the next sampling time, they are held back.

We are of the opinion that such abstractions are not suitable for complex control algorithms, which might *e.g.*, involve video and complex signal processing, and also modern (multicore) processor architectures, compiler optimizations, and distributed implementation platforms. Here, for example, by scheduling the computations of the different components of the

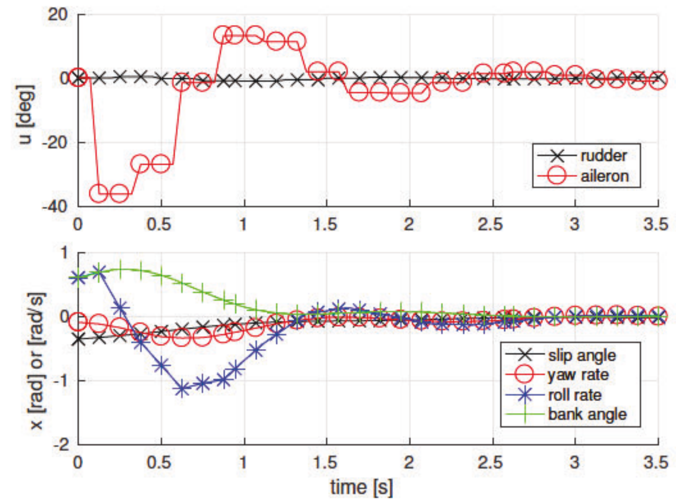

Fig. 5. Implementation in Fig. 2 is stable even at a low sampling rate.

control input appropriately, much better control performance might be obtained. There are two possibilities of realizing this. The first would require the control theorist to understand these implementation-level optimization options and incorporate them in the controller design. This option has been followed in Fig. 2, where the model incorporates the fact that one of the components of the control input should be computed first and applied before the next component is ready. However, this possibility is difficult to realize since the number of implementation and optimization options might be too large.

The second possibility is to automate this procedure, and incorporate it in the code generator. In other words, corresponding to each implementation option, the code generation should construct the model of the corresponding dynamical system (*i.e.*, plant + controller) and be able to analyze its properties. By using the outcome of such analysis, it should decide on the optimal implementation. In other words, the code generation does not attempt to strictly preserve the semantics of the model. In fact, the model in such a case would be more abstract – with just a template of the controller and the desired control performance objectives. Also, unlike conventional code generators or compilers whose goals are typically to minimize execution time or memory footprint, such a code generator would attempt to optimize metrics related to the dynamics of the system.

How to do this is not straightforward. Clearly, there are two main technical hurdles. The first is to automatically construct the underlying model of the dynamical system corresponding to each implementation choice (*e.g.*, the order in which the different components of a control input should be computed and applied) and analyze its performance. The second is to formulate an optimization problem over the space of all such implementation choices. We believe that some recent advances in distributed controller synthesis [15] could be a good starting point for addressing these hurdles.

III. Related work and other layers

As discussed in Section I, in addition to the need for modeling the interactions between the model and the code, various other subsequent implementation layers also impact both, the efficiency of the implementation and also the extent of the deviation from the semantics of the model. In this section we briefly discuss some of these issues and related results in the literature.

A. Model and timing

While controller models often assume that sensor reading are available instantaneously to the controller, control inputs based on these sensor values get computed instantaneously, and these computed values are also communicated instantaneously to the actuators, on many distributed platforms these assumptions are not true. Even if a one sample delay is incorporated in the design (as in Equation 5), the computation and communication times might be highly variable on modern processing platforms [16], making such designs to be over pessimistic.

In the control theory literature, this issue of timing has been widely addressed in the context of *networked control systems* [17]. Here, the focus has been on distributed control systems where control signals are communicated over lossy and delay-prone wireless networks. The goal has been to design control algorithms that can mitigate such delays and packet losses and nevertheless achieve the desired control performance. However, in all of these studies, the wireless network and its loss and delay properties are assumed to be given. In our context, the delays are introduced by the implementation architecture that is not given, but is a part of our design task. Hence, there is the opportunity of co-designing the control algorithms and the implementation architecture (and thereby its delay properties) [18], [19], [20]. It is only recently, that the *architecture design* problem and the co-design of the controller and its sensing, actuation and communication architecture is being studied in the control theory literature [21] and we expect more work in this direction now.

The characterization of the tolerable delays experienced by control signals, while still ensuring stability was studied in [22]. How to incorporate such delay characterization in a schedulability analysis problem however still remains to be seen. Most of the timing/schedulability analysis techniques from the real-time systems literature rely on different task and timing models and there has been much less work on timing models geared towards control applications [23].

