Coordinated Control for Highly Reconfigurable Systems

(Invited Paper*)

Markus P.J. Fromherz, Lara S. Crawford, and Haitham A. Hindi

Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304, USA
{fromherz,lcrawford,hhindi}@parc.com,
http://www.parc.com/era

Abstract. The remarkable drop in the cost of embedded computing, sensing, and actuation is creating an explosion in applications for embedded software. As manufacturers make use of these technologies, they attempt to reduce complexity and contain cost by modularizing their systems and building reconfigurable products from simpler but smarter components. Of particular interest have recently been highly reconfigurable systems, i.e., systems that can be customized, repaired, and upgraded at a fine level of granularity throughout their lifetime.
High reconfigurability is putting new demands on the software that is dynamically calibrating, controlling, and coordinating the operations of the system's modules. There is much promise in existing software approaches, in particular in model-based approaches; however, current techniques face a number of new challenges before they can be embedded in the kind of real-time, distributed, and dynamic environment found in highly reconfigurable systems. Here, we discuss challenges, solutions, and lessons learned in the context of a long-term project at PARC to bring such techniques to a highly reconfigurable paper path system.

1 Introduction

The remarkable drop in the cost of embedded computing as well as sensing and actuation hardware is creating an explosion in applications for embedded software. Yet while manufacturers are able to add ever more functionality and safety features to their products, they also struggle with the resulting complexity. Increasingly, companies attempt to reduce this complexity, decrease development time, and contain cost by modularizing their systems and building *reconfigurable products* from simpler but smarter networked components. This in turn requires new capabilities from the software that is controlling and coordinating these modules in order to provide an integrated system that is flexible, effective, robust, and safe.

As an example, consider modern high-end printers. One such product comes with about one hundred embedded processors, controlling everything from individual motors in the paper transport to image processing functions to high-level

* Published at HSCC 2005, Zurich, Switzerland. Copyright © 2005 Springer-Verlag.

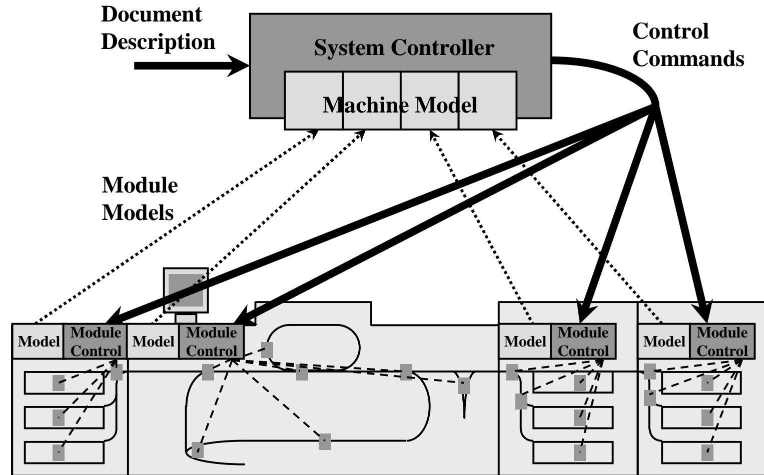

Fig. 1. A modular printing system (feeder, marking engine, and finishers) with model-based auto-configuration and control at three levels

coordination of the entire system to the interaction with the operator. Figure 1 sketches such a system, together with controllers at three different levels (system, module, and component). In this prototypical system, each of the four modules comes with a model, a declarative description of its capabilities, which is passed to the system controller at boot time. This system controller accepts a stream of document descriptions (print jobs) and, using the models, plans and schedules the necessary operations. This results in a stream of control commands to the modules, which in turn control their individual components, many of which have their own low-level controllers. The many controllers in such a system together enable the totally automated operation of a highly complex system that can be considered one of the most sophisticated robots today. These distributed controllers monitor, coordinate, calibrate, optimize, and compensate hundreds of processes with virtually no human involvement.

Today, such high-end print systems are put together from about ten to twenty feeder, marking, and finishing modules. Given the trends and motivations indicated above, it is conceivable that these numbers will increase by an order of magnitude with a corresponding reduction in module size, leading to highly reconfigurable (or hyper-modular) systems. We define a highly reconfigurable system as a modular system that can be reconfigured both in the factory and in the field, often dynamically and at a fine level of granularity. Consequently, there is no final configuration, and both hardware and software modules have to be designed without knowledge of future configurations and other modules that form the context in which a module will operate. Where so far most of a system's behaviors were confined to individual modules, with little regard to concurrent activities in other modules, now most of the behavior comes from the interaction and collaboration of networked, tightly coupled modules.

We believe that highly reconfigurable systems with coordinated control will appear in a number of domains. In some high-end cars, for example, a braking operation already involves the coordinated execution of subsystems such as engine and suspension control, in addition to the coordinated control of the brakes in all four wheels. Today, though, these controllers require careful tuning, and subsystems cannot be upgraded easily. Similarly, in the domain of assembly lines and production systems, retooling and reprogramming the robot stations for new product models sometimes takes days, if not weeks or even months, as much of the equipment works without awareness of the environment. Adding coordinating controllers that can reasons about the capabilities and coupled actions of multiple robots will allow the overall system to adapt automatically when robots are added, upgraded, or replaced over time. In other domains, there are strong incentives to modularize systems from current monolithic designs. In the space exploration domain, for example, weight is a dominant cost factor in the deployment of robots and material. Sustainable planetary missions will only be possible with modular robots and reconfigurable structures that allow for local reuse and reduced material transport across space. Overall, modular architectures promise to lower production, deployment, and maintenance costs and at the same time improve flexibility, performance, and safety. As a consequence, more emphasis will be on the coordinated control of the diverse functions of modular systems.

There is much promise in existing software approaches to address the challenges of highly reconfigurable systems. In particular, *reasoning techniques* such as model-predictive control, model-based planning and scheduling, knowledge-based diagnosis, and intelligent configuration [21] promise powerful solutions to the problems of embedded control and coordination. However, current techniques face a number of challenges that revolve around the *location and communication of knowledge* in a distributed control system, namely knowledge about the system's capabilities, its states, and its goals. In designing architectures and algorithms for such systems, we have to consider where this knowledge is generated, where it will be applied, how it is to be communicated, and how it has to be transformed in order to provide fully integrated system behavior without losing the advantages of high reconfigurability. This leads to the fundamental tension between *module autonomy* and *integrated behavior*: module controllers need to be able to make valid and efficient local decisions that are consistent and even optimal with respect to decisions of other relevant controllers.

This paper discusses challenges, solutions, and lessons learned in the context of a long-term project to embed reasoning techniques in a highly reconfigurable system. We provide a first description of our domain in Section 2. In Sections 3 and 4, we describe the top control design challenges we experienced so far, and we present a set of principles for compositional control that we found useful in addressing these challenges. In Section 5, the approach to our concrete control coordination problem is presented and discussed. We note that the discussion of design challenges and principles will necessarily be somewhat abstract. We invite the reader to jump from Section 2 to Section 5 for a concrete embodiment. We end with conclusions and thoughts about future work.

