

Deployment Architectures for Cyber-Physical Control Systems

Shih-Hao Tseng and James Anderson

Abstract—We consider the problem of how to deploy a controller to a (networked) cyber-physical system (CPS). Controlling a CPS is an involved task, and synthesizing a controller to respect sensing, actuation, and communication constraints is only part of the challenge. In addition to controller synthesis, one should also consider how the controller will work in the CPS. Put another way, the cyber layer and its interaction with the physical layer need to be taken into account.

In this work, we aim to bridge the gap between theoretical controller synthesis and practical CPS deployment. We adopt the system level synthesis (SLS) framework to synthesize a state-feedback controller and provide a deployment architecture for the standard SLS controller. Furthermore, we derive a new controller realization for open-loop stable systems and introduce four different architectures for deployment, ranging from fully centralized to fully distributed. Finally, we compare the trade-offs among them in terms of robustness, memory, computation, and communication overhead.

Notation and Terminology: Let \mathcal{RH}_∞ denote the set of stable rational proper transfer matrices, and $z^{-1}\mathcal{RH}_\infty \subset \mathcal{RH}_\infty$ be the subset of strictly proper stable transfer matrices. Lower- and upper-case letters (such as x and A) denote vectors and matrices respectively, while bold lower- and upper-case characters and symbols (such as \mathbf{u} and $\Phi_{\mathbf{u}}$) are reserved for signals and transfer matrices. Let A^{ij} be the entry of A at the i^{th} row and j^{th} column. We define A^{i*} as the i^{th} row and A^{*j} the j^{th} column of A . We use $\Phi_u[\tau]$ to denote the τ^{th} spectral element of a transfer function $\Phi_{\mathbf{u}}$, i.e., $\Phi_{\mathbf{u}} = \sum_{\tau=0}^{\infty} z^{-\tau} \Phi_u[\tau]$.

We briefly summarize below the major terminology:

- **Controller model:** a (linear) map from the state vector to the control action.
- **Realization:** a control block diagram/state space dynamics based on some controller model.
- **Architecture:** a cyber-physical system structure built from basic components.
- **Synthesize:** derive a controller model (and some realization).
- **Deploy/Implement:** map a controller model (through a realization) to an architecture.

I. INTRODUCTION

We consider a linear time-invariant (LTI) system with a set of sensors $s_i, i = 1, \dots, N_x$ and a set of actuators $a_k, k = 1, \dots, N_u$, with plant dynamics

$$x[t+1] = Ax[t] + Bu[t] + d_x[t] \quad (1)$$

Shih-Hao Tseng and James Anderson are with the Division of Engineering and Applied Science, California Institute of Technology, Pasadena, CA 91125, USA. Emails: {shtseng, james}@caltech.edu

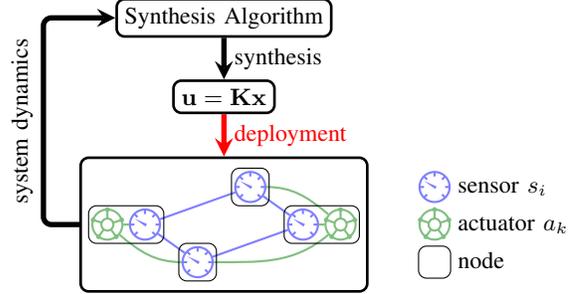


Fig. 1. A model-based system control scheme consists of two phases – synthesis and deployment. We can adopt SLS in the synthesis phase to obtain the optimal controller model \mathbf{K} , and this paper focuses on how to deploy \mathbf{K} to the underlying CPS.

where $x[t] \in \mathbb{R}^{N_x}$ is the state vector, $u[t] \in \mathbb{R}^{N_u}$ is the control, and $d_x[t] \in \mathbb{R}^{N_x}$ is the disturbance. Suppose the system is open-loop stable, i.e., $(zI - A)^{-1}B \in \mathcal{RH}_\infty$. The goal of this paper is to address how a state-feedback controller can be deployed to this system and what the corresponding cyber-physical structures and trade-offs are.

A model-based approach to control design involves two phases: the *synthesis phase* and the *deployment phase*, as illustrated in Fig. 1. In the synthesis phase, the control engineer derives the desired controller model by some synthesis algorithm based on a model of the system dynamics, a suitable objective function, and operating constraints on sensing, actuation, and communication capabilities. The *optimal* controller in the model-based sense is thus the model achieving the best objective value.

Synthesizing an optimal controller for a cyber-physical system is, in general, a daunting task. Recently, a framework named *system level synthesis (SLS)* was proposed to facilitate distributed controller synthesis for large-scale (networked) systems [1]–[3]. Instead of designing the controller itself, SLS directly synthesizes desired closed-loop system responses subject to system level constraints, such as localization constraints [4] and state and input constraints [5]. Using the closed-loop system response, SLS derives the optimal linear controller model, which admits multiple (mathematically equivalent) control block diagrams/state space realizations (or simply *realizations*) [2], [6].

On the other hand, the deployment phase (often referred to as implementation) is concerned with how to map the derived controller model/realization to the target system. We can usually implement one controller realization by multiple different *deployment architectures* (or simply *architectures*). Although all architectures lead to the “optimal” controller,

they can differ in aspects other than the objective, for example; memory requirements, robustness to failure, scalability, and financial cost. Therefore, it is important to consider the *trade-offs* among those architectures, in order to deploy the most suitable architecture.

Approaches to deployment vary greatly in the literature. In the control literature, most work gives the controller design at the realization level and implicitly relies on some interface provided by the underlying CPS for deployment [7], [8]. Providing the appropriate abstraction and programming interface is itself a design challenge [9]–[12]. Some papers also examine the deployment down to the circuit level [13], [14]. The networking/system community, on the other hand, mostly adopts a bottom-up instead of a top-down approach to system control. It usually involves some carefully designed gadgets/protocols and a coordination algorithm [15]–[17].

In this work we take an alternative approach to deployment, which lies between the realization and the circuit level. Rather than binding the design to some specific hardware, we specify the basic components of the system and use them to implement the derived controller realization. As such, we can easily map our architectures to the real CPS – as long as the system supports all basic functions. It is the flexibility of the SLS approach which makes such a process possible.

