UC Berkeley
UC Berkeley Previously Published Works

Title
Metronomy

Permalink
https://escholarship.org/uc/item/6927f1jf

ISBN
9781450330510

Authors
Guo, Liangpeng
Zhu, Qi
Nuzzo, Pierluigi
et al.

Publication Date
2014-10-12

DOI
10.1145/2656075.2656093

Peer reviewed

Metronomy: A Function-Architecture Co-simulation Framework for Timing Verification of Cyber-Physical Systems

Liangpeng Guo
University of California, Berkeley
glp@eecs.berkeley.edu

Qi Zhu
University of California, Riverside
qzhu@ee.ucr.edu

Pierluigi Nuzzo
University of California, Berkeley
nuzzo@eecs.berekeley.edu

Roberto Passerone
University of Trento
roberto.passerone@unitn.it

Alberto Sangiovanni-Vincentelli
University of California, Berkeley
alberto@eecs.berkeley.edu

Edward A. Lee
University of California, Berkeley
eal@eecs.berkeley.edu

ABSTRACT

As the design complexity of cyber-physical systems continues to grow, modeling the system at higher abstraction levels with formal models of computation is increasingly appealing since it enables early design verification and analysis. One of the most important aspects in system modeling and analysis is timing. However, it is very challenging to analyze and verify timing at the early design stages, as the design representation is quite abstract and trade-offs have to be made between the performance requirements defined in terms of system functionality and the cost of the feasible architecture that can implement the functionality. In this paper, we present Metronomy, a function-architecture co-simulation framework that integrates functional modeling from Ptolemy and architectural modeling from the MetroII environment via a mapping interface. Metronomy exploits contract theory for timing verification and design space exploration via co-simulation. Two case studies on an electrical power system and a paper-feed sub-system for a high speed printing press demonstrate the effectiveness of our approach.

Categories and Subject Descriptors

I.6.4 [**Computing Methodologies**]: Model Validation and Analysis; J.6 [**Computer Applications**]: Computer-aided Engineering—*Computer-aided design (CAD)*

General Terms

Design

Keywords

Cyber-Physical System, Co-simulation, Timing

1. INTRODUCTION

The design of a modern cyber-physical system (CPS) is a multi-disciplinary practice carried on by engineers from different domains. As the complexity of such systems continues to grow, it is highly beneficial to separate different design concerns and capture them using formal models of computation (MoCs) for system verification and analysis. In the Platform-Based Design (PBD) methodology [17], two types of models are generally used to represent different design aspects: the *functional model* defines what the design does in terms of a set of services and the *architectural model* describes how these services are implemented by a collection of architectural primitives. Specifically, for CPS control design, the functional model is used to describe the control algorithm and its interaction with the physical plant. The architectural model is used to describe the implementation platform for the control algorithm, including embedded processors, sensors, actuators, communication primitives, as well as the operating system, firmware and drivers. Simulation of the functional model helps verify the functional behavior of the system, while simulation of the architectural model helps evaluate non-functional properties of the implementation platform such as physical time, power consumption, and monetary cost.

However, during a typical design process, trade-offs need to be made between the system performance and the cost of the implementation platform. As an example, a faster control loop can provide better performance but also requires a more expensive implementation platform; on the other hand, a slower control implementation tends to sacrifice performance in order to achieve a cheaper solution. To facilitate the exploration of such trade-offs, it is critical to analyze and verify the real-time performance of a system across the boundary between the functionality and the architecture. In fact, timing properties, such as sampling periods and latencies from sensing to actuation can significantly affect the control performance and even the func-

tional correctness of the design. However, whether certain sampling periods are actually allowed and what the values of the sensor-to-actuator latencies are ultimately depend on the implementation platform. As a result, the design process would largely benefit from bridging functional and architectural models with a co-simulation approach to allow analyzing the system properties that are relevant to both such aspects.

In this paper, we present Metronomy, a modeling and co-simulation framework that bridges the functional and the architectural aspects of the design. In Metronomy, the functional model is captured in the Ptolemy modeling environment [16], while the architectural model is described in the MetroII design environment [5]. Ptolemy provides a rich set of commonly used MoCs (e.g. dataflow, state machines, discrete event, discrete time) to effectively model and simulate the system functionality and its interaction with the physical plant. MetroII provides support for architectural modeling, in particular for models in SystemC [9], and for interfacing functional and architectural models. Metronomy is a natural framework for multi-domain system engineering and integration. Control engineers can leverage the plethora of MoCs made available by Ptolemy to capture the functionality of their controller; software/hardware engineers can benefit from the flexibility of MetroII and SystemC to design the architectural platform; system engineers can effectively combine the two aspects in a co-simulation environment to explore the whole design space and verify correctness and performance of their design.

To support multi-domain system engineering and integration, we exploit contract-based design theory [18] to facilitate timing verification and design space exploration using co-simulation. A timing contract can be seen as a set of timing assumptions and guarantees that are agreed upon by the control engineers, who develop the functional model, and the software/hardware engineers, who design the architectural platform for implementation. We implement timing checkers in Metronomy to monitor whether both the functional and architectural timing assumptions and guarantees are satisfied during co-simulation.

The separation between functional behavior and execution platform is adopted by several design frameworks for software/hardware co-design, mainly in the areas of multimedia and signal processing. Representative frameworks include MAPS [4], featured with a variety of mapping heuristics, Daedalus [13], featured with multi-level design space exploration, Spade [12], featured with a Y-chart based approach, and Sesame [15], featured with trace-based design space exploration. However, most of these frameworks only support Kahn Process Networks (KPN), Dataflow or similar MoCs. Furthermore, all previous works are based on the assumption that functional behaviors can be pre-determined, and captured by the functional model, while the implementation platform only affects the system performance. This assumption does not necessarily hold in CPS design, where the function is tightly intertwined with the physical plant (or environment). Different behavioral timings may trigger different reactions from the environment, which result in different further behaviors of the system.

A preliminary attempt at integrating Ptolemy and MetroII is presented in [11]. However, the framework in [11] can only support a simple MoC, which schedules actors periodically, and lacks the capability of handling heterogeneous MoCs

(e.g. continuous time, discrete time). With respect to [11], this work offers a formalization of the interactions between functional and architectural models in terms of contracts for timing verification and system integration. The use of contracts to analyze the complex coupling of timing and behaviors has been first advocated in [18]. However, while a few rigorous contract theories have been developed over the years (e.g. see [2, 6]), the concrete application of contracts for timing verification in CPS has not been throughly explored. In [7] different types of timing contracts, such as the Logical Execution Time (LET) [10, 8] and Bounded Execution Time (BET) contracts, denoted as "design contracts", are informally presented as a mean to facilitate the independent refinement of functionality and architecture. In this work, based on the theoretical foundations in [2], we propose a formalization of the timing constraints of a design in terms of assume-guarantee contracts, expressed as assertions on system traces, i.e. sequences of events. Such a formalization is general enough to encompass different kinds of design contracts, and suitable for building monitors to validate them via co-simulation.

The contribution of this paper is threefold: (a) we formalize the interactions between the functional model and the architectural model via the concept of timing contract; (b) we propose a methodology for timing contract verification and design space exploration through co-simulation; (c) we implement a function-architecture co-simulation framework that supports the methodology, based on the Ptolemy and MetroII design environments.