B. Image and video processing in the control loop

There is now a rapid growth in the number of applications that involve some image or video processing within a control loop [24]. In other words, one or more sensors in such applications are video cameras or radar sensors, and they are referred to as visual servoing systems [25]. As we move towards adopting autonomous systems – be it in robotics or autonomous vehicles – the need for *certifying* such systems with video processing in the loop is becoming a necessity. There are two main technical challenges involved in this process. The first is that timing analysis of complex video processing applications is non-trivial. Standards like OpenVX offer abstractions with the aim of developing real-time computer vision algorithms, and recently there have been efforts to further improve these abstractions to allow efficient mapping of computer vision tasks on both general purpose and graphics processors [26].

The second challenge is to provide guarantees on the results returned by such image and video processing algorithms. For example, what is the guarantee on the distance estimates returned by a video processing algorithm in an automated cruise control application? This is equivalent to questions on the reliability of the state estimation in closed loop control. While there are techniques to address this within control theory, with the increasing use of machine learning in computer vision algorithms, the verification questions of the above from are becoming more important [27]. Incorporating the uncertainties in such estimates as a lumped error when designing controllers is increasingly becoming too pessimistic.

C. Implementation platform metrics in controller design

In Section I-B we have discussed current design flows where control algorithms are first designed to ensure stability and optimize control performance metrics, *e.g.*, settling time. This is followed by optimizing the generated code for execution time, memory usage and other related metrics, although the high-level (control) algorithm has been frozen by now without taking these issues into account. In order to optimize both classes of metrics, implementation platform features need to be incorporated in the controller design stage. In [28] it has been shown that when trying to optimize the usage of the memory architecture on an implementation platform during the controller design stage, the resulting controllers are fundamentally different from those that would have focused on stability and control performance alone.

Similar conclusions were also drawn in [29] that tried to exploit operating system level task schedules when designing the control algorithms. In all of these cases, the recurring questions that arise are: How to design switched control systems where the switching patterns are known? Or, How to optimally co-design control algorithms and their associated switching strategies? The design and analysis of switched control systems have been extensively studied [30]. But questions such as stability in the case of a *given* (rather than an arbitrary) switching pattern, or the synthesis of switching patterns – that would translate into implementation-level schedules – that ensure stability, have not been sufficiently addressed.

IV. Concluding Remarks

In this paper we tried to highlight the need for modeling and analyzing the interactions between different implementation layers in embedded control systems or CPS design. We conclude that as implementation platforms and compilation techniques become more advanced, traditional abstractions used for controller design and code generation with the aim of preserving model-level semantics are no longer suitable. Instead, more dynamical systems modeling and analysis knowledge needs to be incorporated in code generators and compilers, and new optimization techniques need to be developed. In other words, the focus needs to shift from "lower level" metrics like execution time and memory footprint, to more system level properties that directly relate to the dynamics of the system being controlled. Our discussion of the different layers was by no means exhaustive. In particular, we believe that the modeling of circuit level details (like semiconductor aging and reliability) and the impact of analog components will also play an important role in controller design in the future.

References

[1] R. Alur, T. Henzinger, G. Lafferriere, and G. Pappas, "Discrete abstractions of hybrid systems," *Proceedings of the IEEE*, vol. 88, no. 7, pp. 971–984, 2000.

[2] P. Tabuada, *Verification and Control of Hybrid Systems - A Symbolic Approach.* Springer, 2009.

[3] S. Chakraborty, M. A. A. Faruque, W. Chang, D. Goswami, M. Wolf, and Q. Zhu, "Automotive cyber-physical systems: A tutorial introduction," *IEEE Design & Test*, vol. 33, no. 4, pp. 92–108, 2016.

[4] Y. Yang, S. Tripakis, and A. L. Sangiovanni-Vincentelli, "Efficient distribution of triggered synchronous block diagrams on asynchronous platforms," in *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, 2015.

[5] S. Tripakis, C. Pinello, A. Benveniste, A. L. Sangiovanni-Vincentelli, P. Caspi, and M. D. Natale, "Implementing synchronous models on loosely time triggered architectures," *IEEE Trans. Computers*, vol. 57, no. 10, pp. 1300–1314, 2008.

[6] P. Caspi, N. Scaife, C. Sofronis, and S. Tripakis, "Semantics-preserving multitask implementation of synchronous programs," *ACM Trans. Embedded Comput. Syst.*, vol. 7, no. 2, pp. 15:1–15:40, 2008.

[7] L. Ju, B. K. Huynh, S. Chakraborty, and A. Roychoudhury, "Context-sensitive timing analysis of Esterel programs," in *46th Design Automation Conference (DAC)*, 2009.

