Cyber-Physical System Design Contracts

Patricia Derler
University of California, Berkeley
pd@eecs.berkeley.edu

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

Martin Törngren
KTH Royal Institute of Technology
martin@md.kth.se

Stavros Tripakis
University of California, Berkeley
stavros@eecs.berkeley.edu

ABSTRACT

This paper introduces design contracts between control and embedded software engineers for building Cyber-Physical Systems (CPS). CPS design involves a variety of disciplines mastered by teams of engineers with diverse backgrounds. Many system properties influence the design in more than one discipline. The lack of clearly defined interfaces between disciplines burdens the interaction and collaboration. We show how *design contracts* can facilitate interaction between 2 groups: control and software engineers. A design contract is an agreement on certain properties of the system. Every party specifies requirements and assumptions on the system and the environment. This contract is the central point of inter-domain communication and negotiation. Designs can evolve independently if all parties agree to a contract or designs can be modified iteratively in negotiation processes. The main challenge lies in the definition of a concise but sufficient contract. We discuss design contracts that specify timing and functionality, two important properties control and software engineers have to agree upon. Various design approaches have been established and implemented successfully to address timing and functionality. We formulate those approaches as design contracts and propose guidelines on how to choose, derive and employ them. Modeling and simulation support for the design contracts is discussed using an illustrative example.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-purpose and Application based Systems—*Real-time and embedded systems*

1. INTRODUCTION

[1] Cyber-Physical Systems (CPS) design involves various distinct disciplines such as control engineering, software engineering, mechanical engineers, network engineering, etc. The complexity and heterogeneity of all the different design aspects require methodologies for bridging the gaps between the disciplines involved. This is known to be challenging since the disciplines have different views, encompassing terminology, theories, techniques and design approaches.

In this paper, we focus on interactions between control and embedded software engineers. A multitude of modeling, analysis and synthesis techniques that deal with *codesign* of control functions and embedded software have been developed since the 1970s. We use the term codesign for approaches that provide an awareness of constraints across the disciplines such that design of different aspects can proceed in parallel.

Advances in embedded system design methods and supporting tools have led to a strong adoption of model-based design, in which systems are designed at a higher level of abstraction, followed by the generation of an implementation. This is a well-established practice in control engineering and it is also becoming more common in embedded software engineering.

Despite these advances, gaps between the control and embedded software disciplines remain. There is no clear specification of required interactions and many assumptions such as timing are still implicit. Differences in concepts and concerns between control and software engineering, some of which are mentioned in Table 1, hinder communication. For instance, performance refers to bandwidth and settling time in the control domain, and to response time and context switching time in the embedded software domain. Central concepts such as the *sampling-to-actuation* (StA) delay in control theory and the response time or execution time for embedded software, do not, by definition, correspond to the same time interval [34].

The interdisciplinary dependency of design aspects poses a major problem as decisions in one domain will affect the other. Dependencies are typically non-linear and, since they affect both the control and embedded software design, they are associated with trade-offs [2]. For instance, changing the required speed of a feed-

[1] This work was supported in part by the iCyPhy Research Center (Industrial Cyber-Physical Systems, supported by IBM and United Technologies), the NSF Expeditions in Computing project *ExCAPE: Expeditions in Computer Augmented Program Engineering*, and the Center for Hybrid and Embedded Software Systems (CHESS) at the UC Berkeley (supported by the National Science Foundation, NSF awards #0720882 (CSR-EHS: PRET) and #0931843 (ActionWebs), the Naval Research Laboratory (NRL #N0013-12-1-G015), and the following companies: Bosch, National Instruments, and Toyota). Additional support was received from the ArtistDesign network of excellence and from the Royal Institute of Technology (KTH).

The models discussed in this paper are available online and hyperlinks behind figures in this PDF link to the individual models. Ptolemy models are available in executable form using Web Start and we provide the source files for Simulink models.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICCPS'13, April 8-11, 2013, Philadelphia, PA, USA
Copyright 2013 ACM 978-1-4503-1996-6/13/04 ...$15.00.