The rest of this paper is organized as follows. Section 2 gives a brief introduction to Ptolemy and MetroII, and presents the motivation for integrating them. Section 3 formalizes the concept of timing contract between functional and architectural models, and describes an illustrating example. Section 4 presents the methodology for timing verification and design space exploration using Metronomy. Section 5 describes several key aspects of the implementation of Metronomy. Section 6 shows its application to the design of an aircraft electrical power system and a paper-feed subsystem of a high-speed printing press. Finally, Section 7 presents the conclusions.

2. BACKGROUND

In CPS design, the system functionality is typically modeled in one environment (e.g. Simulink, Modelica) by control engineers, while the system architecture is modeled in a different one (e.g. SystemC or other programming languages) by software and hardware engineers. Since each design environment has its own strengths, it is not convenient to force engineers to use a uniform design environment. On the other hand, efficiently integrating and analyzing functional and architectural models from different environments are challenging tasks. The contract-based methodology in this work aims to address these issues by formalizing the interactions between system function and architecture, and by providing a framework to efficiently co-simulate and co-analyze functional and architectural models. In what follows, we provide details on the modeling environments used in our proof-of-concept implementation.

2.1 Ptolemy

Ptolemy II is a modeling and simulation environment for heterogeneous systems, which consists of several executable

domains of computation that can be mixed in a hierarchy [3, 16]. All MoCs are described operationally in terms of a common executable interface. For each model, a "director" determines the activation order of the components (or actors). Ptolemy has a user-friendly GUI and features a number of commonly used MoCs, including heterogenous modeling using continuous and discrete domains. However, Ptolemy lacks support for the integration of high-fidelity models of implementation platforms, which are often conveniently built using domain-specific tools.

In Metronomy, we bridge this gap by supporting co-simulation with implementation platform models developed in MetroII. To support such integration, we create a new co-simulation director *CoSimDirector* and customize the Ptolemy directors for the Discrete Event (DE), Synchronous Dataflow (SDF) and Ptides [19] MoCs.

2.2 MetroII

Metro II [5] is the successor of the Metropolis design framework [1], which implements the PBD design methodology based on a SystemC simulation engine. MetroII allows designers to import models developed using external, domain-specific tools. For example, a SystemC architectural model can be imported with only minor changes to the original model interface. Instrumental to such integration is MetroII's rigorous and general mapping semantics, which we use to bridge the functional and architectural views of a system. On the other hand, MetroII lacks the implementation of the most commonly used MoCs, and it has limited support for continuous time models. Metronomy extends the simulation core of MetroII to support co-simulation with functional models developed in Ptolemy.

3. TIMING CONTRACTS

In our co-simulation framework, a *system model* includes a higher-level *functional model*, a lower-level *architectural model* and a *mapping function*. The functional and architectural models provide two different representations of the system at different levels of abstraction, and can possibly cover different design aspects or *viewpoints*. The mapping function links how the behaviors of the functional model are mapped into behaviors of the architecture during co-simulation.

We define a *timing contract* as a tuple $\mathcal{C} = (\mathcal{E}, \mathcal{T}, \mathcal{A}, \mathcal{G})$, where $\mathcal{E}$ is a set of events, $\mathcal{T}$ is a set of time tags, $\mathcal{A}$ is a set of assumptions, and $\mathcal{G}$ is a set of guarantees. We can then denote the interface between the functional model and the architectural model by specifying a functional contract $\mathcal{C}_f = (\mathcal{E}_f, \mathcal{T}, \mathcal{A}_f, \mathcal{G}_f)$, an architectural contract $\mathcal{C}_a = (\mathcal{E}_a, \mathcal{T}, \mathcal{A}_a, \mathcal{G}_a)$, and a mapping function $\mathcal{M}$.

$\mathcal{E}_f$ is a set of events capturing the activity in the functional model, $\mathcal{E}_a$ is a set of events capturing the activity in the architectural model, $\mathcal{T}$ is a set of time tags that define a common notion of time shared by the two models. For each event e ($e \in \mathcal{E}_a$ or $e \in \mathcal{E}_f$), $t_e \in \mathcal{T}$ is its time tag.

An event in the functional model $e \in \mathcal{E}_f$ is represented by a tuple $e = (fun.id, k)$, where id is the identifier of the event, which specifies, for instance, the arrival of sensing data, the beginning or the ending of a computation process, or the application of a certain action; k is an integer index denoting the k-th instance of the event. Similarly, each event $e \in \mathcal{E}_a$ is a tuple $e = (arch.id, k)$. When there is no confusion, we will abbreviate all the events as (id, k).

$\mathcal{A}_f$ and $\mathcal{G}_f$ are, respectively, the set of assumptions made by the functional model, and the set of guarantees provided by the model under the assumptions. Following the formulation in [2], in our framework, both $\mathcal{A}_f$ and $\mathcal{G}_f$ are sets of *behaviors* over $\mathcal{E}_f$. A behavior is defined as a *trace*, i.e. a sequence of events. Sets of traces are captured using assertions including constraints on their event time tags. Similarly, $\mathcal{A}_a$ and $\mathcal{G}_a$ represent, respectively, the sets of assumptions and guarantees related to the architectural model, and can also be expressed as assertions on the time tags of the events in $\mathcal{E}_a$.

More specifically, both assumptions and guarantees can be expressed using first order logic formulas defined as follows. A linear inequality defined on a set of event time tags t_e and other variables is a formula. If α is a formula, then $\neg\alpha$ is a formula. If α_1 and α_2 are formulas, then $\alpha_1 \wedge \alpha_2$ ($\alpha_1 \vee \alpha_2$, $\alpha_1 \rightarrow \alpha_2$) is a formula. If α is a formula and x is a variable, then $\forall x, \alpha$ ($\exists x, \alpha$) is a formula.

Finally, $\mathcal{M}$ maps events in the functional model into events in the architectural model. For a pair of events $e_1 \in \mathcal{E}_f$ and $e_2 \in \mathcal{E}_a$, if $\mathcal{M}(e_1) = e_2$, then $t_{e1} = t_{e2}$.

Examples of assertions used to express assumptions and guarantees are provided below:

- End-to-end path latency:

$$\forall k, \quad t_{(p.e,k)} - t_{(p.b,k)} \leq d \qquad (1)$$

  where $(p.b, k)$ is the beginning event corresponding to the k-th computation of path p, $(p.e, k)$ is the ending event corresponding to the k-th computation of path p, and d is the deadline for the path latency.

- Periodic events:

$$\forall k, \quad t_{(id,k+1)} - t_{(id,k)} = T \qquad (2)$$

  where T is the period.

- Sporadic events:

$$\forall k, \quad t_{(id,k+1)} - t_{(id,k)} \geq T_s \qquad (3)$$

  where T_s is the minimum interval of two consecutive events with the same id.

- Partial order of events:

$$t_{e_1} < t_{e_2}, \qquad (4)$$

  which can be used to encode several type of constraints such as the amount of requested computation or data dependencies.