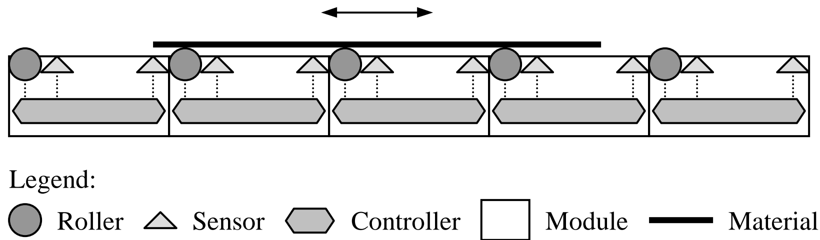

Fig. 2. Modular material path

2 A Simple Domain Example

As a simple model domain, consider a linear sequence of rollers that together are transporting an object, such as a sheet of paper (Figure 2). Each roller is powered by its own independent motor, and each motor is controlled by its dedicated controller. The rollers are spaced such that a typical object will be moved by several rollers at a time, e.g., between two and six consecutive rollers depending on the object's size. For our purposes, we treat each roller with associated motor and controller as a separate transport module. We assume exactly one roller per module and call its controller the module controller. Each module further has associated sensors to detect the presence of the object. In general, all modules do not have to be identical, but instead may differ in their elements (e.g., the number of sensors) and in their behavior characteristics (e.g., velocity and acceleration limits). When used in a production line, there will be multiple parallel and interconnected material paths, with special branching modules for splitting and merging these material paths. We disregard these capabilities in this paper.

At the top level of this system, a centralized planner (and scheduler) receives a series of job requests and determines the overall flow of material to produce these jobs [6, 20]. In the following, we provide a short overview of the planning level. The remainder of this paper will focus on the problem of implementing the planner's output, in particular on controlling the transport of objects along modular reconfigurable material paths.

A typical job description is a set of literals that describe an initial state and a desired output, as in the following example.

Job-23

initial:	goal:
Location(Job-23, Source)	Location(Job-23, Destination)
Blank(Job-23)	Image(Job-23, Black)
Color(Job-23, White)	Color(Job-23, White)
Size(Job-23, A4)	Size(Job-23, A4)
¬Aligned(Job-23)	

In this example, `Source` and `Destination` are virtual locations where all sources or destinations are placed. All other literals describe initial or desired attributes of the job.

The movement of material by transports and the transformation of material by machine actions can be directly translated from the plant model into traditional logical preconditions and effects that test and modify attributes of the material. A simple example is as follows.

Print(?object)	
preconditions:	Location(?object, Machine-2-Input)
	Blank(?object)
	Aligned(?object)
	CanPrintSize(Machine-2, ?size)
effects:	Location(?object, Machine-2-Output)
	¬Location(?object, Machine-2-Input)
	Size(?object,?size)
	¬Blank(?object)
	Image(?object, Black)
duration:	13.2 secs
allocations:	M-2-Printer at ?start + 5.9 for 3.7 secs

This action model describes preconditions before the action, such as its location, and the effects of the action, such as a new location and an image. The model also specifies a duration, with the intended semantics that the logical effects become true exactly when the action's duration has elapsed. Actions can specify the exclusive use of unit-capacity resources for time intervals specified relative to the action's start or end times. For example, the Print action in the example above specifies exclusive use of the M-2-Printer from 5.9 seconds after the start of the action until 3.7 seconds later.

There may be several different sequences of actions and thus different paths that can produce a given job. A typical system may have anywhere from a few to a few hundred transport modules moving objects between the manufacturing stations. The setting is on-line in the sense that additional jobs arrive asynchronously, perhaps several per second, while plans for previous jobs are being executed.

We have implemented various temporal planners adapted to this on-line domain [6, 20]. The overall objective is to minimize the end time of the known jobs. The latest planner [20] uses state-space regression to plan each job. Temporal constraints are used to represent the order and durations of actions and to resolve resource contention. A* search is used to find the optimal plan for the job, in the context of all previous jobs [21].

We assume that each transport module runs its roller at one of several predetermined velocities that are known to the planner through their action models. For each job, the planner produces a plan that states where each object will be at what time as it is transported along its path, taking into account the capabilities of the modules as well as the plans for other objects in the material paths. The necessary control commands are then sent to the selected modules for execution at the selected times. In addition to being physically connected, module actions are further coupled through the object. In our domain, with typical velocities of 0.5 to 3 m/s and roller spacing of no more than 15 cm, the set of modules acting

on the sheet is changing rapidly. Sheets are typically packed tightly in the material path, and sheet collisions are not allowed, so the tolerance for deviations from the scheduled plan is very tight. Also, even minor velocity discrepancies between the modules acting on the sheet may tear the paper. Thus, tight control and coordination of the transport modules is required.

3 Control Challenges in Highly Reconfigurable Systems

In this section, we will reflect on control challenges that we have found in building a prototype of such a highly reconfigurable system. We will focus on the issues arising out of the reconfigurability of the system. We present these challenges in three categories: compositional knowledge, hierarchical control, and distributed coordination.

3.1 Compositional Knowledge

In a compositional system, the capabilities of the system arise out of the capabilities of its constituent modules. In conventional control systems, knowledge about these capabilities is often not available on-line. Even as many components, from stepper motors to anti-lock braking systems, come with increasingly sophisticated built-in controllers, these controllers are typically closed to the outside. For reconfigurable systems, we believe that all factors relevant for interaction should be captured in *formal models* and made available in open modules in order to enable system auto-configuration. It can be surprisingly difficult, though, to describe module features and constraints in a decomposable manner.

A suggested solution from the model-based reasoning community has been the "no function in structure" principle [5], which requires that the laws of the parts of a system may not presume the functioning of the whole. Such laws may include forces on jointly controlled objects, the use of shared resources, and the timing of operations. For example, behavior constraints often restrict the interaction of different actions in the system: "if action `A` happens, action `B` cannot happen at the same time." This constraint on the second operation is expressed directly with respect to the first operation, which may or may not occur in a given configuration. A composable alternative is to express such interactions as constraints on common resources which can then be resolved by a separate coordinator: "action `A` (`B`) will require resource `R`", where `R` is a shared resource, with a resource coordinator that requires that any two actions using this resource must be sequentialized.

In a compositional system, knowledge about module capabilities often needs to be *integrated* at multiple levels of abstraction to plan, schedule, control, and coordinate the actions that will achieve the goals. A first challenge is *model composition*, i.e., integrating the module models into subsystem and system models. It is particularly difficult to integrate information about exceptions and exception handling, i.e., to capture and reason about abnormal behavior. A related challenge is the reuse of lower-level models to generate higher-level models. This

model abstraction could start, for instance, from the detailed models of the individual modules (perhaps even their electromechanical drawings) and automatically generate the higher-level, coarser-grained models for the controllers at supervisory or coordinating levels.

Model abstraction is further complicated by its integration with the *control abstraction* at each level. For example, one abstraction from transport modules to the planning level in our domain is to assume piecewise linear trajectories, where the modules run at constant velocities, and velocities change discontinuously from one module to another when slowing down or speeding up. In reality, of course, the transport modules will likely change the velocity more gradually. While this abstraction makes the planning problem significantly more tractable, it has consequences for the interface and protocol between planner and module controller. Even if all models are written "by hand" ahead of time, these challenges in model granularity, composition semantics, and interface protocols have to be addressed.