Concepts/ Domain:	Control	Embedded software
Metrics, Constraints	Robustness, noise sensitivity, bandwidth, overshoot, settling time	Utilization, response time, memory footprint, WCET, slack, power consumption
Design parameters	Choice of strategy (PID, optimal, adaptive, etc.), noise/robustness trade-off	Task partitioning, scheduling, inter-process communication
Formalisms, Theory	ODEs, continuous-time control theory, sampled data control theory	C code, synchronous languages, scheduling theory, model-checking, task models, UML/SysML

Table 1: Concepts in control and embedded software design

back control system will have an impact on the sampling period implemented by the computer system and may also have a significant impact on the StA delay that can be tolerated. Likewise, the partitioning of executable code into tasks will, together with a triggering and scheduling scheme, have an impact on the characteristics of time delays in the control system.

As a basis for mutual understanding, it is essential to establish an explicit semantic mapping between the two domains. In this paper, we propose a framework of *design contracts* between control and embedded software engineers, with the goals of bridging the gap between the two worlds, making implicit assumptions explicit, and thus facilitating interaction and communication. Figure 1 illustrates the concept. System-level requirements and external constraints form boundary conditions for design contracts. External constraints (which could correspond to other contracts) restrict the freedom available to control and embedded software engineers. For example, the choice of processing platform bounds achievable execution speeds and thus the control performance. The closed-loop control design, including choice of control strategy and sampling periods, is ultimately constrained by the available sensing and actuation capabilities, and the characteristics of the controlled process (plant).

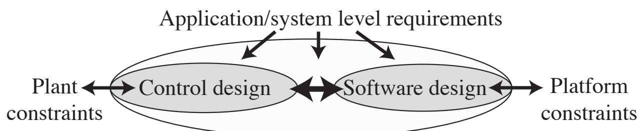

Figure 1: Design contracts are arrow between control and software design; additional arrows illustrate the context

A design contract, like any other contract, is an agreement, which entails rights and obligations to the agreeing parties. Design contracts apply this general concept to the engineers involved in the design of a CPS. Making rights and obligations explicit in the design contract provides a basis for proper CPS design and decision making. We focus in particular on the expected timing behavior and functionality of the embedded control subsystem of a CPS. The design contract states at what times certain functions must be computed. The control engineers provide the functions, typically as mathematical models, and guarantee correct behavior of the CPS under the assumption that the functions are computed at the right times. The embedded software engineers assume functional correctness and implement the functions such that the timing constraints can be met.

The elements of our design contract framework are the following: (a) concepts relevant to timing constraints and functionality; (b) design contracts utilizing these concepts; (c) a process and guidelines on how to choose, derive and employ design contracts. The guidelines encompass specific considerations for modeling and simulation support. In the rest of the paper, we present the above elements in detail. Section 2 reviews the state of the art and provides an overview of design approaches and supporting tools. Examples of formalized design contracts are given in Section 3. In Section 4 we describe guidelines for deriving and applying design contracts. Section 5 presents conclusions and future directions.

2. RELATED WORK

We first provide a broad overview of research areas related to the codesign of control systems and embedded software followed by a brief survey of work related to contracts.

2.1 Design Approaches

The research field of embedded control systems as part of a CPS was initiated already in the early 1970s, a time when computer resources were scarce and thus the problems of implementing a controller were critical. The Artist roadmaps give an overview of the evolution of the field[2]. A number of aspects that have to be considered in CPS design have been identified over the years. These include concepts that link control and embedded software, as well as trade-offs regarding memory, accuracy and speed. In this paper, we focus mainly on timing properties such as periods, StA delays and jitter. With respect to control and embedded software codesign, we identify the following strands in state of the art research.

Separation of concerns. The basic idea is to decouple control and embedded software design. Representative approaches include the synchronous programming languages [7], and the Logical Execution Time approach (LET) of Giotto [24] and its successors [29, 23]. This research also includes robust control design, which guarantees robust performance despite bounded timing variations. This enables partial decoupling as shown in [37, 22].

Optimization and synthesis of timing parameters. In this popular research strand a cost function is defined to, for example, optimize the control system performance by adequate choice of controller periods under given resource limitations. Optimization often refers to the choice of periods for controllers and processor utilization and typically only one parameter is considered. One exception is [8] which considers both periods and delays. Recent efforts in task and message scheduling in a distributed system setting include for example [15, 27, 30].

Run-time adaptation. This is another popular research direction encompassing the provisioning of online compensation as well as online optimization. While many (often legacy) computer systems cannot provide an easily characterized timing behavior, it might still be possible to provide run-time measurements of the actual timing behavior which can then be used in control system design to compensate for the imperfect timing. The topic has been studied both for wired and for wireless networks. Common approaches use extended estimators that address time-varying delays and data-loss [32, 4]. Online optimization takes online measurements/estimations of actual control system performance and/or computer system performance to adjust task attributes (e.g. periods) or schedules with the goal of optimizing control performance, see e.g. [19].