As an example, for a functional model, $\mathcal{A}_f$ could be (1), while $\mathcal{G}_f$ could be a conjunction of assertions in (2), (3) and (4). Since we only focus on timing assertions, we assume that the available architecture platform can implement any behavior ($\mathcal{A}_a = True$) at possibly different costs. On the other hand, $\mathcal{G}_a$ is a set of performance guarantees from the services implemented on the architecture. A typical architecture guarantee on the execution of a service could be

$$t_{(id.e,k)} - t_{(id.b,k)} \leq w \qquad (5)$$

as in (1), where w is now the execution time for service id. Note that all the parameters (e.g. w) in the assertions above are not necessarily constant but can be dynamically changed during the simulation.
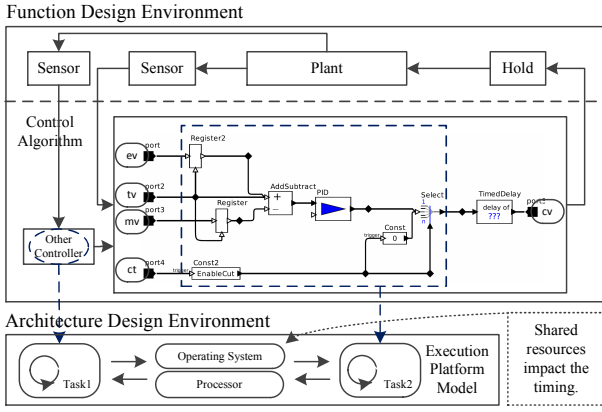
Figure 1: A simplified controller in a printing press paper feed system. Controllers in the function design environment are mapped to a single processor multi-task execution platform in the architecture design environment.

3.1 Illustrating Example

To illustrate how timing contracts can be formulated on a control system, we consider the system in Figure 1 showing a simplified controller in a paper feed subsystem of a printing press (see also Section 6.2 for further details). The controller regulates the surface velocity of a roller by adjusting the drive voltage of a motor. The controller has four inputs: tv, ev, mv, ct, and one output cv. Input tv is the profiled target velocity, ev is a real-time adjustment on the profiled target velocity based on the state of the other rollers, mv is the measured velocity of the roller, ct is a signal that turns off the motor, and cv provides the drive voltage to the motor. The controller tries to minimize the error between the measured velocity mv and the target velocity $tv + ev$. When a sporadic signal ct occurs, the controller outputs 0. p_1, p_2, p_3, p_4 are the execution paths from ev, tv, mv, ct to cv, respectively. By following the semantics of Ptolemy, each component of the functional model in Figure 1 is implemented as an actor. Strictly speaking, a path in the model consists of a cascade of a sensor actor, a series of interconnected execution actors and an actuator actor. In this example, we assume the delays of the sensor and the actuator are zero for simplicity and thus the end-to-end latency is the sum of delays due to the execution actors along the path.

We denote as $(p_i.b, k)$ and $(p_i.e, k)$ the events corresponding to the beginning and the ending of the k-th computation of path p_i. If an output is generated in the k-th computation, the output occurs at $t_{p_i.e,k}$. l represents the latency between the activation of the controller and the output. Different l may deliver different control performances.

Events $(c.b, k)$ and $(c.e, k)$ correspond to the beginning and the ending of the k-th firing of the controller, which carries on the computation of its paths. Note that one firing of the controller may carry on the computations of multiple paths. The firing of the controller is triggered by the events at the input ports. Between $(c.b, k)$ and $(c.e, k)$, there are also events indicating the beginnings or the endings of the firings of internal actors. For example, $(c.pid.b, k)$ and $(c.pid.e, k)$ indicate the beginning and the ending of one firing of the internal actor PID (Proportional-Integral-Derivative).

The assumptions of the functional contract $\mathcal{C}_f$ are specified by the conjunction of assertions on the end-to-end latencies of paths. For example,

- *End-to-end latency* of path p_2:

$$\forall k, \qquad t_{(p2.e,k)} - t_{(p2.b,k)} = l \leq d$$
$$\forall k_1, k_2, \quad t_{(p2.b,k1)} = t_{(c.b,k2)} \rightarrow t_{(p2.e,k1)} = t_{(c.e,k2)}$$

which state that all computations must complete before the deadline d, which is typically associated with the requirements from the plant.

The guarantees of the functional contract $\mathcal{C}_f$ are also specified by a conjunction of assertions. Examples of assertions are

- *Controller activation*: $\forall k, t_{(p2.b,k)} = t_{(tv,k)}$ guarantees that computation of path p_2 is triggered by signal tv and $\forall k_1, \exists k_2, t_{(p2.b,k1)} = t_{(c.b,k2)}$ guarantees that there is always a firing of the controller that carries on the computation of path p_2.

- *Periodic event tv*: $\forall k, t_{(tv,k+1)} - t_{(tv,k)} = T_{tv}$ guarantees that tv is a periodic input.

- *Sporadic event ct*: $\forall k, t_{(ct,k+1)} - t_{(ct,k)} \geq T_{ct}$ guarantees that the minimal interval of two consecutive ct is T_{ct}.

- *Amount of requested computation*:

$$\forall k, \forall j_1, j_2,$$
$$(t_{(c.b,k)} \leq t_{(c.reg.b,j_1)} \wedge t_{(c.reg.e,j_1+r_1)} \leq t_{(c.e,k)}$$
$$\wedge (t_{(c.b,k)} \leq t_{(c.reg2.b,j_2)} \wedge t_{(c.reg2.e,j_2+r_2)} \leq t_{(c.e,k)})$$
$$\rightarrow r_1 + r_2 \leq r_{c.reg}$$

where $r_{c.reg}$ specifies the bound for the number of firings of the *Register* actors (including *Register* and *Register2* in Figure 1). Similarly, $r_{c.add}$, $r_{c.con}$, $r_{c.pid}$ and $r_{c.sel}$ specify the bounds for the number of firings of each type of actor during one firing of the controller.

Finally, while the architecture assumptions are always true, the guarantees of the architectural contract $\mathcal{C}_a$ are specified by the conjunction of assertions on the execution time of services. For example, the assertion on the computation of PID would be:

$$t_{(task2.pid.e,k)} - t_{(task2.pid.b,k)} \leq w_{pid}$$

where w_{pid} is a variable depending on the processor speed and possible preemptions of $Task1$.

4. TIMING VERIFICATION AND DESIGN EXPLORATION METHODOLOGY

Metronomy can be used for timing verification as well as design space exploration. We denote the functional component (model) $Func$ as the set of all the possible traces $\{tr_{f1}, tr_{f2}, ...\}$ defining its behavior, where each trace tr_{fi} is an infinite sequence of events $(e_{fi_1}, e_{fi_2}, \ldots)$. Similarly, the architectural component $Arch$ can be seen as a set of traces. Then, the mapping function $\mathcal{M}$ corresponds to a set of rendezvous constraints on events in the two models: $t_{e_f} = t_{e_a}$ if $\mathcal{M}(e_f) = e_a$.