The paper is organized as follows. In Section II, we briefly review *system level synthesis*, propose a new, simpler control realization, and show it is internally stabilizing. With this realization, we propose four different partitions and their corresponding deployment architectures in Section III, namely, the centralized (Section III-B), global state (Section III-C), naive and memory conservative distributed (Section III-D) architectures. Then, in Section IV we discuss the trade-offs made by the architectures on robustness, memory, computation, and communication. Finally, we conclude the paper with possible future research directions in Section V.

II. SYNTHESIS PHASE

We briefly review the SLS method for the distributed control synthesis phase. Then we describe two “standard” SLS controller realizations and draw attention to their respective architectures for the deployment phase. Finally, we derive a new controller realization for open-loop stable systems which we will show in later sections has many favorable properties in the deployment stage.

A. System Level Synthesis (SLS) – An Overview

To synthesize the state-feedback closed-loop controller for the system described in (1), SLS introduces the *system response* $\{\Phi_x, \Phi_u\}$ transfer matrices. The system response is the closed-loop mapping from disturbance \mathbf{d}_x to state \mathbf{x} and control action \mathbf{u} , under the feedback policy $\mathbf{u} = \mathbf{K}\mathbf{x}$. Compactly this is written as

$$\begin{bmatrix} \mathbf{x} \\ \mathbf{u} \end{bmatrix} = \begin{bmatrix} \Phi_x \\ \Phi_u \end{bmatrix} \mathbf{d}_x.$$

where $\{\Phi_x, \Phi_u\}$ have the realizations $\Phi_x = (zI - A - BK)^{-1}$ and $\Phi_u = \mathbf{K}(zI - A - BK)^{-1}$. It was shown in [2]

that the set of all achievable internally stabilizing controllers is parametrized by an affine subspace.¹ Using this property, *system level synthesis* problem takes the form:

$$\begin{aligned} & \text{minimize } g(\Phi_x, \Phi_u) \\ & \text{s.t. } \begin{bmatrix} zI - A & -B \end{bmatrix} \begin{bmatrix} \Phi_x \\ \Phi_u \end{bmatrix} = I \\ & \quad \Phi_x, \Phi_u \in z^{-1}\mathcal{RH}_\infty \\ & \quad \begin{bmatrix} \Phi_x \\ \Phi_u \end{bmatrix} \in \mathcal{S}. \end{aligned} \quad (2)$$

As long as g is a convex functional, and \mathcal{S} defines a convex set, the SLS problem (2) is convex. For details on how classical optimal control problems can be formulated in this manner, as well as many other aspects of SLS, the reader is referred to [3, §2.2]. The constraint set \mathcal{S} is used to provide spatial and temporal locality to the closed-loop response. This is typically done by enforcing sparsity on the spectral components of $\{\Phi_x, \Phi_u\}$ and making them finite-impulse-response (FIR) filters. Full details can be found in [3], [4].

With (2) solved, a stabilizing controller is readily obtained from the system response:

$$\mathbf{K} = \Phi_u \Phi_x^{-1}. \quad (3)$$

The controller above is useful for theoretical purposes. However, inverting Φ_x is undesirable in most cases as it is heavily dependent on conditioning, and all the structure that Φ_x has will likely be lost. Indeed, a controller realization based on the block diagram in Fig. 2 is more suitable for distributed control. In state space, it takes the form:

$$\begin{aligned} \delta[t] &= x[t] - \hat{x}[t], \\ u[t] &= \sum_{\tau \geq 1} \Phi_u[\tau] \delta[t + 1 - \tau], \\ \hat{x}[t + 1] &= \sum_{\tau \geq 2} \Phi_x[\tau] \delta[t + 2 - \tau], \end{aligned} \quad (4)$$

where $-\hat{x}$ is the output of the $I - z\Phi_x$ block. This controller is a disturbance-rejection controller as the signal δ estimates the disturbance signal \mathbf{d}_x . From a realization perspective, this is better than realizing (3) by a single block. Unlike (3), which inverts Φ_x , any structure imposed on $\{\Phi_x, \Phi_u\}$ is inherited by the two blocks in Fig. 2, i.e., closed-loop constraints are passed on to the controller. One of the main contributions of this paper is to show that there is another controller realization, which, from a deployment perspective, may be more economical to implement whilst maintaining the desirable properties of (4).

B. A Simpler SLS Controller Realization

We have seen in the previous section how the distributed controller obtained from Φ_x and Φ_u can be realized. To give a bit more insight into this design, observe that

$$\mathbf{K} = \Phi_u \Phi_x^{-1} = (z\Phi_u)(z\Phi_x)^{-1}.$$

¹Recent work has shown the equivalence between the SLS closed-loop parameterization and the well-known Youla parameterization [18].

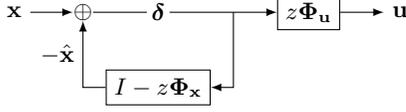


Fig. 2. SLS control block diagram derived in [1]. The signals of interest are $\delta = \mathbf{x} - \hat{\mathbf{x}}$, $\mathbf{u} = z\Phi_{\mathbf{u}}$, and $-\hat{\mathbf{x}} = I - z\Phi_{\mathbf{x}}$. These follow from the block diagram or by taking the z-transform of (4).

The realization in Fig. 2 then follows by putting $z\Phi_{\mathbf{u}}$ in the forward path and realizing $(z\Phi_{\mathbf{x}})^{-1}$ as the feedback path through the $I - z\Phi_{\mathbf{x}}$ block.

Although the design in Fig. 2 is straightforward, and certainly preferable to direct implementation via (3), we note that two convolutions are required to construct $u[t]$. Furthermore, the memory required to hold the convolution kernels grows rapidly with horizon length (of the FIR constraint in S) and number of system states [6]. We will show that when the plant is open-loop stable, we can simplify the design by replacing a convolution with two matrix-multiplies as in Fig. 3. In Section III-B we carry out an explicit cost comparison between the architectures resulting from Fig. 2 and Fig. 3.