[8] L. Ju, B. K. Huynh, A. Roychoudhury, and S. Chakraborty, "Timing analysis of esterel programs on general-purpose multiprocessors," in *47th Design Automation Conference (DAC)*, 2010.

[9] ——, "Performance debugging of Esterel specifications," *Real-Time Systems*, vol. 48, no. 5, pp. 570–600, 2012.

[10] "Simulink code generation bugs," https://de.mathworks.com/support/search_results.html?q=&fq=asset_type_name:bugreport%20product:RT&page=1, 2018.

[11] "Mathworks Embedded Coder bugs," https://de.mathworks.com/support/search_results.html?q=&fq=asset_type_name:bugreport%20product:EC&page=1, 2018.

[12] "MATLAB Coder code generation bugs," https://de.mathworks.com/support/search_results.html?q=&fq=asset_type_name:bugreport%20product:ME&page=1, 2018.

[13] J. Park, M. Pajic, O. Sokolsky, and I. Lee, "Automatic verification of finite precision implementations of linear controllers," in *23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2017.

[14] F. Alegre, E. Feron, and S. Pande, "Using ellipsoidal domains to analyze control systems software," *CoRR*, vol. abs/0909.1977, 2009. [Online]. Available: http://arxiv.org/abs/0909.1977

[15] J. C. Doyle, N. Matni, Y. S. Wang, J. Anderson, and S. H. Low, "System level synthesis: A tutorial," in *56th IEEE Annual Conference on Decision and Control (CDC)*, 2017.

[16] L. Thiele and R. Wilhelm, "Design for timing predictability," *Real-Time Systems*, vol. 28, no. 2-3, pp. 157–177, 2004.

[17] J. P. Hespanha, P. Naghshtabrizi, and Y. Xu, "A survey of recent results in networked control systems," *Proceedings of the IEEE*, vol. 95, no. 1, pp. 138–162, 2007.

[18] D. Roy, L. Zhang, W. Chang, D. Goswami, and S. Chakraborty, "Multi-objective co-optimization of flexray-based distributed control systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016.

[19] D. Goswami, R. Schneider, and S. Chakraborty, "Co-design of cyber-physical systems via controllers with flexible delay constraints," in *16th Asia South Pacific Design Automation Conference (ASP-DAC)*, 2011.

[20] D. Roy, L. Zhang, W. Chang, S. K. Mitter, and S. Chakraborty, "Semantics-preserving cosynthesis of cyber-physical systems," *Proceedings of the IEEE*, vol. 106, no. 1, pp. 171–200, 2018.

[21] N. Matni and V. Chandrasekaran, "Regularization for design," *IEEE Trans. Automat. Contr.*, vol. 61, no. 12, pp. 3991–4006, 2016.

[22] D. Goswami, R. Schneider, and S. Chakraborty, "Relaxing signal delay constraints in distributed embedded controllers," *IEEE Trans. Contr. Sys. Techn.*, vol. 22, no. 6, pp. 2337–2345, 2014.

[23] P. Ramanathan, "Overload management in real-time control applications using (m, k)-firm guarantee," *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, no. 6, pp. 549–559, 1999.

[24] M. Geier, F. Franzen, and S. Chakraborty, "Hardware-accelerated data acquisition and authentication for high-speed video streams on future heterogeneous automotive processing platforms," in *International Conference on Computer-Aided Design (ICCAD)*, 2018.

[25] B. Espiau, F. Chaumette, and P. Rives, "A new approach to visual servoing in robotics," *IEEE Trans. Robotics and Automation*, vol. 8, no. 3, pp. 313–326, 1992.

[26] M. Yang, T. Amert, K. Yang, N. Otterness, J. H. Anderson, F. D. Smith, and S. Wang, "Making OpenVX really 'real time'," in *39th IEEE Real-Time Systems Symposium (RTSS)*, 2018.

[27] R. Ehlers, "Formal verification of piece-wise linear feed-forward neural networks," in *15th International Symposium on Automated Technology for Verification and Analysis (ATVA)*, 2017.

[28] W. Chang, D. Goswami, S. Chakraborty, L. Ju, C. J. Xue, and S. Andalam, "Memory-aware embedded control systems design," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 36, no. 4, pp. 586–599, 2017.

[29] W. Chang, D. Goswami, S. Chakraborty, and A. Hamann, "Os-aware automotive controller design using non-uniform sampling," *TCPS*, vol. 2, no. 4, pp. 26:1–26:22, 2018.

[30] D. Liberzon, *Switching in systems and control.* Springer, 2003.