Given the timing contracts $\mathcal{C}_f$ and $\mathcal{C}_a$, and a system level specification in the form of a contract $\mathcal{C}_s$, the *timing verification* problem translates into checking whether $\mathcal{C}_s$ and the

composition of $\mathcal{C}_f$ and $\mathcal{C}_a$ are satisfied by the behaviors of the *mapped model Func* $\times$ *Arch*$|_{\mathcal{M}}$, i.e. the system model obtained by mapping function behaviors into architecture behaviors. Formally,

$$Func \times Arch|_{\mathcal{M}} \models \mathcal{C}_f \otimes \mathcal{C}_a$$
$$Func \times Arch|_{\mathcal{M}} \models \mathcal{C}_s \qquad (6)$$

where $Func \times Arch|_{\mathcal{M}}$ is a set of traces including both function and architecture events. Each trace is, in general, an infinite sequence of events, obtained by merging a trace in $Func$ and a trace in $Arch$ that satisfy the rendezvous constraints specified by $\mathcal{M}$. We say that a component (or a system) satisfies a contract (denoted by $\models$ in (6)) when its event time tags satisfy the guarantees in the context of the assumptions, i.e., for the functional model,

$$Func \cap \mathcal{A}_f \subseteq \mathcal{G}_f, \qquad (7)$$

where $Func \cap \mathcal{A}_f$ represent the function behaviors that satisfy the assertions of $\mathcal{A}_f$.

As in [2], the assumptions and guarantees of the composite contract $\mathcal{C}_f \otimes \mathcal{C}_a$ can be defined as follows:

$$\mathcal{G}_\otimes = \mathcal{G}_f \cap \mathcal{G}_a$$
$$\mathcal{A}_\otimes = \mathcal{A}_f \cap \mathcal{A}_a \cup \neg \mathcal{G}_\otimes,$$

where $\neg$ denotes the complement of a set, and all the assumptions and guarantees are assumed to be extended to the same set of variables including both function and architecture events, via a reverse projection operation. If (6) hold then we can also conclude that $\mathcal{C}_f$ and $\mathcal{C}_a$ are *consistent*, i.e. there exists an implementation that satisfies both contracts. In Metronomy, we check all the assertions of the composite contracts using monitors during co-simulation of the functional and the architectural models.

In addition to timing verification, we can use Metronomy to perform *design space exploration*, by using timing checkers in an optimization loop, where an objective function (or a set of objectives) is optimized. As an example, if $Func$, $Arch$, $\mathcal{C}_f$, or $\mathcal{C}_a$ are expressed in parametric form, the design space exploration problem can be formulated as follows:

$$\min_{x_f, x_a, x_c} J(Func(x_f), Arch(x_a))$$

$$\text{s.t.} \left\{ \begin{array}{l} Func(x_f) \times Arch(x_a)|_{\mathcal{M}} \models \mathcal{C}_f(x_f, x_c) \otimes \mathcal{C}_a(x_a, x_c) \\ Func(x_f) \times Arch(x_a)|_{\mathcal{M}} \models \mathcal{C}_s(x_f, x_c) \\ x_f \in X_f, x_a \in X_a, x_c \in X_c \end{array} \right.$$

where x_f and x_a are sets of parameters that encode design choices in the functional and the architectural model, respectively; x_c are parameters of the contracts (e.g. see d, $r_{c.reg}$, $r_{c.pid}$ in the illustrating example) which allow relaxing or tightening design requirements; J is cost function. The models obtained from the optimization process, $Func^*$ and $Arch^*$, can then be provided as specifications to be independently implemented (refined) by the control and the embedded system engineers.

5. METRONOMY IMPLEMENTATION

As shown in Figure 2, Metronomy model includes two components, i.e. the functional model and the architectural model. At the top level, the two parts are composite actors governed by the co-simulation director *CoSimDirector*, which extends the execution semantics of MetroII.
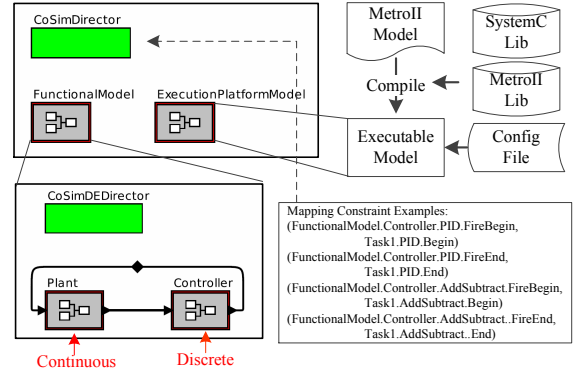

Figure 2: A CPS model in Metronomy.

The functional model is an actor-based hierarchical model inside a composite actor. For CPS, a customized discrete event director *CoSimDEDirector* is used to govern the execution of the functional model. Under *CoSimDEDirector*, the physical plant is typically modeled using directors for continuous MoCs, while the controller is typically modeled with discrete MoCs, which are all adapted to the co-simulation execution semantics. The architecture is a MetroII model, which is based on a SystemC simulation engine. The model is compiled with SystemC and MetroII libraries into an executable, which then runs in a separate process during co-simulation. The running model is wrapped by a composite actor using inter-process communication. The architectural model also has a configuration file that contains the its parameters.

The mapping function $\mathcal{M}$ is implemented as a set of mapping constraints used by the co-simulation director. A mapping constraint is a rendezvous constraint on a pair of events, where each event is specified by its name. Figure 2 also shows examples of mapping constraints, in which the beginning and the ending of firings of PID and $AddSubtract$ actors in the functional model are mapped to the beginning and the ending of the PID and $AddSubtract$ services of $Task1$ in the architecture.

5.1 Co-simulation Director

Each actor under the co-simulation director is either a functional model or an architectural model, which is a process that controls a set of concurrent processes. The simulation progress of each model is controlled by the co-simulation director via passing events.

An event $e = (id, k)$ is associated to a tuple (id, t, s, V), where id is the event identifier, $t \in \mathcal{T}$ is its time tag ($t = null$ for un-timed events), $s \in \{proposed, waiting, notified\}$ is the state of the event, and V is a set of additional values that can be used for passing messages between actors (models). Intuitively, each event marks the beginning or the ending of an activity (e.g. a computation process or the application of a certain action). When the event is passed from an actor to the co-simulation director, the state is always *proposed*. When the event is passed back from the co-simulation director to the actor, the state is either *notified* or *waiting*, indicating whether the activity associated with the event can proceed or not. Multiple events can be proposed by one actor to model the concurrency. An event being proposed indicates that an activity "may happen" in the functional or
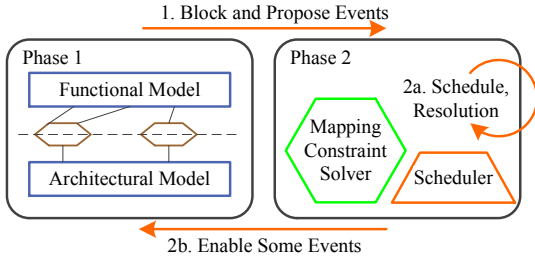
Figure 3: The two-phase execution semantics.

architectural model. Then, if the proposed event is *notified*, the associated activity will actually happens. If the proposed event is instead set to *waiting* by the co-simulation director, the associated activity will have to wait.

Figure 3 shows the execution semantics of the co-simulation director, which is articulated into two phases:

- *Phase* 1: *Base Model Execution.* Each top-level actor (functional or architectural model) executes until it blocks after proposing events. After all the top-level actors block the simulation transitions to phase 2.