Theorem 1. Let $A \in \mathbb{R}^{N_x \times N_x}$ in (1) be Schur stable. The dynamic state-feedback controller $\mathbf{u} = \mathbf{K}\mathbf{x}$ realized via

$$\delta[t] = x[t] - Ax[t-1] - Bu[t-1], \quad (5a)$$

$$u[t] = \sum_{\tau \geq 1} \Phi_u[\tau] \delta[t+1-\tau], \quad (5b)$$

is internally stabilizing and is described by the block diagram in Fig. 3.

Proof. To derive block diagram in Fig. 3, observe that any feasible $\{\Phi_{\mathbf{x}}, \Phi_{\mathbf{u}}\}$ pair satisfies (2), which implies

$$\Phi_{\mathbf{x}} = (zI - A)^{-1}(I + B\Phi_{\mathbf{u}}). \quad (6)$$

In other words, the information of $\Phi_{\mathbf{x}}$ is already encoded in $\Phi_{\mathbf{u}}$. Substituting (6) into the transfer function $\mathbf{K} = \Phi_{\mathbf{u}}\Phi_{\mathbf{x}}^{-1}$, we have

$$\begin{aligned} \mathbf{K} &= \Phi_{\mathbf{u}}(I + B\Phi_{\mathbf{u}})^{-1}(zI - A) \\ &= (z\Phi_{\mathbf{u}})(I + z^{-1}B(z\Phi_{\mathbf{u}}))^{-1}(I - z^{-1}A). \end{aligned} \quad (7)$$

The block diagram follows immediately from (7). The time domain dynamics (5) are obtained from an inverse z-transform.

To prove internal stability, additional perturbations are added to the closed-loop, and it is shown that they decay with time. Consider the plant and controller connected in feedback as in Fig. 4. It is sufficient to examine how the internal states \mathbf{x} , \mathbf{u} , and δ are affected by those external signals. The signals are related via the following equations:

$$\begin{aligned} z\mathbf{x} &= A\mathbf{x} + B\mathbf{u} + \mathbf{d}_{\mathbf{x}}, \\ \mathbf{u} &= (z\Phi_{\mathbf{u}})\delta + \mathbf{d}_{\mathbf{u}}, \\ \delta &= -z^{-1}B(z\Phi_{\mathbf{u}})\delta + (I - z^{-1}A)\mathbf{x} + \mathbf{d}_{\delta}, \end{aligned}$$

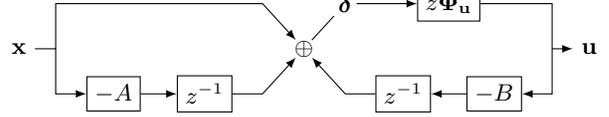


Fig. 3. A simpler control block diagram corresponding to (5). Note the single convolution $z\Phi_{\mathbf{u}}$ according to $\mathbf{K} = \Phi_{\mathbf{u}}\Phi_{\mathbf{x}}^{-1} = (z\Phi_{\mathbf{u}})(I + z^{-1}B(z\Phi_{\mathbf{u}}))^{-1}(I - z^{-1}A)$.

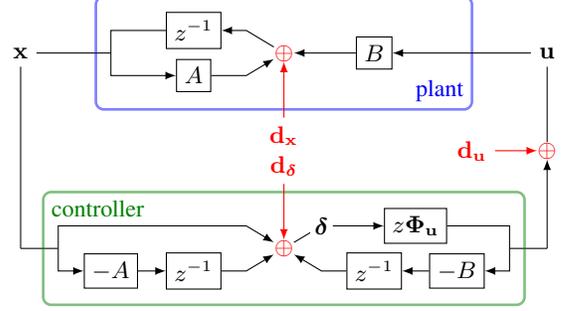


Fig. 4. To show internal stability, we interconnect the proposed controller (Fig. 3) to the plant and examine the effect of injected external signals $\mathbf{d}_{\mathbf{x}}$, $\mathbf{d}_{\mathbf{u}}$, and \mathbf{d}_{δ} on the internal states \mathbf{x} , \mathbf{u} , and δ .

which, after rearranging, gives

$$\begin{bmatrix} \mathbf{x} \\ \mathbf{u} \\ \delta \end{bmatrix} = \begin{bmatrix} \Phi_{\mathbf{x}} & \Phi_{\mathbf{x}}B & (zI - A)^{-1}B(z\Phi_{\mathbf{u}}) \\ \Phi_{\mathbf{u}} & I + \Phi_{\mathbf{u}}B & z\Phi_{\mathbf{u}} \\ z^{-1} & z^{-1}B & I \end{bmatrix} \begin{bmatrix} \mathbf{d}_{\mathbf{x}} \\ \mathbf{d}_{\mathbf{u}} \\ \mathbf{d}_{\delta} \end{bmatrix},$$

where $\Phi_{\mathbf{x}} = (zI - A)^{-1}(I + B\Phi_{\mathbf{u}})$. If all nine entries in the transfer matrix above are stable, then the (bounded) injected signals $\mathbf{d}_{\mathbf{x}}$, $\mathbf{d}_{\mathbf{u}}$, and \mathbf{d}_{δ} produce bounded signal \mathbf{x} , \mathbf{u} , and δ , which implies internal stability. By assumption, $(zI - A)^{-1}B$ is stable, and (2) constrains $\Phi_{\mathbf{x}}$, $\Phi_{\mathbf{u}}$ to $z^{-1}\mathcal{RH}_{\infty}$. It is then easy to verify stability of all nine entries, thus proving the system is internally stable. \square

The advantage of Fig. 3 is that it uses only one convolution $z\Phi_{\mathbf{u}}$. Also, when $\Phi_{\mathbf{u}}$ has finite impulse response (FIR) with horizon T , we have

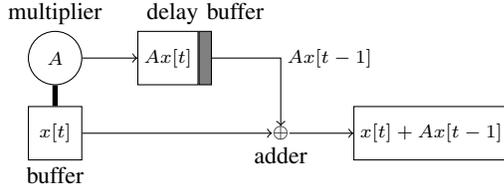
$$u[t] = \sum_{\tau=1}^T \Phi_u[\tau] \delta[t+1-\tau]$$

regardless of whether $\Phi_{\mathbf{x}}$ is FIR or not.