A common extension of the first two categories also considers

[2]http://www.artist-embedded.org/artist/-Roadmaps-.html

modes of operation. For each mode, a different configuration of the system (including periods and schedules) is derived during design.

A range of computational models has been proposed for all three directions, including different schemes for how to map control functionalities to task and execution strategies [2].

A complementary and more recent strand, which has been gaining interest in the past years, is the study of a theory for event-triggered feedback control. Promising initial results indicate that event-triggered control allows for a reduction of the required computations and actuations without reducing the performance of a regulator control system [12]. Event-triggered control clearly poses challenges for scheduling, motivating the need for further work on codesign.

Regardless of the design approach, explicit timing assumptions are useful, and design contracts can help with that. For instance, the use of an optimization approach will, in itself, make some assumptions about the control design and the embedded software design, and will generate a timing behavior that can be captured as a contract. In the case of run-time approaches, constraints and boundaries for adaptation will be assumed or provided as part of the design, and can thus be made explicit.

2.2 Research Related to Design Contracts

Various research ideas relate to our concept of design contract either by similarity in principle, or by similarity of name.

Contracts are an essential aspect of component-based design and interface theories [16]. In these frameworks, components are captured by their interfaces, which can be seen as abstractions that maintain the relevant information while hiding irrelevant details. Interfaces can also be seen as a contract between the component and its environment: the contract specifies assumptions on the behavior of the environment (e.g., that a certain input will never exceed a certain value) as well as the guarantees on the behavior of the component (e.g., that a certain output will never exceed a certain value).

Abstracting components in a mathematical framework that offers stepwise refinement and compositionality goes back to the work of Floyd and Hoare on proving program correctness using pre- and post-conditions [21, 25]. A pair of pre- and post-conditions can be seen as a contract for a piece of sequential code. These ideas were further developed in a number of works, including the design-by-contract paradigm implemented in the Eiffel programming language [28]. Although related, the term 'design contract' is not to be confused with 'design by contract'.

The above works focus mainly on standard software systems and as such use mainly discrete models. Nevertheless, contract-based design methods have also been studied in the context of CPS, and formalisms have been developed to deal with real-time and continuous dynamics [17, 33]. Particularly relevant to our study is the work on scheduling interfaces [1], which specify the set of all admissible schedules of a control component. Scheduling interfaces assume a time-driven scheduling context, where time slots are defined and allocated to control tasks.

All the above works are compatible with the design contract framework. In particular, some of the aforementioned formalisms could be used as concrete mechanisms for describing contracts. Nevertheless, the focus of our design contract framework is how to use contracts to solve the broader-in-scope problem of designing both the control and embedded software of a CPS. In that respect, the goals here are very much aligned with the ones presented in [31]. In particular, design contracts could be used as 'vertical contracts', using the terminology of [31]. Whereas 'horizontal contracts' that focus on the relationships of different components at the same level of abstraction are primarily used for composition, vertical contracts focus on the relationships between components at different layers, in particular, between specifications and implementations, or between high-level design requirements and execution platform constraints.

Finally, this work has been greatly inspired by the presentation of the evolution of real-time programming given in [26]. Part of our terminology, such as the terms ZET and BET, is borrowed from there.

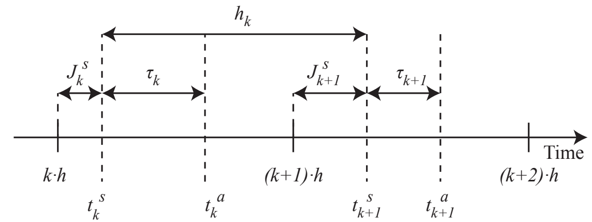

Figure 2: Timing variables in periodic control systems

3. DESIGN CONTRACTS

We now formulate some popular design approaches as design contracts. Our goal is by no means to be exhaustive, but to illustrate the concept of a design contract concretely. The design contracts presented here focus on functional and timing aspects. Some preliminary concepts are discussed first.