- *Phase* 2: *Scheduling or Constraint Solving.* The states of the proposed events are updated based on the resolution of the mapping constraints. A subset of the proposed events are enabled and their states are updated to *notified*, which simultaneously allows their associated composite actors to resume. The rest of the events remain suspended, i.e. their states are updated to *waiting*.

Inside a composite actor, the internal actors are organized hierarchically and each actor is seen as a separate process which is scheduled by the governing MoC director as well as the co-simulation director. In phase 1, an actor has a chance to propose events only when the actor is scheduled by the governing MoC director; an actor can proceed to the phase 1 of the next round only when the proposed events are *notified* in phase 2.

5.2 Mapping Semantics

Metronomy uses rendezvous constraints and event synchronization to implement the mapping $\mathcal{M}$ between functional architectural models, a powerful and flexible mechanism, which allows mapping constraints to be established between arbitrary pairs of events.

Let e_f and e_a be two events in the functional and architectural models, respectively, and such that $e_a = \mathcal{M}(e_f)$. A rendezvous constraint on e_f and e_a requires that both of them be in the *proposed* state when the constraint is resolved in phase 2. Events e_f and e_a will then be set to *notified* only when both of them are in the *proposed* state in the same round; if only one of them is *proposed*, it will be just set to *waiting*. As an example, if we assume e_a is proposed in each round, it will only be notified when the mapped event e_f is also proposed, which implies the activity in the architectural model is "driven" by the functional model. Symmetrically, if e_f is proposed in each round, it will only be notified once e_a gets proposed, meaning that the execution of the functional model is now "driven" by the architectural model.
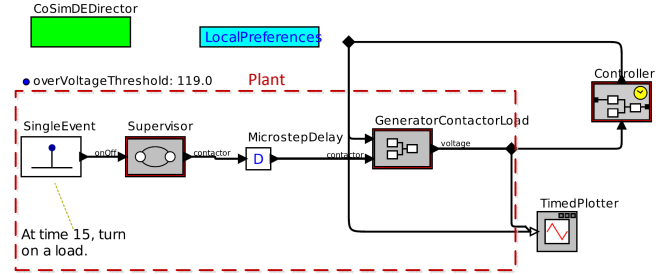

Figure 4: The functional model of a simplified electrical power system.
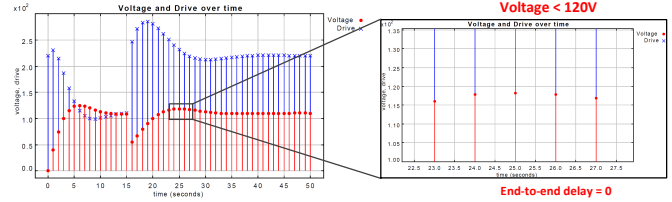

Figure 5: Simulation results from an ideal functional model with zero end-to-end latency.

6. CASE STUDY

We demonstrate our methodology and the use of our co-simulation framework on design examples of embedded controllers for an aircraft electric power system and a printing press paper feed system.

6.1 Aircraft Electric Power System

Due to the increase in electrification of modern aircraft, the design of the electrical power distribution system has become very challenging because of the safety-critical nature of the system, subject to tight reliability constraints [14]. Figure 4 shows a simplified functional model of a power system, including a composite *GeneratorContactorLoad* actor, modeling the power system plant, and a hierarchical controller, built out of two composite actors, the *Supervisor* and the *Controller*. The power system plant consists of a set of power sources (generators), loads and electromechanical switches (contactors), all lumped into the continuous-time actor *GeneratorContactorLoad*. We assume that the *Supervisor* is a fixed, pre-designed finite state machine which configures the power plant by actuating the contactors to connect the power sources to the loads in each aircraft operation mode. Our goal is to explore the trade-offs involved in the design of the *Controller* that regulates and stabilizes the amplitude of the voltage on the power network, which is required to never exceed 120 V to prevent any damage in the loads. To do so, our simulation setup includes a *SingleEvent* actor modeling the insertion of a new load in the system at time 15 s.

Figure 5 shows the result of a purely functional simulation where the ideal controller undergoes zero end-to-end latency. The controller outputs its drive voltage (represented in blue in the figure) by instantly reacting to the voltage level sensed on the power network. The requirement is always satisfied.

On the other hand, in Figure 6, we represent a more realistic, composite *Controller* actor, where blocks *Sensing-*

Comm and *ActuatingComm* capture the effects of communication between the PID controller and the plant via sensors and actuators. Sensing and actuation delays are not always available while prototyping the control algorithm at the functional level. Therefore, instead of delving into the implementation details and the specific delay breakdown between computation and communication, the control designer may conveniently rely on a simpler interface, defined by a timing contract. In such a contract, an assumption can be made on the whole end-to-end latency between sensing and actuation, which can be captured by the following assertion:

$$t_{(ActuatingComm.e,k)} - t_{(SensingComm.b,k)} \leq d, \qquad (8)$$

where d represents both the communication (e.g. bus) and computation (e.g. processor) delays related to the implementation architecture. Under the assumption in (8), the functional model guarantees that the *Controller* is triggered every 1 second: $\forall k, t_{(SensingComm.b,k)} = k$. Moreover, the amount of computation in each activation is bounded. For example, *PID* needs to compute at most once whenever *PIDController* is triggered:

$$\forall k, \forall j,$$
$$(t_{(PIDController.b,k)} \leq t_{(PIDController.PID.b,j)}$$
$$\wedge t_{(PIDController.PID.e,j+r)} \leq t_{(PIDController.e,k)}) \rightarrow (r = 0).$$

The functional model is then accompanied by an architectural model, including a processor, a sensor, an actuator, a bus and a simple OS layer that supports the following services: reading values from the sensor, writing values to the actuator, arithmetic computations and PID computation. In its contract, the architecture guarantees that the delay of each service is bounded.

To provide realistic worst case estimations of the end-to-end delays, we co-simulate the composition of the functional model with the architectural model, where firing events of *SensingComm*, *ActuatingComm* and *Controller* are mapped to "service" events in the architecture. As stated earlier, such a mapping mechanism allows accounting for the impact of architectural choices on the system functionality, while keeping the details of the architecture "hidden" from the pure functional model. We perform verification of the timing contract by checking that for each event arriving at *port2* in Figure 6 the end-to-end timing assumption is satisfied, i.e. it is discharged by the architecture guarantees. If this is not the case, a timing violation exception will be thrown.

As an example, we investigate the impact of the latency assumptions on the final controller design, a crucial parameter for the development of this system. A pessimistic latency bound d may end up with a degraded controller performance, while an optimistic bound at the functional level may require a fast and expensive architecture to be supported. In Figure 7, we prototype a controller by assuming a loose assumption on the end-to-end latency ($d = 0.3$ s). Such an assumption is compatible with an inexpensive architecture, with high (pessimistic) sensing and actuation delays, and a "slow" communication bus. However, the mapped controller violates our requirement since an overdrive voltage exceeding 120 V is observed at time 24 s and 25 s. We can then overcome this issue by either providing a "faster" bus, or modifying the functional architecture in the first place to accommodate any impairments related to the implementation platform.