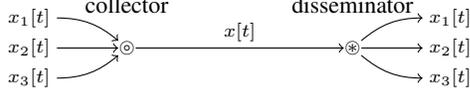
We remark that it is possible to internally stabilize some open-loop unstable systems by the controller in Theorem 1 (Fig. 3). Consider a decomposition of the system matrix such that $A = A_u + A_s$ where A_u is unstable and A_s is Schur stable. Then using the robustness results from [19], the controller designed for (A_s, B) will stabilize (A, B) if $\|A_u\Phi_{\mathbf{x}}\| < 1$ for any induced norm.

III. DEPLOYMENT ARCHITECTURES

We now explore the controller architectures for the deployment phase. The crux of our designs centers on the partitions of the controller realization described in Theorem 1 and illustrated in Fig. 3. We first introduce the basic components. Using those components, we propose the centralized, global



(a) Computation and storage components



(b) Communication components

Fig. 5. The basic functions that a node in the system can perform. These functions serve as the basic components of the proposed deployment architectures. They can be categorized into (a) computation and storage components and (b) communication components.

state, and distributed architectures accordingly. As long as the real system is capable of providing the basic components (or some equivalent parts), mapping the architectures into the system is straightforward.

A. Basic Components

We assume that the nodes in the target system are capable of performing the functions shown in Fig. 5. These functions cover variable storage, arithmetic computation, and communication. Below we describe these functions in details.

The essential component is a *buffer*. A buffer is where the system keeps the value of a variable. It can be a memory device such as a register or merely a collection of some buses (wires). On the other hand, a *delay buffer* is the physical implementation of z^{-1} in a block diagram, which keeps the variable received in the current time step and releases its value in the next time step.

For computation, a *multiplier* with a matrix A senses the variable in a targeted buffer and multiplies it by A as the output. An *adder* performs entry-wise addition of two vectors or matrices of compatible dimensions. Although an adder is a two-input component, we can merge cascaded adders into a multiple-input adder in practice.

A node can also communicate with other nodes through *disseminator-collector pairs*. A *disseminator* sends some parts of a variable to designated nodes. At the receiving side, a *collector* assembles the received parts appropriately to reconstruct the desired variable.

B. Centralized Architecture

The most straightforward deployment architecture is the centralized architecture, which partitions the block diagram as in Fig. 6. It packs all the control functions into one node, the centralized controller. The centralized controller maintains communications with all sensors and actuators to collect state information and dispatch the control signal.

Fig. 7 shows the architecture of the centralized controller. For each time step t , the controller first collects the state information $x_i[t]$ from each sensor s_i for all $i = 1, \dots, N_x$.

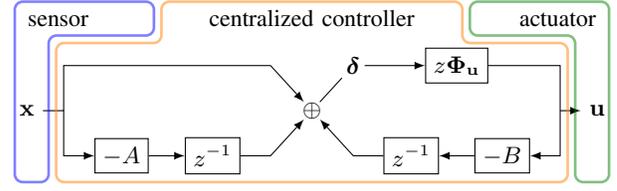


Fig. 6. The partitions of the block diagram for the centralized architecture.

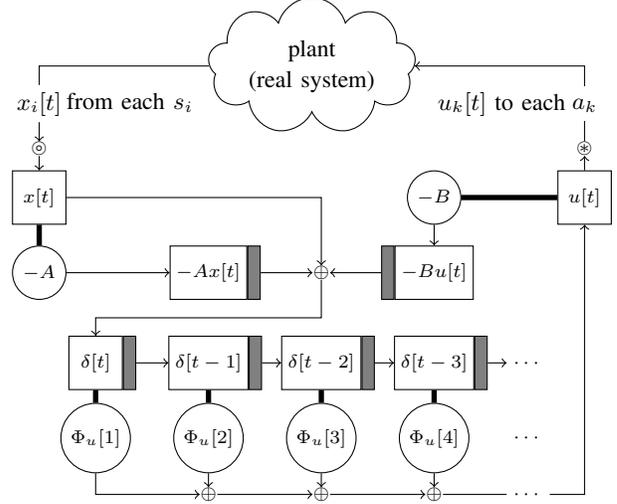


Fig. 7. The architecture of the centralized controller.

Along with the stored control signal u , the centralized controller computes $\delta[t]$ as in (5a). $\delta[t]$ is then fed into an array of delay buffers and multipliers to perform the convolution (5b) and generate the control signals. The control signals $u_k[t]$ are then sent to each actuator a_k for all $k = 1, \dots, N_u$.

Deploying a synthesized solution to the centralized architecture is simple: We take the spectral components $\Phi_u[\tau]$ of Φ_u from $\tau = 1, 2, \dots$ and insert them into the array of the multipliers. Also, A and B are adopted directly from the system model.

We can compare the centralized architecture of Fig. 3 with the architecture of the original block diagram Fig. 2, which is shown in Fig. 8. As mentioned in Section II, the original architecture (Fig. 8) is expensive both computationally and storage-wise. Specifically, when both Φ_x and Φ_u are FIR with horizon T , the original architecture depicted in Fig. 8 performs

$$\underbrace{(T-1)N_x^2}_{\Phi_x[\cdot]\delta[\cdot]} + \underbrace{TN_xN_u}_{\Phi_u[\cdot]\delta[\cdot]}$$

scalar multiplications per time step and needs

$$\underbrace{(T-1)N_x^2}_{\Phi_x[\cdot]} + \underbrace{TN_xN_u}_{\Phi_u[\cdot]} + \underbrace{(T+2)N_x + N_u}_{\delta[\cdot], x[t], -\hat{x}[t+1], \text{ and } u[t]}$$

scalar memory locations to store all variables and multipliers. On the other hand, Fig. 7 performs

$$\underbrace{N_x^2 + N_xN_u}_{-Ax[t] \text{ and } -Bu[t]} + \underbrace{TN_xN_u}_{\Phi_u[\cdot]\delta[\cdot]} \quad (8)$$