3.2 Hierarchical Control

Compositional systems generally call for a *hierarchical control* approach, where the system's actions are monitored and directed at multiple levels of abstraction in time and space. Designing a clean architecture that integrates pre-existing module controllers and allocates the remaining control responsibilities remains a difficult and often domain-specific challenge.

One particular issue where reconfigurable systems pose both a challenge and an opportunity is reconciling the *logical* and *physical architectures* of the control system. The logical architecture specifies the roles and connections of different controllers in the system. The physical architecture specifies where those controllers are executed and what interfaces and protocols are used internally and to interface with the environment. The two architectures are typically conflated in conventional systems. In reconfigurable systems, which often have a certain redundancy in both computing and physical capabilities, the designer has the opportunity to keep the two architectures separate. In fact, with open modules, sensors and actuators may be the only elements whose roles are fixed, and controllers may be assigned their processors based on computation and communication needs. As a consequence, the control system may also become more robust, since control roles can be moved flexibly from failing to healthy components when necessary.

We found the most challenging aspect of hierarchical control to be in *exception handling*. When a transport module in our domain fails to move an object as expected, it is usually not able to correct the problem by itself. Worse, if one object is delayed, other objects may have to be delayed as well in order to avoid collisions, all while the modules try to get the objects back on track. In conventional systems with larger modules, all exception handling can be delegated to individual modules. Internal compensation is not possible with the kind of fine-grained modularity of our domain, as the object is never completely inside just one module. Instead, a module must be able to cooperate with other modules

to correct problems, e.g., to try to speed up or slow down the object. In other words, any unplanned operation has to be understood immediately in context in order to take appropriate action to correct or contain the behavior.

Thus, the challenge in exception handling is how to coordinate compensation and when to escalate recovery when things go wrong. To further illustrate this point, consider some available options in our domain. Using a traffic metaphor, it could be argued that each module controller, or an "object controller", should decide how to direct objects in case of an exception, akin to a car driver who may decide to exit a highway and take an alternative route if she has been told of a traffic jam ahead. However, this would not only require endowing the controller with the full capabilities of the planner, but it would also require coordination with every object in the path. In contrast to traffic, there is often little slack in the system. Conversely, all deviations could be escalated immediately to the planner, which would require tight supervisory control of all module controllers as well as potentially constant replanning. Neither of these options is attractive (or even feasible).

3.3 Distributed Coordination

We have repeatedly emphasized the challenge of coordination among multiple controllers in what is inherently a highly distributed system. While the previous subsection primarily addressed the coordination of different control roles in a control hierarchy, the control of many tightly coupled modules also requires significant lateral coordination. This is the issue that posed the most challenges for us from a control theory point of view. One of these challenges is *observer coordination*. Since both sensors and controllers are distributed, the controllers acting on the same object ideally receive and act on the same sensor data. This requires that sensor updates are shared among all relevant controllers in a uniform fashion. For example, as an object in our model domain moves through multiple transport modules, sensors in different modules will pick up the edge transitions at different times, and this information must be communicated to all other modules in such a way that all module controllers can act on the same information. Solutions need to take into account communication issues such as protocol limitations, network delays, and bandwidth constraints. In some domains, sensor data from multiple sensors may have to be aggregated before it can be sent to the controllers. While sensor fusion is not a new issue per se, reconfigurable systems require that the algorithms are independent of the configuration and potentially compensate for differences in the available modules.

A related challenge is *controller synchronization*. In our domain, it is easiest to guarantee tight tolerances if all module controllers intrinsically behave in the same way (e.g., implement the same control approach). Such controllers will act in synchrony when presented with the same sensor data. Even this simplification of a homogeneous system, however, requires that all the controllers cooperating on the same task are synchronized. In particular, as module controllers join the coordinated action, they need to be brought up to speed, so to speak, before they can be relied upon to help control the joint process. More generally, as

multiple controllers cooperate temporarily on the control of a coupled process, both their control processes and the membership process need to be coordinated. This problem becomes even more complex in a heterogeneous system, with different types of controllers, potentially acting at different time scales, where new controllers cannot learn from existing controllers as easily as in a homogeneous system.

4 Design Principles for Compositional Control

In the process of designing and implementing a control system for our domain, we have identified a number of principles as guidelines for the system's design.

Multi-scale Control. A basic principle is to decompose or aggregate control roles horizontally and vertically guided by the locality and timeliness of knowledge required for the control tasks. This suggests, for example, to separate the control of a module's actuators from the coordination of multiple modules, and it forced us to think deeply about each controller's model and interface.

Closed Loop. Another basic principle is to allow for feedback throughout the system and between all levels. This may be obvious, but it appears that many existing control systems still have a significant amount of open-loop control in both supervisory and low-level controllers. This is often acceptable for well-engineered systems where assumptions about the behavior of subsystems can be built into the controllers. Open-loop control is less suitable for highly reconfigurable systems, with its need for tight synchronization and behavior coordination among multiple modules.

Control Coordination. Where multiple modules interact in an immediate sense and require integrated feedback of their actions, new, "floating" controllers can facilitate the coordination of these modules. Such controllers are associated with a task or a process that is determined by multiple controllers and are logically "between" or "above" the modules. They may be either installed permanently or exist only temporarily and expressly to facilitate a particular task. The use of coordinating controllers was not immediately obvious to us at first. Alternatives would be to assign this role to a single supervisory controller (e.g., the planner) or to apply one of the decentralized coordination techniques commonly used in multi-agent systems (e.g., an auction mechanism). Our analysis suggested, however, that it is more efficient and more powerful to create a temporary task controller that facilitates the coordination of the individual controllers. In our domain, the result is an object controller that coordinates all aspects, from membership to synchronization, of those modules currently acting on an object. In a sense, coordination is adding a wider awareness to the self-awareness of an individual module controller, but restricts it to the task at hand.

Encapsulation. Whenever possible, we try to encapsulate knowledge about module or system behavior together with the algorithms acting on it; in other words, to keep knowledge together, to act where the knowledge is, and not to replicate the same control role at multiple levels in the control hierarchy. This means, for example, that all models for planning and scheduling in our domain are located in (or moved to) the planner, and therefore all decisions about plans, rerouting, and plan exception handling have to be made by the planner. A downside is that many deviations and changes in an execution may have to be escalated to supervisory controllers. This leads to the need for delegation.

Delegation. Delegation of control responsibility may sound like a straightforward hierarchical decomposition of goals or commands to lower and lower levels. However, with the responsibility to control we also want to delegate the responsibility to correct and compensate, within bounds. This is only practically feasible if we also give the lower-level controller sufficient insight into the context within which it will be working. For instance, the planner can give each module controller information about the current state and imminent plans for the rest of the system. To make this efficiently possible, this context information can be summarized and only contain what is relevant to the module, e.g., in form of a safe envelope around the commanded behavior. This tells the module controller how much it can deviate from the plan without violating any constraints (e.g., leading to collisions with other objects). This general principle, to communicate both goals and constraints between controllers, can be applied at all levels in the control hierarchy. Delegation can be quite demanding, since it may require substantial computation at the supervisory or coordinating control levels.