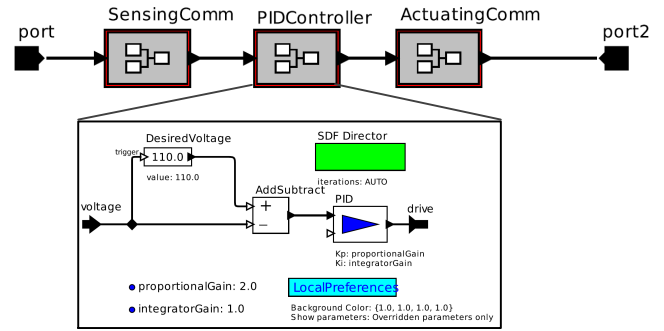The results obtained after implementing the former solu-

Figure 6: The controller in an electrical power system. *PID-Controller* is a sampled-data feedback controller. The *PID* control filter simply takes the difference between the measured voltage and the desired one.
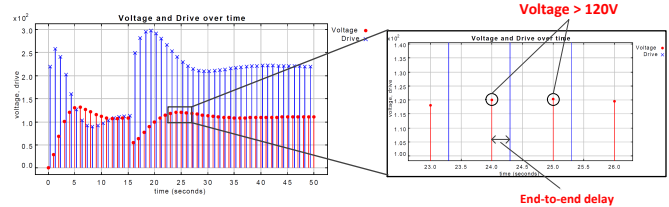

Figure 7: Simulation of the functional model and the architecture with a slow bus.

tion are visualized in Figure 8, where the functional model is now assuming $d = 0.09$ s, albeit at additional architecture-related costs. On the other hand, Figure 9 shows how the functional model can be modified when the latter approach is adopted. To avoid over-voltage problems, the functional model is equipped with an additional voltage protection mechanism for the loads. Whenever the voltage level exceeds 119 V, the voltage protection kicks in and disconnects the loads from the power network until the desired voltage is restored on the line. By utilizing the additional voltage protection, a looser bound is acceptable for the architecture contract, which allows leveraging cheaper solutions.

Figure 10 shows the simulation result of the mapped model with voltage protection. At time 24 s, the protection circuit detects that the voltage exceeds 119 V and disconnects the loads, which will be reconnected later on, when the voltage stabilizes. Different solutions result in different timing contracts ($d = 0.3$ s or $d = 0.09$ s) between function and architecture designers, which in turn restrict the further refinement of both function and architecture.

6.2 Printing Press Paper Feed System

Figure 11 shows the paper feed system of a high-speed printing press. The system consists of three types of rollers: two drive rollers, a feed roller, and a reserve roller. A roll of paper is driven by the feed roller to feed the printing machine. When the radius of the feed roll becomes lower than a first threshold, a signal is sent to bring up the reserve roller and its velocity will eventually match that of the drive roller. When the radius of the feed paper roll is lower than a second threshold, a tape detector begins sensing the tape on the reserve roll. When the presence of a strip of tape on the reserve roll is detected, the contact controller computes
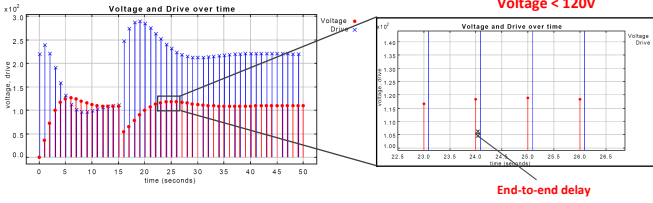
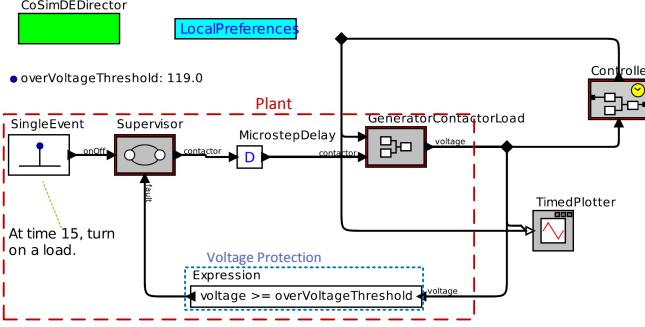Figure 8: Simulation of the functional model with accelerated architecture.

Figure 9: Functional model of an electrical power system with over-voltage protection.

Figure 10: Simulation of the functional model with over-voltage protection.

Figure 11: The paper feed subsystem.

when the strip of tape will be directly opposite the contact actuator and prior to this point of time, it sends a signal to the contact actuator, which forces contact between the active and reserve paper so that they are attached by the tape. Right after that, the cutter actuator cuts the paper from the feed roller so that the reserve roll continues to feed the printing machine. The most critical scenario for timing requirements occurs when the contact actuator reacts after the radius of the feed roll falls below the second threshold and the tape is detected.

Figure 12 shows the functional model $Func$ of the power feed subsystem; the controller consists of nine composite actors, numbered from 6 to 14. We assume that the controller senses and actuates the plant with a sampling period $T_{sample} \in \mathcal{T}$ which is a design choice, with $\mathcal{T} = \{0.02, 0.04, 0.06, 0.08, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8\}$ s. The architectural model $Arch$ is a single processor multi-task platform. To implement the mapping function $\mathcal{M}$, each of the actors 6-14 in Figure 12 is mapped to a task, nine in total, which implies that all the "begin" and "end" events of the atomic actors enclosed in the composite actors are mapped to the "begin" and "end" events of the corresponding services of the tasks. These tasks are scheduled by a priority-based operating system supporting preemption. The priorities are given from high to low in the following order: 6,7,8,12,13,14,9,10,11. The processor is connected to sensors and actuators via Ethernet. End-to-end latency measurements consist of the following contributions: sensor and actuator delay, communication delay, processor execution delay. We assume that the sensing, actuation and communication delays are constants. The frequency of the processor $f_{proc} \in \mathcal{F}$ is a design choice, with $\mathcal{F} = \{3.3, 5, 10, 16.6, 20, 33.3, 50, 100, 133\}$ MHz; faster processors are more expensive.