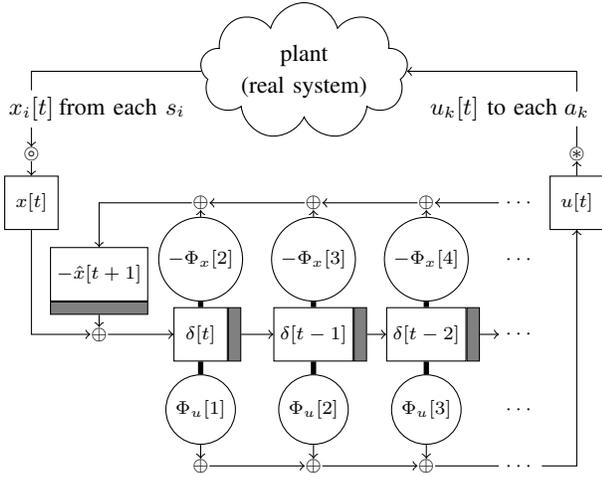


Fig. 8. The architecture of the original controller model in Fig. 2. We remark that (2) implies $\Phi_x[1] = I$ and hence we don't need a multiplier at $\delta[t]$ for the convolution $I - z\Phi_x$. When Φ_x and Φ_u are both FIR with horizon $T > 3$, $N_x \geq N_u$, and $N_x \geq 2$, this architecture requires more computation and storage resources than Fig. 7.

scalar multiplications and needs

$$\underbrace{N_x^2 + N_x N_u}_{A \text{ and } B} + \underbrace{2N_x}_{-Ax[t] \text{ and } -Bu[t]} + \underbrace{TN_x N_u + (T+1)N_x + N_u}_{\Phi_u[\cdot]\delta[\cdot]} \quad (9)$$

storage space, which is more economic when $N_x \geq N_u$, $N_x \geq 2$, and $T > 3$, as is the case for under-actuated systems, and typical for distributed control problems.

We note that the above analysis is based on the most general case – assuming that all matrices are dense. In reality, A and the spectral components $\Phi_x[\cdot], \Phi_u[\cdot]$ can be sparse and structured, which admits some specialized multiplier and buffer designs to significantly reduce the overhead (such as [20]–[22]). This is one reason why the original SLS controller design (Fig. 2 and Fig. 8) is in some cases quite efficient: When \mathcal{S} enforces spatial-localization, the spectral elements $\Phi_x[\cdot]$ and $\Phi_u[\cdot]$ exhibit sparsity patterns, and thus computing all $\Phi_x[\cdot]\delta[\cdot]$ could potentially be easier than obtaining $-Ax[t]$ and $-Bu[t]$. In most cases, the new architecture (Fig. 7) saves computation by replacing $T - 1$ matrix-vector multiplies with two (sparse) matrix-vector multiplies $-Ax[t]$ and $-Bu[t]$.

Despite the intuitive design, the centralized architecture raises several operational concerns. First, the centralized controller becomes the single point of failure. Also, the scalability of the centralized scheme is poor: The centralized controller has to ensure communication with all the sensors/actuators and deal with the burden of high computational load. In the next subsection, we explore some other system architectures to address these issues.

C. Global State Architecture

To avoid overloading the centralized controller, we can offload the computations to the nodes in the system, which

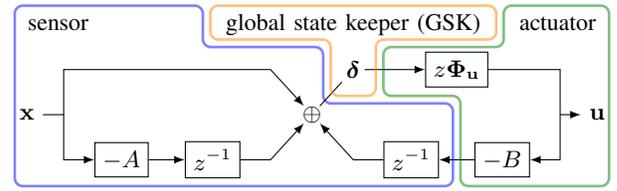


Fig. 9. The partitions of the block diagram for the global state architecture.

results in the global state architecture. Fig. 9 shows the partitions of the block diagram. We have a centralized global state keeper (GSK) which keeps track of the global state $\delta[t]$ instead of the raw state $x[t]$ at each time step t . Rather than directly dispatching the control signals $u[t]$ to the actuators, GSK supplies $\delta[t]$ to the actuators and relies on the actuators to compute $u[t]$.

We illustrate the details of each node in Fig. 10. GSK collects $\delta_i[t]$ from each sensor s_i . To compute $\delta_i[t]$, each s_i stores a column vector $-A^{*i}$. Using the sensed state $x_i[t]$, s_i computes $-A^{ji}x_j[t]$ and sends it to s_j . Meanwhile, s_i collects $-A^{ij}x_j[t]$ from each x_j and $-B^{ik}u_k[t]$ from each a_k . Together, s_i can compute $\delta_i[t]$ by

$$\begin{aligned} \delta_i[t] &= x_i[t] - A^{i*}x[t-1] - B^{i*}u[t-1] \\ &= x_i[t] - \sum_j A^{ij}x_j[t-1] - \sum_k B^{ik}u_k[t-1]. \end{aligned}$$

The $\delta[t]$ term is then forwarded to each actuator by GSK. The actuator a_k can compute the control signal using the multiplier array similar to the structure in the centralized architecture. The difference is that each actuator only needs to store the rows of the spectral components $\Phi_u^{k*}[\tau]$ of Φ_u . After getting the control signal $u_k[t]$, a_k computes $-B^{ik}u_k[t]$ for each sensor s_i .

One outcome of this communication pattern is that s_i only needs to receive $-A^{ij}x_j[t]$ from s_j if $A^{ij} \neq 0$. Similarly, only when $B^{ik} \neq 0$ does s_i need to receive information from a_k . This property tells us that the nodes only exchange information with their neighbors when a non-zero entry in A and B implies the adjacency of the corresponding nodes (as shown in Fig. 14(b)). Notice that this property holds for any feasible $\{\Phi_x, \Phi_u\}$, regardless of the constraint set \mathcal{S} .

The global state architecture mitigates the computation workload of the centralized architecture and hence improves the scalability. However, this architecture is subject to a single point of failure – the GSK. Therefore, we explore some decentralized architectures in the following subsection.

D. Distributed Architectures

To avoid the single point of failure, we can either reinforce the centralized unit by redundancy or deconstruct it into multiple sub-units, each responsible for a smaller region. Here we take the latter option to an extreme: We remove GSK from the partitions of the control diagram (Fig. 9) and distribute all control functions to the nodes in the network.