Autonomy. The goal of delegation is to allow individual controllers to monitor and determine their behavior without constant external monitoring and synchronization. Delegation, in other words, asks controllers to control their behaviors with respect to both goals and constraints, and it gives them the autonomy to act locally, and to keep changes local, while their behavior is within bounds.

Escalation. The corollary principle to delegation is escalation. This simply means that controllers report feedback when their behavior is out of bounds as defined by the constraints given by the higher-level controller. The exception is then to be handled by the level that has the necessary information and time horizon to consider all effects of the exception. Delegation, autonomy, and escalation determine a trade-off between locally fast and globally appropriate action.

Explicit Contracts. In a reconfigurable system, the joint principles of control coordination, delegation, autonomy, and escalation are best implemented through explicit representation of capability models, goals, and contexts. In our domain, module models form the contracts from module controllers to the planner of what behaviors can be executed. Conversely, goal constraints used in

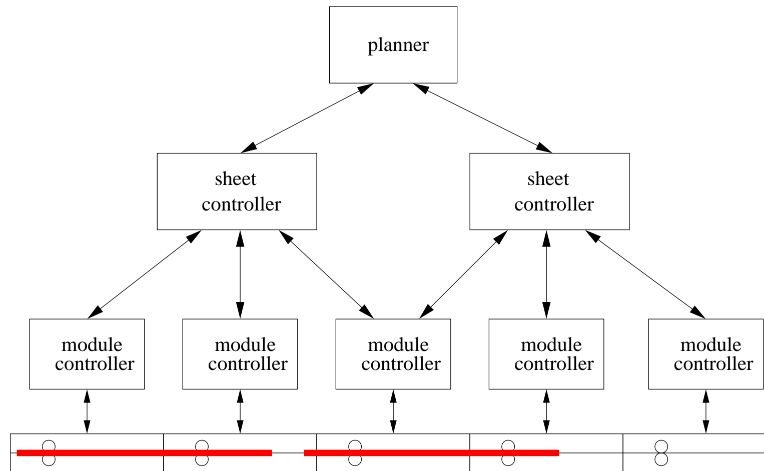

Fig. 3. Control hierarchy. The bottom portion of the diagram shows two sheets inside a sequence of modules controlled by module controllers. Each sheet controller communicates with the module controllers acting on or about to act on its sheet (a dynamically changing set). There is one sheet controller per sheet; a sheet controller is created when each new sheet enters the system. At the top, the planner communicates with all the sheet controllers.

delegation are the contracts from the planner to the module controllers about what they are allowed to change during execution and what not.

These principles together yield a "compositionally aware" system in which knowledge about states and capabilities is shared and control is coordinated as appropriate and no more, in turn leading to a system that is efficient in its separation of concerns and still robust in the face of real-time distributed actions of tightly coupled modules.

5 Coordinated Control for Tightly Coupled, Reconfigurable Modules

This section describes a concrete approach to a specific application, highly modular printing systems. In these systems, as discussed above, there is a strong need for coordinating multi-module behavior. A given sheet of paper is typically in multiple modules at once, so in order to process the sheet correctly without damaging it, the actuators in different modules must cooperate. This coordination is particularly important because of the fast, real-time nature of the system.

5.1 Hierarchical System Architecture

In our hierarchical approach, shown in Figure 3, the coordination task is assigned to an entity called a *sheet controller*. There is one sheet controller per sheet; a

new controller is activated for each new sheet in the system. From the point of view of the planner, the sheet controller serves as a proxy for the sheet control task. The sheet controller communicates over a network with all the module controllers currently interacting with its sheet as well as those about to interact with it (see Figure 3). The sheet controller has three main roles:

- Interpreting the plan for the sheet, translating it into trajectories for the actuators to track, and distributing these trajectories, ensuring that the trajectories of different modules are synchronized
- Serving as a conduit of feedback information between the module controllers
- Monitoring the progress of the sheet through the system and reporting to the planner if necessary

The *module controllers*, in turn, are responsible for tracking the trajectories provided by the sheet controller and thus moving the sheet appropriately. They also have direct access to sensor information. In order to perform tracking, each module controller must maintain a model of the local (single-module) dynamics. As an example of a concrete implementation of a module controller, consider a 2-degree-of-freedom LQG controller [1]. This controller takes as input a reference signal, r, computed from the trajectory supplied by the sheet controller, and the delayed and asynchronous sensor signals, y. The estimation (model) part of the LQG controller is a Kalman filter that includes a simple model of the worst-case network delay. The filter is implemented in a time-varying measurement-update/time-update form to cope with the asynchronous measurements. Throughout this section, we will use such an LQG controller as an illustrative example of a physical module controller, though many of the approaches described here would apply to other implementations as well.

At all levels of the hierarchy, the controller designs are model-based. They make use of the principles of encapsulation, delegation, autonomy, escalation, and coordination. Thus, for example, the sheet controller encapsulates all the sheet-level knowledge, while the module controllers need not know there is such an entity as a sheet at all. Individual module controllers need not even know anything about any other module controllers; they simply communicate with the sheet controller. The sheet controller, in turn, need not know anything about the low-level details of the actuators and sensors. These principles allow the control architecture to achieve locally fast but globally appropriate behavior, an efficient separation of concerns, and robustness under tightly coupled distributed actions.

The next several sections will discuss various elements of our control and coordination approach. We will first describe the sheet controller and its roles in the system in more detail. Then, our mechanism for coordinating feedback among the modules, using the sheet controller, will be further delineated. Next, we will discuss our method for synchronizing the distributed module controllers when they first join a control action (module-module coordination). Then we will describe a method for implementing self-awareness at the module control level. Finally, we will discuss various implementation issues.

There are other control challenges in the domain of high performance copiers that we will not cover here. These include image registration, color consistency

control, banding artifact reduction, and also alternative approaches to (centralized) paper path control. These are covered in the survey paper by Hamby and Gross [7] and in more detail in the references [13, 19, 4, 11].

5.2 Coordinating Control

The sheet controller is responsible for sheet-level concerns. It takes the plan received (delegated) from the planner, which is in terms of tuples of modules, operations, and times, and translates it into positions and times. It divides this trajectory into segments for the individual actuators to follow. To fulfill this responsibility, it must be aware of the machine configuration and the capabilities of the various modules and their actuators. In order for the trajectories to be trackable, the sheet controller may need to smooth the piecewise linear trajectory implied by the plan's waypoints. Since the sheet will be in multiple modules at once, the trajectories generated for different actuators will overlap in time (Figure 4). Trajectories for different actuators must be identical during the overlapping portions in order to avoid damaging the sheet. Should anything go wrong in the system that requires the sheet to be rerouted, the sheet controller must accept updated plans from the planner, create updated trajectories accordingly, and communicate the changes to the module controllers.