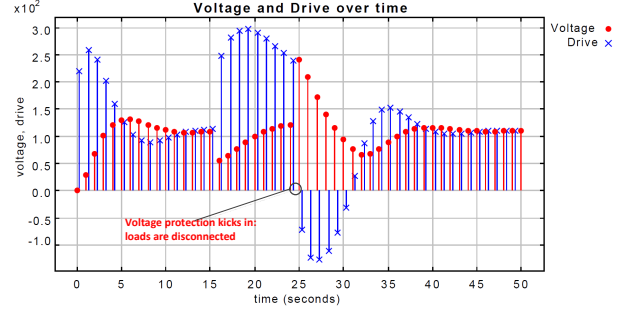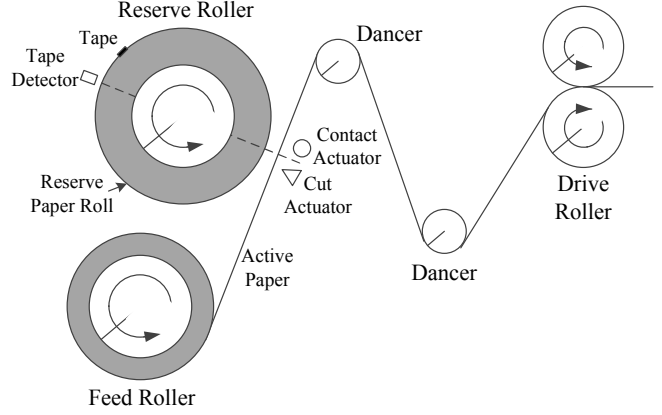A key performance metric in this system is the tracking er-

ror between the paper velocity of the feed (reserve) roller and the one of the drive roller. Although the two dancers shown in Figure 11 help compensate the difference in the velocities of the drive roller and the feed roller, a controller is still needed to minimize the tracking error, especially when the drive roller accelerates soon after the system starts. Since the radius of paper roll is also changing, the drive signal to the feed roller has to be dynamically adjusted to maintain a proper surface velocity. We add monitors to the functional model to measure the RMS tracking error ϵ_{RMS} using the following formula:

$$\epsilon_{RMS} = \sqrt{\frac{T_{mon}}{T_{sim}} \sum_{i=0}^{T_{sim}/T_{mon}} \left(V_{drive}^{iT_{mon}} - V_{feed}^{iT_{mon}} \right)^2},$$

where T_{mon} is the sampling period of the monitor, V_{drive}^t and V_{feed}^t are the velocities of the drive roller and the feed roller at time t respectively, and T_{sim} is the duration of the simulation.

We cast the design space exploration problem as a multi-objective optimization problem subject to timing contracts; we aim to minimize:

$$\min_{T_{sample}, f_{proc}} \quad (\epsilon_{RMS}, f_{proc})$$
$$\text{s.t.} \quad \begin{cases} Func(T_{sample}) \times Arch(f_{proc})|_{\mathcal{M}} \\ \models (\mathcal{A}_f, \mathcal{G}_f(T_{sample})) \otimes (True, \mathcal{G}_a(f_{proc})) \\ Func(T_{sample}) \times Arch(f_{proc})|_{\mathcal{M}} \models \mathcal{C}_s \\ T_{sample} \in \mathcal{T}, f_{proc} \in \mathcal{F} \end{cases} \quad (9)$$

where both the functional and architectural contracts have been concisely denoted as pairs of assumptions and guar-

Figure 12: The functional model of a paper-feed subsystem.
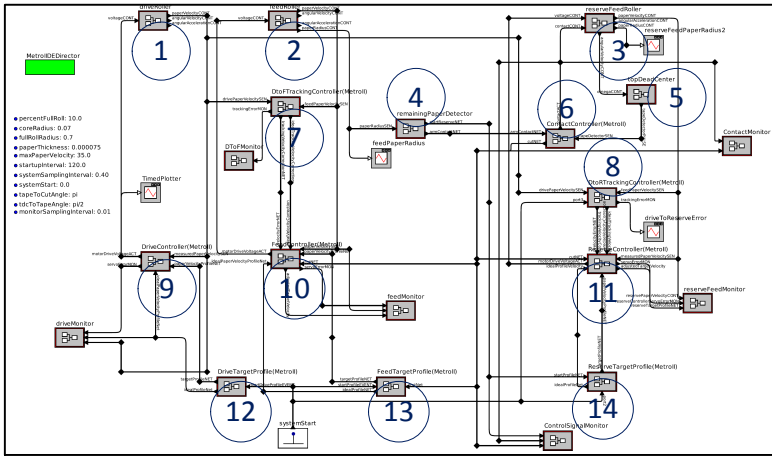
1. Drive Roller
2. Feed Roller
3. Reserve Roller
4. Remaining Paper Detector
5. Tape Detector
6. Contact Controller
7. Drive to Feed Tracking Controller
8. Drive to Reserve Tracking Controller
9. Drive Controller
10. Feed Controller
11. Reserve Controller
12. Drive Target Velocity Profile
13. Feed Target Velocity Profile
14. Reserve Target Velocity Profile
The rest are monitors.

antees, by dropping the set of events and time tags. $\mathcal{A}_f$, $\mathcal{G}_f(T_{sample})$ and $\mathcal{G}_a(f_{proc})$ are obtained by composition of the contracts of all the controllers, among which the contract for the *Feed Controller* has been illustrated as an example in Section 3.1. $\mathcal{C}_s$ is the contract that specifies the system level requirements (e.g. the timing requirements on the contact actuator and the cut actuator).

The tracking error ϵ_{RMS} depends on the sampling period T_{sample} and the end-to-end latency l of the feedback controller ($l \leq T_{sample}$). As shown in Figure 13, as the sampling period increases, the tracking error (in blue) significantly increases. In addition, the velocity of the paper roll becomes unstable when $T_{sample} \geq 0.6$. Figure 14c shows the linear velocity of the feed roller, the target velocity, and the error between them when the sampling rate is 0.8 s, which causes the system to fail.

In Figure 13, the curve in red shows the tracking error ϵ_{RMS} versus f_{proc} Pareto front. Given a processor speed f_{proc} we can find the sampling period T_{sample} that minimizes the tracking error while satisfying all the timing constraints. For example, when $f_{proc} = 10$ MHz, the optimal ϵ_{RMS} is 3.1 m/s, as obtained from the red curve in Figure 13. This corresponds to an optimal sampling period of $T_{sample} = 0.3$ s, as obtained from the blue curve in Figure 13. Any point below the ϵ_{RMS}-versus-f_{proc} Pareto front is an infeasible design due to timing violations (e.g. $f_{proc} = 10$ MHz, $T_{sample} = 0.2$ s). Any point above the Pareto front is instead non-optimal; e.g. $T_{sample} = 0.4$ s is not an optimal choice for a 10-MHz frequency since the point (10, 3.46) is dominated by (10, 3.1), obtained for $T_{sample} = 0.3$ s.

Given an architecture, we can also explore the design space at the functional level. For example, Figure 14d shows that with an economic processor (5 MHz) and a slow sampling rate (0.5 s) it is still possible to satisfy our tracking error requirement, but at the cost of a smaller roll acceleration. The compromise at the functional level enables us to use inexpensive platforms. Once that sampling period and the processor speed are decided, the functional and architectural timing contracts are also defined, and can be used for the next steps in the design process. Therefore, while the two models are kept separated, their interaction is still captured by the timing contract, which, together with co-simulation, makes it easier to verify the impact of different choices and
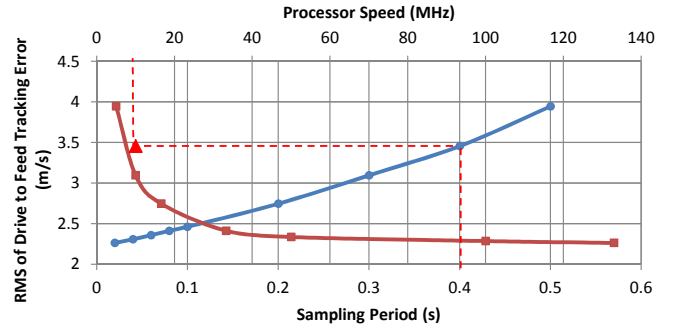

Figure 13: Design space exploration results, while minimizing both the tracking error and the processor speed.

explore trade-offs across the function/architecture boundary.