A naive way to remove GSK from the partitions is to send the state $\delta[t]$ directly from the sensors to the actuators,

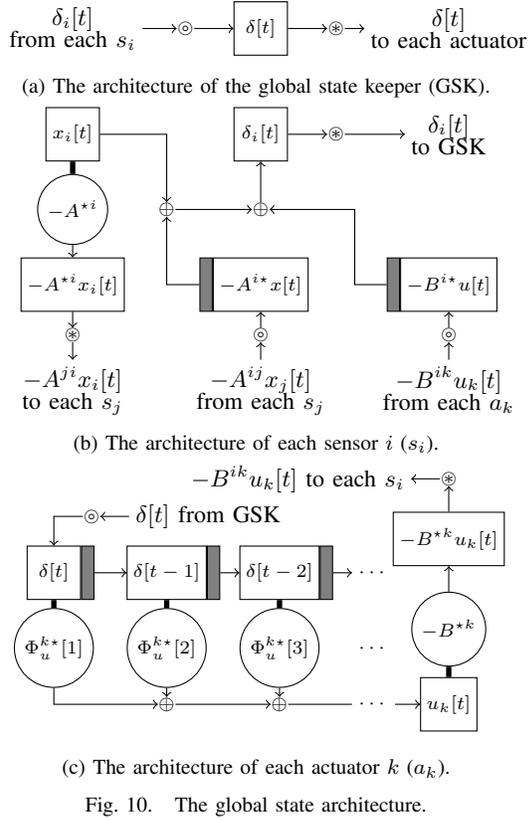


Fig. 10. The global state architecture.

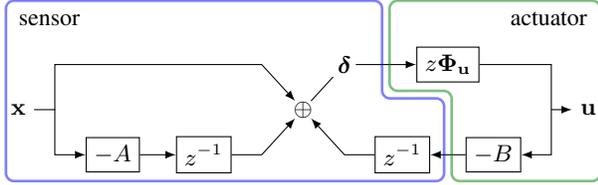


Fig. 11. A naive way to partition the block diagram in a distributed manner: The sensors directly send $\delta[t]$ to each actuator. Those duplicated copies of $\delta[t]$ waste memory resources.

as shown in Fig. 11. More specifically, each sensor s_i would send $\delta_i[t]$ to all the actuators. Each actuator a_k then assembles $\delta[t]$ from the received $\delta_j[t]$ for all j and computes $u_k[t]$ accordingly.

This approach avoids the single point of failure problem. However, it stores duplicated copies of $\delta_i[t]$ at each actuator, which wastes memory resources. To conserve memory usage, we propose to send processed information to the actuators instead of the raw state $\delta[t]$. We depict such a memory-conservative distributed scheme in Fig. 12.

The only difference between Fig. 11 and Fig. 12 is that we move the convolution $z\Phi_u$ from the actuator side to the sensor side. Implementation-wise, this change leads to the architectures in Fig. 13.

In Fig. 13, each sensor s_i not only computes $\delta_i[t]$, but s_i also feeds $\delta_i[t]$ into a multiplier array for convolution. Each multiplier in the array stores the i^{th} column of a spectral component of Φ_u . The convolution result is then disseminated to each actuator.

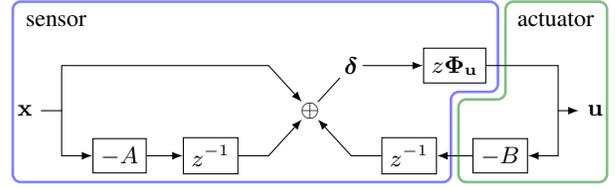


Fig. 12. The partition of the block diagram for the distributed architecture that conserves memory usage.

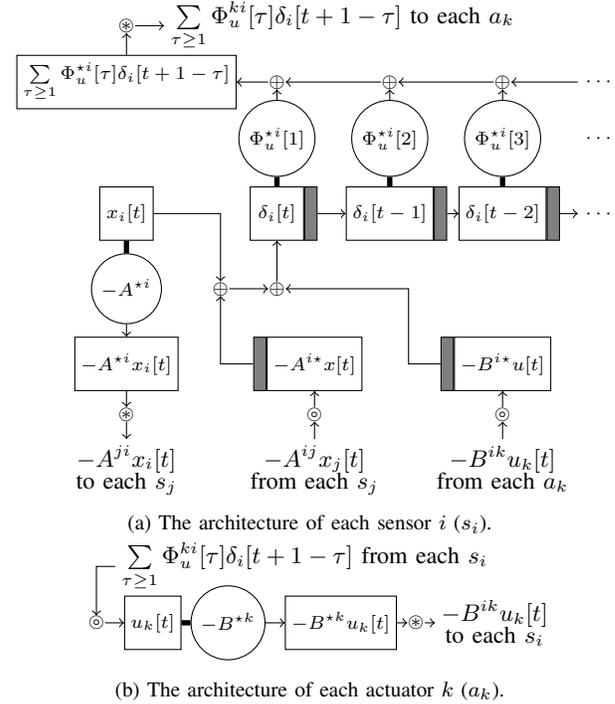


Fig. 13. The memory conservative distributed architecture.