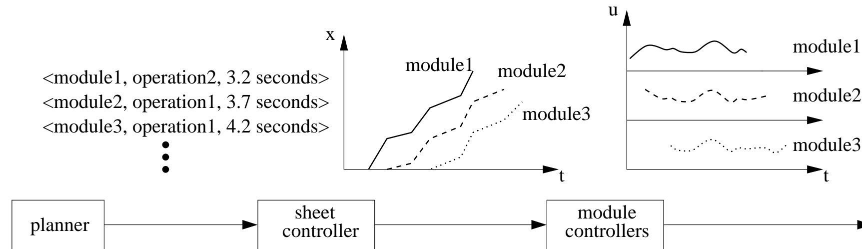

Fig. 4. Plan translation and distribution by the sheet controller. The planner sends a plan in the form of a list of tuples to the sheet controller, which converts it into trajectories for the modules to track. During times of overlap, the trajectories are identical up to positional translation. The module controllers track the identical trajectories, producing synchronized actuator outputs.

The sheet controller also monitors the progress of the sheet. It receives sensor messages from the control modules and uses these to maintain an internal model of the sheet's progress. If something goes wrong, this kind of self-awareness allows it to anticipate unacceptable errors and notify the planner. The definition of an "unacceptable" error can be supplied by the planner based on the separation between neighboring sheets; this knowledge provides a level of context awareness and is important for successful delegation and autonomy. Additionally, since the sheet controller is monitoring the sheet progress in the context of the system

configuration, it can identify where in the system (in which module) the error has occurred. It could then use its sensor information to distinguish between some different types of problems (e.g., a paper jam vs. a simple delay). This information may be useful for escalation, particularly should the system need to reconfigure to avoid blocked or damaged modules.

Finally, the sheet controller serves as a clearinghouse for sensor information from the modules. Each module has edge sensors that detect when the leading or trailing edge of the sheet crosses them. Since multiple modules are acting on the sheet at once, they all need access to this sensor information in order to accurately track the desired sheet trajectories given them by the sheet controller. In order to preserve the encapsulation of knowledge, so that the module controllers need not know about each other, the modules send their sensor data to the sheet controller, which then distributes it appropriately.

5.3 Distributed Feedback

In a tightly coupled system sharing sensor information over a network, timing is crucial in coordinating the distributed feedback. When a sheet sensor is tripped, for example, the module local to that sensor has access to the information immediately. If the module were to use the sensor data right away, it would update its internal model (observer), and its controller would respond well before the other modules could do so, as they must receive the sensor information over the network. The synchronization between module controllers would be destroyed. Therefore, in our design the sensor information is not acted on immediately, but is rather sent to the sheet controller without being used by the module. The sheet controller then determines the set of modules that need that sensor data (that are or are about to be acting on the sheet). It uses its knowledge of the machine configuration to translate the sensor trigger into a sheet position, and sends that data to the relevant modules, along with an *apply time*, t_a. The apply time is based on the maximum network delay and tells the modules when they are allowed to use the data. Given a maximum round-trip network and processing delay (module controller to sheet controller to module controller) d and a sensor trigger time t_s, $t_a = t_s + d$. The modules, including the one that originated the sensor message, wait until t_a to use the sensor data, at which time they all update their internal models simultaneously, preserving their coordination (see Figure 5). This approach assumes that the modules and the sheet controller all have synchronized clocks, so that t_a is the same globally for all modules.

Because the sensor data is not used until the apply time, it is delayed when it is incorporated into the module controllers' observers or other models. The module controllers must therefore save a history of their local state and control values for an amount of time d, so that they can roll back to the appropriate time to apply the sensor data. Thus, at time t_a, the module controller must perform the following steps:

1. Access the saved state from time $t_a - d = t_s$, $x(t_s)$, and update it with the new sensor data. In our example LQG controller, this update consists of performing the measurement update portion of the Kalman filter.

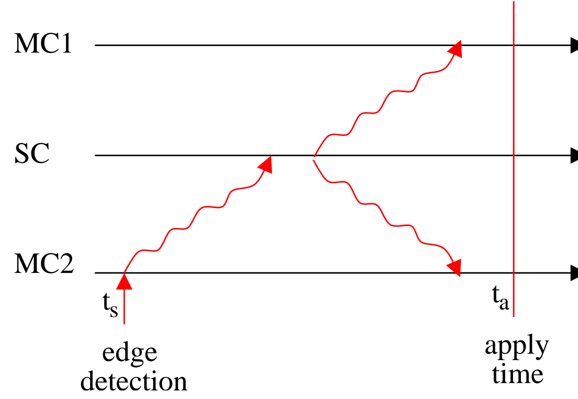

Fig. 5. Diagram illustrating apply time. An edge detection occurs at a sensor in module 2, whose module controller sends that information over the network to the sheet controller. The sheet controller translates the edge detection event into a sheet position and sends the data back to all module controllers involved with the sheet. The module controllers wait until the apply time to make use of the sheet position data.

2. Evolve the state forward from time t_s until time t_a to obtain $x(t_a)$, using the control history, and update the state history accordingly. This step means performing multiple time updates in the Kalman filter setting.

5.4 Distributed Control Synchronization

Feedback coordination, as just described, keeps the module controllers synchronized once they are running. In our system, however, new controllers are constantly being added to (and leaving) the coordinated control action as the sheet passes through the system. Thus, there is also the issue of bringing each new controller into the control action appropriately; new controllers must synchronize with the ones already in the control action. Our approach has been to synchronize the internal controller state directly. The challenge is in obtaining the current state in the presence of network delays. Here, we briefly outline the mechanism from Hindi, *et al.* [9]. Similar work has recently appeared in other areas, notably networked computer games [14, 17].

Consider a set of control processes $\{p_0, \ldots, p_{n-1}\}$, where each process p_i runs the following state based iterations over time $t = 0, 1, 2, \ldots$

$$x_i(t+1) = f\left(x_i(t), y_i(t-d), t\right); \quad x_i(0) = x_{i0}$$
$$u_i(t) = g\left(x_i(t), y_i(t-d), t\right)$$

where x_i is the state, u_i is the control output, y_i is measurement input, d is some nonnegative fixed integer delay (e.g., the worst-case network delay), and f and g are some functions of state, measurement, and time. Note that we make no assumptions about the spaces over which x, u, or y are defined: they could

be numbers, symbols, discrete, or continuous. (For $t < d$, we assume that f and g are functions of only x and t, and that they do not depend explicitly on y. Hence, we can take $y_i(t) = \emptyset$ (undefined) for $t < d$.)

It is clear that if the initial conditions are all equal and the processes are driven with the same measurements, then the states and control outputs are *identical* for all time. In other words, if

$$\begin{aligned} x_{i0} &= x_0; \quad \forall i \\ y_i(t) &= y(t); \forall i, t, \end{aligned}$$

then the processes all run the same recursion:

$$\begin{aligned} x(t+1) &= f\left(x(t), y(t-d), t\right); \quad x(0) = x_0 \\ u(t) &= g\left(x(t), y(t-d), t\right). \end{aligned} \tag{1}$$

Note that this is true for *any* functions f and g of x, y, and t. We refer to such a set of processes, with $x(t)$ and $u(t)$ identical for all time, as *synchronized.*

We are concerned with the following synchronization problem: We seek a method for synchronizing a new process p_n, which starts at some time $t' \geq d$, to the existing processes $\{p_0, \ldots, p_{n-1}\}$, for all time $t \geq t'$. We assume p_n knows f and g, but not x_0. We would like this method to work for *any* choice of functions f and g of x, y, and t.