7. CONCLUSIONS

We have proposed a methodology for the verification and design space exploration of cyber-physical systems subject to real time constraints. In our framework, co-simulation of a high-level, functional model together with a lower-level, architectural model is used to *accurately* capture the effects of the implementation platform and the physical plant on the system functionality. Assume-guarantee contracts are used to *rigorously* formalize the timing requirements at different levels of abstractions and generate simulation monitors.

To support our methodology, we have implemented Metronomy, a *versatile* co-simulation framework that enables the integration of the most suitable modeling environments to capture both the functional and architectural aspects of a design. In Metronomy, the functional aspect is captured by exploiting the *variety* of models of computation made available by the Ptolemy environment and its intuitive graphical user interface. The architectural aspect is captured within the MetroII environment, capable of modeling implementation platforms to a greater level of *detail*. Models in the two environments are co-simulated based on a rigorous *mapping* semantics. The effectiveness of our approach is demonstrated on the design of embedded controllers for an
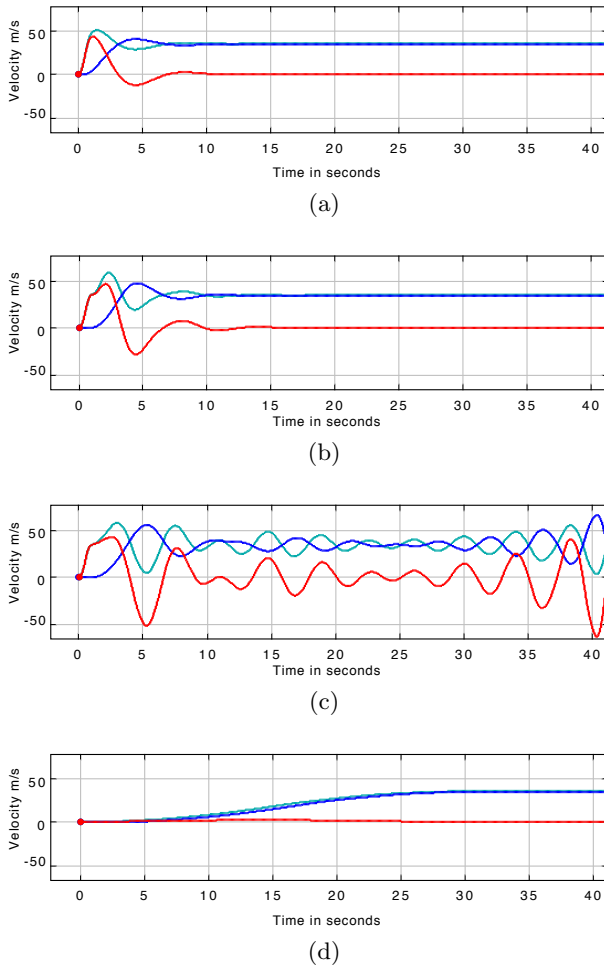
Figure 14: Simulation results for the paper feed system, including the target velocity (green) of the feed roller, the actual velocity (blue), and the error (red). (a) $T_{sample} = 0.1$ s, $f_{proc} = 33$ MHz. (b) $T_{sample} = 0.5$ s, $f_{proc} = 5$ MHz; (c) $T_{sample} = 0.8$ s, $f_{proc} = 3.3$ MHz; the velocity oscillates and the system does not operate correctly. (d) $T_{sample} = 0.5$ s, $f_{proc} = 5$ MHz, but with a slower acceleration.

aircraft electrical power system and a paper-feed sub-system of a high-speed printing press.

Acknowledgments

8. REFERENCES

[1] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: an integrated electronic system design environment. *Computer*, 36(4):45–52, 2003.

[2] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis. Formal methods for components and objects. chapter Multiple Viewpoint Contract-Based Specification and Design, pages 200–225. Springer-Verlag, Berlin, Heidelberg, 2008.

[3] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng, editors. *Heterogeneous concurrent modeling and design in Java (Volume 1: Introduction to Ptolemy II)*. Tech. rep. UCB/ERL M05/21, University of California, Berkeley, 2005.

[4] J. Castrillon, R. Leupers, and G. Ascheid. MAPS: Mapping concurrent dataflow applications to heterogeneous MPSoCs. *IEEE Transactions on Industrial Informatics*, (99):1, 2011.

[5] A. Davare, D. Densmore, L. Guo, R. Passerone, A. L. Sangiovanni-Vincentelli, A. Simalatsar, and Q. Zhu. MetroII: A design environment for cyber-physical systems. *ACM Transactions on Embedded Computing Systems*, 12(1s):49:1–49:31, 2013.

[6] L. de Alfaro and T. A. Henzinger. Interface automata. In *Proc. European Software Engineering Conference*, ESEC/FSE-9, pages 109–120, 2001.

[7] P. Derler, E. A. Lee, S. Tripakis, and M. Törngren. Cyber-physical system design contracts. In *Proc. International Conference on Cyber-Physical Systems*, pages 109–118, 2013.

[8] A. Ghosal, A. Sangiovanni-Vincentelli, C. M. Kirsch, T. A. Henzinger, and D. Iercan. A hierarchical coordination language for interacting real-time tasks. In *Proc. International Conference on Embedded Software*, pages 132–141, 2006.

[9] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Springer, 2002.

[10] T. Henzinger, C. Kirsch, M. Sanvido, and W. Pree. From control models to real-time code using Giotto. *IEEE Control Systems*, 23(1):50–64, 2003.

[11] H. Kim, L. Guo, E. A. Lee, and A. Sangiovanni-Vincentelli. A tool integration approach for architectural exploration of aircraft electric power systems. In *Proc. International Conference on Cyber-Physical Systems, Networks, and Applications*, pages 38–43, 2013.

[12] P. Lieverse, P. van der Wolf, and E. Deprettere. A trace transformation technique for communication refinement. In *Proc. International Symposium on Hardware/Software Codesign*, pages 134–139, 2001.

[13] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. Deprettere. Daedalus: toward composable multimedia MP-SoC design. In *Proc. Design Automation Conference*, pages 574–579, 2008.

[14] P. Nuzzo, H. Xu, N. Ozay, J. Finn, A. Sangiovanni-Vincentelli, R. Murray, A. Donze, and S. Seshia. A contract-based methodology for aircraft electric power system design. *IEEE Access*, 2:1–25, 2014.

[15] A. Pimentel, C. Erbas, and S. Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Transactions on Computers*, 55(2):99 – 112, 2006.

[16] C. Ptolemaeus, editor. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014.

[17] A. Sangiovanni-Vincentelli. Defining platform-based design. *EEdesign*, 2002.

[18] A. Sangiovanni-Vincentelli, W. Damm, and R. Passerone. Taming Dr. Frankenstein: Contract-based design for cyber-physical systems. *European Journal of Control*, 2012.

[19] Y. Zhao, J. Liu, and E. A. Lee. A programming model for time-synchronized distributed real-time systems. In *Proc. Real Time and Embedded Technology and Applications Symposium*, pages 259–268, 2007.