At the actuator a_k , the control signal $u_k[t]$ is given by the sum of the convolution results from each sensor:

$$\begin{aligned}
 u_k[t] &= \sum_{\tau \geq 1} \Phi_u^{k*}[\tau] \delta[t + 1 - \tau] \\
 &= \sum_i \sum_{\tau \geq 1} \Phi_u^{ki}[\tau] \delta_i[t + 1 - \tau].
 \end{aligned}$$

To confirm that Fig. 12 is more memory efficient than Fig. 11, we can count the number of scalar memory locations in the corresponding architectures. Since both architectures store the matrices A , B , and $\Phi_u[\cdot]$ in a distributed manner, the total number of scalar memory locations needed for the multipliers is

$$N_x^2 + N_x N_u + T N_x N_u. \quad (10)$$

We now consider the memory requirements for the buffers. Suppose Φ_u is FIR with horizon T (or, there are T multipliers in the $z\Phi_u$ convolution). In this case Fig. 11 has the same architecture as Fig. 10 without the GSK. Therefore, the

total number of stored scalars (buffers) is as follows:

$$\begin{aligned}
&\text{At each } s_i: N_x + 4, \\
&\text{At each } a_k: (T + 1)N_x + 1, \\
&\text{Total: } N_x(N_x + 4) + N_u((T + 1)N_x + 1) \\
&\quad = (T + 1)N_x N_u + N_x^2 + 4N_x + N_u. \quad (11)
\end{aligned}$$

On the other hand, the memory conservative distributed architecture in Fig. 13 uses the following numbers of scalars:

$$\begin{aligned}
&\text{At each } s_i: N_x + N_u + T + 3, \\
&\text{At each } a_k: N_x + 1, \\
&\text{Total: } N_x(N_x + N_u + T + 3) + N_u(N_x + 1) \\
&\quad = 2N_x N_u + N_x^2 + (T + 3)N_x + N_u. \quad (12)
\end{aligned}$$

In sum, the memory conservative distributed architecture requires fewer scalars in the system, and the difference is $(T - 1)N_x(N_u - 1)$, which is non-negative since T, N_x , and $N_u \geq 1$.

Future work will look at specializing the above computation/memory costs to specific localization constraints. The above expressions serve as upper bounds that are tight for systems that difficult to localize in space (in the sense of (d, T) -localization defined in [4]). For systems localizable to smaller regions, the actuators only collect local information, and hence we don't need every sensor to report to every actuator.

IV. ARCHITECTURE COMPARISON

As mentioned at the beginning of Section III, different architectures implementing the same controller model allow the engineer to consider different trade-offs. Here we compare the proposed architectures and discuss their differences. Our findings are summarized in Table I.

In terms of robustness, the centralized and the global state architectures suffer from a single point of failure, i.e., the loss of the centralized controller or the GSK paralyzes the whole system. This also makes the system vulnerable from a cyber-security perspective. On the contrary, the distributed architectures can still function with some nodes knocked out of the network.²

For information storage, the centralized controller uses the fewest buffers, and the global state architecture stores the most variables (because its GSK has to relay $\delta[t]$). We remark that although the centralized scheme achieves the minimum storage usage at the system level, the single node memory requirement is high for the centralized controller. Conversely, the other architectures store information in a distributed manner, and a small memory is sufficient for each node.

We evaluate the computational load at each node by counting the number of performed multiplication operations. The centralized architecture aggregates all the computation at the centralized controller, while the other architectures perform distributed computing. For distributed settings, the

computation overhead is slightly different at each node. The global state and naive distributed scenarios let the actuators compute the convolution. Instead, the memory conservative distributed architecture puts the multiplier arrays at the sensors.

Finally, we discuss the communication loading of the architectures. In Fig. 14, we sketch the resulting cyber-physical structures of each scheme. The centralized architecture ignores the underlying system interconnection and installs a centralized controller to collect information and dispatch control actions. Under this framework, the sensors and the actuators only need to recognize the centralized controller, but the centralized controller must keep track of all the nodes in the system, which limits the scalability of the scheme.

The global state architecture also introduces an additional node into the system, the GSK, with which all nodes should be contact with. Meanwhile, the sensors and actuators also communicate with each other according to the matrices A and B . In other words, if two nodes are not directly interacting with each other in the system dynamics, they don't need to establish a direct connection.

Similarly, in the distributed architectures, the sensors and the actuators maintain connections according to A and B . Additionally, direct communications, which are governed by the structure of Φ_u , are added to replace the role of GSK. Although it would be slightly more complicated than having a GSK as the relay, we can localize Φ_u at the synthesis phase to have a sparse communication pattern.

Besides the centralized architecture, the physical structure has a direct influence on the cyber structure in all other architectures. As such, we believe further research on the deployment architectures of SLS would lead to better cyber-physical control systems.

V. CONCLUSION AND FUTURE DIRECTIONS

A new internally stabilizing state-feedback controller was derived for systems that are open-loop stable. The controller was shown to have a block diagram realization that is in some ways simpler than the "standard" SLS state-feedback controller.

We considered various architectures to deploy this controller to a real CPS. We illustrated and compared the memory and computation trade-offs among four different deployment architectures: centralized, global state, naive distributed, and memory conservative distributed architectures.

Future work involves removing the open-loop stability requirement and considering the output-feedback setting. Also, as pointed out in Section III-D, there are still many decentralized architecture options left to explore. For example, in addition to the distributed schemes, one can install multiple local controllers to form clustered architectures and employ localization constraints to limit information exchange among clusters.

REFERENCES

- [1] J. C. Doyle *et al.*, "System level synthesis: A tutorial," in *Proc. IEEE CDC*, 2017.

²The system operator might need to update Φ_u to maintain performance – such a re-design is well within the scope of the SLS framework.

TABLE I
COMPARISON AMONGST THE PROPOSED ARCHITECTURES

Architecture Property	Centralized (Fig. 6)	Global State (Fig. 9)	Naive Distributed (Fig. 11)	Memory Conservative Distributed (Fig. 12)
Single Point of Failure	yes	yes	no	no
Overall Memory Usage	lowest (see (9))	highest (equal to $(10)+(11)+N_x$)	second highest (equal to $(10)+(11)$)	second lowest (equal to $(10)+(12)$)
Single Node Memory Usage	high	low (actuator needs more memory)	low (actuator needs more memory)	low (sensor needs more memory)
Single Node Computation Loading	high (see (8))	low (actuator performs convolution)	low (actuator performs convolution)	low (sensor performs convolution)
Single Node Communication Loading	sensor/actuator: low controller: high	sensor/actuator: medium GSK: high	sensor/actuator: high, but localizable	sensor/actuator: high, but localizable