Synchronization with Delayed Measurements. For any $t' \geq 0$, it follows immediately from (1) that p_n would be synchronized with $\{p_0, \ldots, p_{n-1}\}$ for all time $t \geq t'$, and for any functions f and g of x, y and t, if we set

$$\begin{aligned} x_n(t') &= x(t') \qquad \text{; at time } t' \\ y_n(t-d) &\equiv y(t-d) \text{ ;} \forall t \geq t' \end{aligned} \tag{2}$$

Now suppose that, because of the delay, at the desired synchronization time t', we can receive only $x(t'-d)$ but not the current state $x(t')$. In this case, provided that $t' \geq d$, synchronization proceeds in two phases. First, starting d time steps prior to t', the process p_n must collect a delayed history that is d *time steps deep*, namely $x(t'-d)$ and $\{y(t'-2d), \ldots, y(t'-d-1)\}$. Then, at time t', $x(t')$ is immediately computed by *forward propagating* the state from $x(t'-d)$ to $x(t')$ by performing d iterations of the state recursion in (1) in one time step:

$$\begin{aligned} x(t'-d+1) &= f\left(x(t'-d), y(t'-2d), t'-d\right) \\ x(t'-d+2) &= f\left(x(t'-d+1), y(t'-2d+1), t'-d+1\right) \\ &\vdots \\ x(t') &= f\left(x(t'-d+(d-1)), y(t'-2d+(d-1)), t'-d+(d-1)\right) \\ &\equiv f\left(x(t'-1), y(t'-d-1), t'-1\right). \end{aligned} \tag{3}$$

Thus, p_n is synchronized from t' onwards.

Essentially the same technique can handle the case of asynchronous measurements, where at certain times some of the elements of the measurement sequence $\{y(t-d) \mid t \geq 0\}$ could be missing, but the ones that arrive do so in the right order. This scenario can still be modeled by (1) as follows: at each time t, define $y(t-d)$ as:

$$y(t-d) = \begin{cases} y_m(t-d) & \text{; if measurement arrives} \\ \emptyset & \text{; otherwise} \end{cases}$$

where $\{y_m(t-d) \mid t \geq 0\}$ is some uncorrupted sequence of measurements, and the symbol $\emptyset$ denotes missing measurements. The functions f and g should also be properly defined for values of $\emptyset$. Additionally, in the asynchronous case, synchronization using forward propagation is only possible at times t' when delayed state $x(t'-d)$ is not missing. Otherwise it is necessary to wait until a time at which the delayed state is available [9].

This synchronization mechanism enables distributed module controllers to synchronize with an existing control action. The sheet controller is used as a conduit for sending the state information (as well as the sensor information) to each new controller.

State Machine Implementation. This section gives an example of how the synchronization mechanism can be implemented in practice. The goal in this example is to synchronize p_n to $\{p_0, \ldots, p_{n-1}\}$ from time tDrive until a time tOff. This will be accomplished by *embedding the process in a finite state machine* (FSM), which is shown in Figure 5.4, drawn using Statechart notation [9, 8, 22].

The FSM has four states: off, synch, compute, and drive. In the off-state, the FSM waits until a time tOn, at which point it transitions to the synch-state. The time tOn is chosen to be sufficiently in advance of tDrive, such that there is enough time and enough measurements arrive before tDrive so that synchronization can be completed. The synch-state collects measurements, until a time when it has a delayed measurement history that is d time steps deep *and* a state measurement arrives. At that point it exits the synch-state, initializing the control process by forward propagation. Then it transitions to the compute-state, and executes the entry action, namely performing the first iteration of (1). It then continues to perform the control computation (1), as shown in the do-statement, until a time tDrive, at which point it transitions to the drive-state. The drive-state is very similar to the compute-state, except that the control is actually applied to the target system. Then, at a time tOff, the FSM turns itself off. By embedding the process p_n in an FSM, the desired synchronization can be accomplished in a practical manner.

5.5 Real-time Self-aware Paradigm

Once the module controllers are synchronized, they can track the trajectories supplied by the sheet controller. Though each controller can measure its tracking error through its internal observer, it cannot tell whether this error is considered "large," except possibly through external context cues provided by the

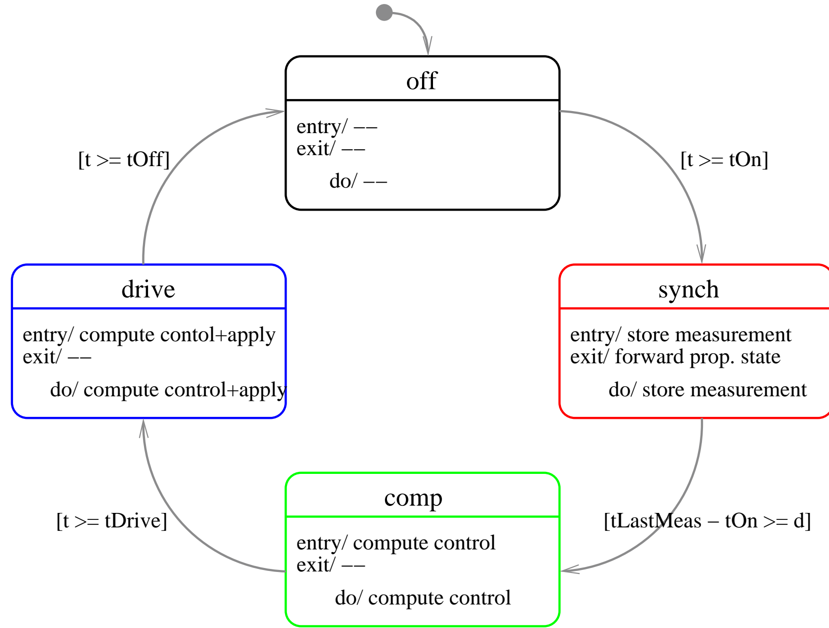

Fig. 6. Simplified finite state machine implementation of synchronization mechanism.

sheet controller. It is useful, therefore, to think about adding an extra level of self-awareness to the module controller, to enable it to better monitor its own performance. The controller could then tell whether it was succeeding or failing to fulfill its contract (in the sense of Section 4) with the sheet controller. If it was failing, it could then decide to employ corrective measures or escalate the problem to the sheet controller. The idea of self-awareness has been gaining recognition, especially in the AI community, the source of our motivation [2]. In this section, we explore this notion in the context of real-time control systems, as presented in Hindi, *et al.* [10].

Modeling Assumptions. We will be concerned with the general discrete time control system shown in Figure 7. The system P is the dynamical plant to be controlled, and K is the controller. The signal $w(t)$ is a sequence of exogenous inputs, and $u(t)$ is the sequence of control inputs. The signal $z(t)$ is the sequence of performance outputs of the system which may not be measurable directly, and $y(t)$ is the sequence of measured outputs which are available to the controller. We assume that both P and K are causal. We refer to a system as *real-time* if, at each time step t, it is able to compute its corresponding output, based on all inputs prior and possibly up to time t, essentially instantly. We assume that both K and P are real-time systems. (P is trivially real-time if it is a physical system.)