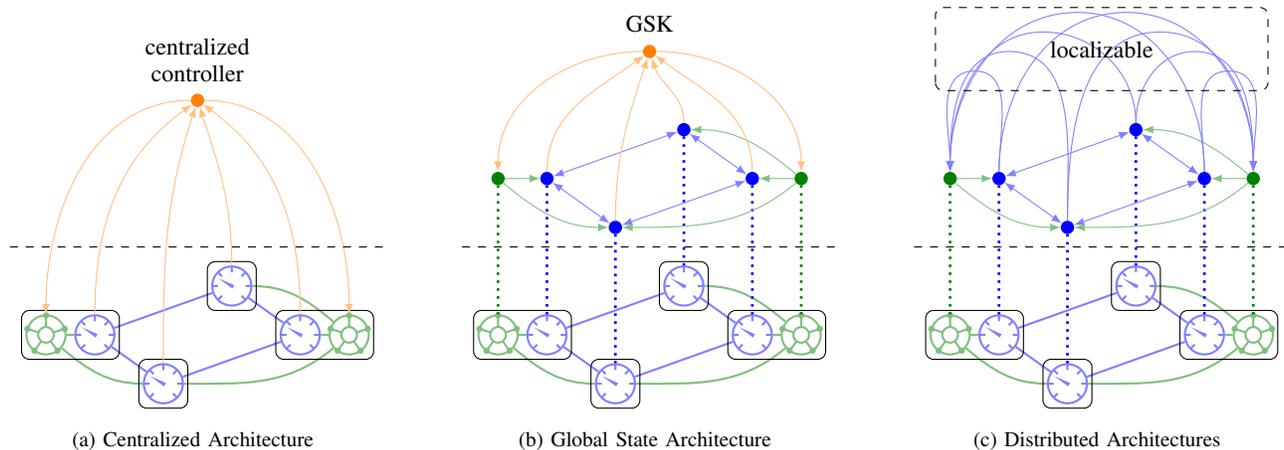


Fig. 14. The cyber-physical structures of the proposed architectures. The horizontal dashed line separates the cyber (top) and the physical (bottom) parts of the system. Each solid node represents a computation unit, and the arrow links are the communication channels. The dotted lines associate the computation units with their locations. The cyber structure of the centralized architecture is ignorant about the underlying physical system, while the other architectures manifest some correlations between the cyber and the physical structures. The distributed architectures (naive and memory conservative) replace GSK by direct communications, which can be trimmed by imposing appropriate localization constraints on Φ_u during the synthesis phase.

- [2] Y.-S. Wang, N. Matni, and J. C. Doyle, "A system level approach to controller synthesis," *IEEE Trans. Autom. Control*, vol. 34, no. 8, pp. 982–987, 2019.
- [3] J. Anderson *et al.*, "System level synthesis," *Annual Reviews in Control*, 2019.
- [4] Y.-S. Wang, N. Matni, and J. C. Doyle, "Separable and localized system-level synthesis for large-scale systems," *IEEE Trans. Autom. Control*, vol. 63, no. 12, pp. 4234–4249, 2018.
- [5] Y. Chen and J. Anderson, "System level synthesis with state and input constraints," *To appear in Proc. IEEE CDC*, *arXiv preprint arXiv:1903.07174*, 2019.
- [6] J. Anderson and N. Matni, "Structured state space realizations for SLS distributed controllers," in *Proc. Allerton*, 2017, pp. 982–987.
- [7] J. Fink, A. Ribeiro, and V. Kumar, "Robust control for mobility and wireless communication in cyber-physical systems with application to robot teams," *Proceedings of the IEEE*, vol. 100, no. 1, pp. 164–178, 2011.
- [8] M. Schwager *et al.*, "Eyes in the sky: Decentralized control for the deployment of robotic camera networks," *Proceedings of the IEEE*, vol. 99, no. 9, pp. 1541–1561, 2011.
- [9] E. A. Lee, "Cyber physical systems: Design challenges," in *Proc. IEEE ISORC*. IEEE, 2008, pp. 363–369.
- [10] S. K. Khaitan and J. D. McCalley, "Design techniques and applications of cyberphysical systems: A survey," *IEEE Systems Journal*, vol. 9, no. 2, pp. 350–365, 2014.
- [11] J. Lee, B. Bagheri, and H.-A. Kao, "A cyber-physical systems architecture for industry 4.0-based manufacturing systems," *Manufacturing letters*, vol. 3, no. 5, pp. 18–23, 2015.
- [12] F. Hu *et al.*, "Robust cyber-physical systems: Concept, models, and implementation," *Future generation computer systems*, vol. 56, no. 12, pp. 449–475, 2016.
- [13] X. Shao, D. Sun, and J. K. Mills, "A new motion control hardware architecture with FPGA-based IC design for robotic manipulators," in *Proc. IEEE ICRA*. IEEE, 2006, pp. 3520–3525.
- [14] J. L. Jerez *et al.*, "Embedded online optimization for model predictive control at megahertz rates," *IEEE Trans. Autom. Control*, vol. 59, no. 12, pp. 3238–3251, 2014.
- [15] J. Hill *et al.*, "System architecture directions for networked sensors," in *ACM SIGOPS operating systems review*. ACM, 2000, pp. 93–104.
- [16] E. Hamed, H. Rahul, and B. Partov, "Chorus: Truly distributed distributed-MIMO," in *Proc. ACM SIGCOMM*. ACM, 2018, pp. 461–475.
- [17] A. Dhekne *et al.*, "TrackIO: Tracking first responders inside-out," in *Proc. USENIX NSDI*, pp. 751–764.
- [18] Y. Zheng *et al.*, "On the Equivalence of Youla, System-level and Input-output Parameterizations," *arXiv preprint arXiv:1907.06256*, 2019.
- [19] N. Matni, Y.-S. Wang, and J. Anderson, "Scalable system level synthesis for virtually localizable systems," in *Proc. of the IEEE 56th Annual Conference on Decision and Control*. IEEE, 2017, pp. 3473–3480.
- [20] R. Vuduc, J. W. Demmel, and K. A. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," in *Journal of Physics: Conference Series*. IOP Publishing, 2005, p. 521.
- [21] S. Williams *et al.*, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in *Proc. ACM/IEEE Supercomputing*. IEEE, 2007, pp. 1–12.
- [22] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA," Nvidia Technical Report NVR-2008-004, Nvidia Corporation, Tech. Rep., 2008.