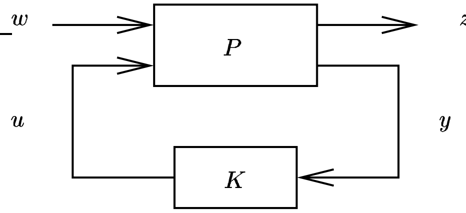

Fig. 7. Generic control system.

We refer to the controller K as *self-aware* if it is able to monitor the progress of its control action and somehow detect if it is failing to achieve its objective; it is *real-time* self-aware if it is able to do this at each point in time. This abstract definition will be made more concrete below.

The following conditions on the system, labeled M, I, and K, are generally assumed to be true:

(M) The plant P is a member of some model set $\mathcal{M}$
(I) The exogenous inputs w are in some input set $\mathcal{I}$
(K) The controller K really is being implemented correctly

The controller is also generally designed to solve some optimization problem with some criterion J, which is usually a function of the performance variable z. We remark that although the condition (K) may seem unnecessary or awkward, experience has shown that getting the software and hardware to implement the controller correctly without bugs or artifacts can be quite nontrivial in practice.

Adding Self-awareness. Self-awareness can easily be added to the controller using ideas from the model validation literature [18, 15]. Suppose that in a situation in which conditions M, I, and K hold, it is also possible to derive certain conditions that the control and the measured output must satisfy, respectively, U and Y[1]. This would be equivalent to the following:

$$(M \wedge I \wedge K) \quad \Rightarrow \quad (U \wedge Y) \tag{4}$$

For example, we could have

(U) The control input u lies in some set $\mathcal{U}$
(Y) The measured output y lies in some set $\mathcal{Y}$

In general, it is not possible to check that U (or Y) is true until the entire sequence $u(t)$ (or $y(t)$) has been observed. However, it is often possible to check for the *violation* of U and Y, at each point in time, i.e., in real-time. For example,

[1] The conditions could be also be joint in u and y, such as $(u, y) \in \mathcal{Z}$. For simplicity we keep them distinct here, but coupling would not change our results.

suppose that U imposes conditions on the signal $u(t)$ that must be true at each point in time:

$$(u \in \mathcal{U}) \quad \Leftrightarrow \quad (u(t) \in \mathcal{U}(t);\ \forall\, t)$$

and similarly for Y. Then violation can be detected by checking that those conditions hold at each time step. Hence, the negation of (4) is more useful for real-time applications:

$$\neg M \vee \neg I \vee \neg K \quad \Leftarrow \quad \neg U \vee \neg Y \tag{5}$$

which is simply a logical statement of the basic model (in)validation paradigm. We remark that in the robust control literature [18, 15], "model (in)validation" has a very specific connotation. However, we will use the term more loosely here, namely, as a reference to any practical implementation of (5).

Equation (5) states that, as long as our physical system satisfies our modeling, input and control assumptions, M, I, and K, then both the measured output and control input will remain in their prescribed sets. However, if the measured output or control input violate conditions U or Y, then we know that at least one of the assumptions on the model, input, or controller is incorrect[2].

The above development shows that real-time self-awareness can be implemented via the augmentation of the controller with *on-line model (in)validation.* Hence, additional machinery would be added to K, which monitors the signals y and u at each time step, and checks that they are consistent with M, I, and K, by checking that U and Y are not violated.

In practice, it might not be possible to perform all the computations associated with Y, U, and the invalidation exactly. This could be either because the computation is too expensive to do in real-time, or simply because we do not even know how theoretically. Thus we will be content with reasonable approximations to these various steps below.

Self-aware LQG Control. As a concrete application of the abstract ideas above, consider the familiar formulation of LQG control [3, 1]:

(M) The plant P is a linear system, with Gaussian initial condition with zero mean and known covariance

(I) The input w is made up of Gaussian i.i.d. processes and sensor noises, which are zero mean, are uncorrelated with each other and the initial condition, and have known covariance

(K) The controller K is a linear LQG optimal controller

[2] Note that the reverse implication does not hold; specifically, it could happen that one of M, I, or K is not true, but Y and U are still true. This could be viewed as a fortuitous situation, where our controller seems to work, even though our original assumptions are false. From a very pragmatic point of view, there should be no objection to this situation, assuming that the variables y and u are able to capture all system parameters and signals of interest.

An important fact about the LQG scenario, which can be used easily in practice, is that all the resulting closed loop states, control signals, and outputs will also be zero mean Gaussian random variables, whose covariances can also be computed explicitly. Thus we can take

(U) $u(t) \sim \mathcal{N}(0, \Lambda_u(t))$
(Y) $y(t) \sim \mathcal{N}(0, \Lambda_y(t))$

where $\mathcal{N}(\mu, \Lambda)$ denotes the normal distribution with mean μ and covariance Λ. Hence an LQG controller can easily implement a very basic level of self-awareness by checking, at each time step, that u and y are within their "5σ" values; otherwise it can flag an error or warning. This can be viewed as a form of (very crude) on-line model validation, in the sense of (5).

Of course, many other more complex options for U and Y are possible, with calculation requirements ranging from solving optimization problems at each time step to maintaining sophisticated real-time statistical estimators of different quantities such as the performance variable z and the objective J. Similar arguments apply to other common control design approaches such as l_1 and $\mathcal{H}_\infty$.

5.6 Implementation

Some of the more painful issues to resolve in developing a prototype often center around integration and implementation. Our first choices for implementation were motivated by the need for rapid prototyping. The module and sheet controllers were developed using C, Matlab, Simulink, and the MathWorks xPC rapid prototyping environment. The xPC environment allows development of code within the Matlab environment that can then be compiled for a target embedded platform. There are, of course, other possible choices for modeling and development environments for real-time embedded systems, such as CORBA [16] or Ptolemy [12].

Within this development framework, the module controllers use a Statechart implementation in C based on Samek's formulation [22] to perform synchronization, as described earlier. The system models and control were developed using linear techniques.

The module controllers in this implementation run on GENE-4310 single-board computers (SBC). Each has a National Semiconductor 300MHz Geode processor (Intel-compatible) and uses less than 16 MB of memory. The controllers use a PC104 interface to two Diamond I/O cards. The SBCs use these I/O boards to link to custom circuitry for the actuators and the sensors. This configuration clearly has more processor power and memory than would be feasible in a real product, but it does allow for freedom of experimentation in the prototyping stage, and is compatible with the xPC environment and operating system. The sheet controllers all run on a single central PC. One could imagine that the sheet controllers could be located on distributed processors as well, and could even travel with their respective sheets; our choice of centralized sheet controllers was made largely for simplicity in early prototyping. The module controllers, sheet

controller PC, and planner communicate with each other over Ethernet, using the UDP protocol. (The xPC environment does not support TCP.) Ethernet and UDP allowed for easy monitoring and debugging of the communications, as well as high bandwidth.

Although our hardware and software environment was useful for rapid prototyping, there were several drawbacks that became apparent throughout the system development and integration process. For example, the xPC environment does allow users to ignore many platform-specific and embedded coding concerns, but it introduces significant overhead into the controllers. Additionally, xPC, Matlab, and Simulink are usable but not all that well suited to developing highly distributed systems with many concurrent components using asynchronous messaging.

6 Conclusion

We have discussed here some of the challenges inherent in the control of highly reconfigurable systems and the design principles we have developed for meeting these challenges. Knowledge modeling, how to formulate what a system knows about itself, others, and the environment, comes to the fore in a distributed, compositional, reconfigurable setting. One aspect of the modeling problem is the question of where knowledge should be located in the system. This issue ties in with that of hierarchical control design, or how to divide up and compose control responsibilities, as control of a system requires knowledge about that system. Finally, a hierarchical, reconfigurable system requires special care in synchronizing control actions and feedback; this is the challenge of distributed coordination. The themes of self-awareness and context awareness run throughout all of these challenges.

We have also described a particular application of these principles to a modular printing system. This implementation makes use of sheet controllers for coordination. Module controllers use state machines to perform synchronization when joining a sheet control action. This synchronization is preserved through distributed feedback using apply times.

The module controllers described here are based on LQG control techniques. They thus do not reason about an explicit on-line model. One direction of future work is to investigate this type of more model-based approach at the module control level. Such an approach could enable further explorations, such as synchronization for heterogeneous module controllers. A model-based reasoning approach would clearly be useful in expanding the self-awareness of the module controllers, enabling them to perform more extensive self-diagnostics and some level of exception handling. Additionally, it would enable them to better interpret some level of context awareness provided by the sheet controller. More detailed modeling of this type might also help in enhancing the sheet controller; for example, the sheet controller could precisely take into account a module's capabilities when smoothing a trajectory for it to track.

There are also a number of larger research issues, three of which we describe here.

One unresolved set of issues centers on control and computing *architectures*. Our system is highly distributed, which gives some benefits in terms of modularity and reconfigurability, but may have a cost in terms of control complexity. A centralized controller becomes infeasible in larger systems with more and more sensors and actuators. Sometimes there may be a happy medium between fully distributed and fully centralized control, but there usually is no single answer. What is missing today is an ability to compare different architectures analytically in terms of control quality, robustness, and reconfigurability, as well as communication and processing requirements.

The architecture question also ties in with the problem of *verification*. With complex, compositional systems, it is increasingly difficult to verify or sometimes even understand their behavior. This complexity is exacerbated when automated optimization or search-based solution methods are used. In these cases, it is not only important to verify correct behavior but also, when a control choice is made, to be able to explain the reasoning behind that choice to external entities, including humans.

Complex, compositional systems also present the issue of *model abstraction*. With each new composition in a model-based system comes the need for a new model at a new granularity, perhaps with a different focus and aimed at a different type of reasoning engine. It would be useful for both development and verification if this model abstraction could be done automatically. Such abstraction, if done properly, would assist both in designing hierarchical architectures and in explaining and verifying compositional behavior.

In closing, highly reconfigurable systems allow designers to make the most of the recent explosions in embedded computing, sensing, and actuator capabilities. Reconfigurable systems can be made up of relatively simple components and rapidly customized for particular needs, thus reducing product complexity and development costs. On-line reconfigurability also enables a high level of system flexibility and robustness. It is now up to software and control engineers to meet the challenges imposed by these new systems and fully realize their potential.

References

1. Astrom, K.J., and Wittenmark, B.: Computer Controlled Systems. Prentice Hall, 1997
2. Bobrow, D., and Fromherz, M.P.J.: Compositional Self-Awareness. DARPA Workshop on Self-aware Computer Systems, Position Statement, May 2004
3. Bryson, A.E., and Ho, Y.-C.: Applied Optimal Control. Academic Press, 1975
4. Chen, C.-L. and Chiu, G.: Incorporating Human Visual Model and Spatial Sampling in Banding Artifact Reduction. American Control Conference, Boston, June 2004
5. de Kleer, J. and Brown, J.S.: A Framework for Qualitative Physics. Proc. of the Sixth Annual Conference of the Cognitive Science Society, 1984, pp. 11–18
6. Fromherz, M.P.J. , Bobrow, D.G., and de Kleer, J.: Model-based Computing for Design and Control of Reconfigurable Systems. AI Magazine, Special Issue on Qualitative Reasoning, vol. 24, no. 4, Winter 2003, pp. 120–130

7. Hamby, E. and Gross, E.: A Control-Oriented Survey of Xerographic Systems: Basic Concepts to New Frontiers. American Control Conference, Boston, June 2004
8. Harel, D.: Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming, vol. 8, 1987, pp. 231–274
9. Hindi, H.A., and Crawford, L.S.: Method and State Machine Implementation of Synchronization of State Based Control Processes with Delayed and Asynchronous Measurements. Palo Alto Research Center (PARC), Internal Report, September, 2004
10. Hindi, H.A., Crawford, L.S., and Fromherz, M.P.J.: Toward Self-aware Real-time Controllers Using Online Approximate Model Validation. Palo Alto Research Center (PARC), Internal Report, December, 2004
11. Krucinski, M., Cloet, C., Horowitz, R., and Tomizuka, M.: A Mechatronics Approach to Copier Paperpath Control. First IFAC Conference on Mechatronic Systems, Darmstadt, Germany, September 2000
12. Lee, E.: Overview of the Ptolemy Project. Technical Memorandum UCB/ERL M03/25, July 2, 2003, University of California, Berkeley, CA, 94720, USA
13. Li, P., Sim, T. and Lee, D.: Time Sequential Sampling and Reconstruction of Tone and Color Reproduction Functions for Xerographic Printing. American Control Conference, Boston, June 2004
14. Mauve, M.: Consistency in Continuous Distributed Interactive Media. ACM CSCW, 2000, pp. 181–190
15. Newlin, M., and Smith, R.S.: A Generalization of the Structured Singular Value and its Application to Model Validation. IEEE Transactions on Automatic Control, 1998, pp. 901-907
16. Object Management Group. http://www.omg.org/
17. Owada, Y., and Asahara, S.: Distributed Processing System, Distributed Processing Method and Client Terminal Capable of Using the Method. US Patent Application Publication US 2002/0194269 A1, December 2002
18. Poolla, K., Khargonekar, P., Tikku, A., Krause, J., and Nagpal, K.: A Time-Domain Approach to Model Validation. IEEE Transactions on Automatic Control, 1994, pp. 951–959
19. Rotea, M. and Lana, C.: A Robust Estimation Algorithm for Printer Modeling. American Control Conference, Boston, June 2004
20. Ruml, W. and Fromherz, M.P.J.: On-line Planning and Scheduling in a High-speed Manufacturing Domain. ICAPS 2004 Workshop on Integrating Planning into Scheduling, Whistler, BC, Canada, June 2004 (updated version submitted to ICAPS 2005)
21. Russell, S. and Norvig, P.: Artificial Intelligence: A Modern Approach, 2nd Ed. Prentice Hall, 2003
22. Samek, M.: Practical Statecharts in C/C++. CMP Books, 2002