We use state machines of type Mealy extended with I/O interface functions to specify the functional part of a design contract. Such a machine M is characterized by: a set of input variables (*inputs*), a set of output variables (*outputs*), a set of state variables (*state*), an initialization function that initializes the state, a sampling function that reads the sensors and assigns values to the inputs, an actuation function that writes the outputs to the actuators, an output function that computes the outputs from current inputs and state, and an update function that computes a new state from current inputs and state. The sampling and actuation functions are the I/O interface functions of the machines.

Figure 2 presents relevant timing variables and corresponding notation. t_k^s and t_k^a refer to the k-th sampling and actuation instants, respectively, for $k = 0, 1, 2, \ldots$. The sampling-to-actuation (StA) delay is $\tau_k = t_k^a - t_k^s$. In periodically-sampled control systems, the nominal sampling period is h and in principle $t_k^s = k \cdot h$. In practice, however, the k-th sampling occurs at instant $t_k^s = k \cdot h + J_k^s$, where J_k^s refers to the sampling instant jitter. We assume $k \cdot h \leq t_k^s \leq t_k^a \leq (k+1) \cdot h$. Note that τ_k and J_k^s in general are non-zero and vary each period (as a convention, a symbol without subscript k refers to a constant). Finally, h_k denotes the 'effective' period, i.e., the delay between the k-th and $(k+1)$-th sampling instants: $h_k = t_{k+1}^s - t_k^s = h + J_{k+1}^s - J_k^s$.

We use the term 'contracts' instead of 'specifications' since the latter are usually viewed as being unidirectional, e.g., in the sense of being "thrown at" a team (e.g., the software engineers) by another team (e.g., the control engineers). Design contracts emphasize the importance of interaction and negotiation between the teams, and are as such bidirectional. Contracts typically include conditional, assume, or guarantee statements, separating the rights and obligations of each party involved in the contract. In the examples of contracts that we provide below, some of these obligations are left implicit. In particular, the obligations of the software engineers include meeting the timing requirements of the contract. The main

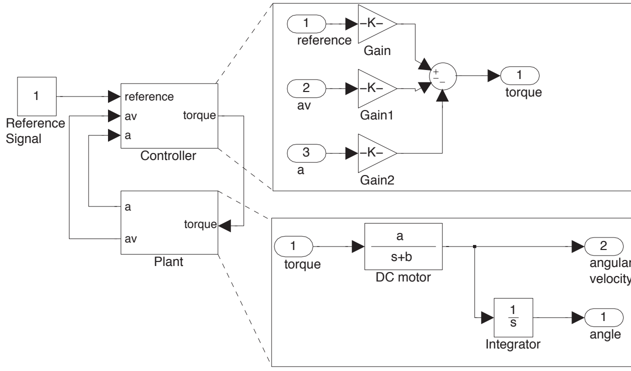

Figure 3: ZET model of DC motor control in Simulink

obligation of the control engineers is to ensure the correct behavior of the closed-loop system, provided the timing requirements are met.

3.1 The ZET (Zero Execution Time) Contract

3.1.1 ZET: single-period version

In the simple case, which we call the 'single-period version' the *Zero Execution Time* (ZET) contract is as follows:

> **ZET**: A ZET contract is specified as a tuple (M, h) where M is a state machine and h is a period of time. The contract states that, at every time $t_k^s = k \cdot h$, for $k = 0, 1, 2, ...$, the inputs to M are sampled, the outputs are computed and written (a-priori instantaneously, i.e., $\tau_k = J_k^s = 0$), and the state machine performs a state update.

How control engineers can derive a ZET contract: A typical control design process naturally results in a ZET contract. Standard results from control theory can be used that assume inputs are sampled periodically and outputs are computed and written instantaneously at the beginning of each period [5].

Figure 3 shows a Simulink[3] model of a simple CPS consisting of a plant (a DC motor) and a controller. The controller computes a requested torque (abbreviated as torque in the Figure) from the measured motor angle and angular velocity. This controller model captures a ZET contract, as it computes and writes outputs in zero time.

How software engineers can implement a ZET contract: For a control engineer, ZET is perhaps the easiest contract to produce. The opposite is true for a software engineer: it is *impossible* to implement a ZET contract in the strict sense. Computation always takes time, therefore, the outputs cannot be written at exactly the same time at which the inputs are read.[4] Moreover, some delays may be associated with the sampling, communication and actuation. Instead of trying to meet an impossible specification, software engineers typically do the next best thing: implement the functionality of the state machine M, perform worst-case execution time (WCET) analysis of the program, and make sure that the WCET obtained is at most h. In order to further minimize the StA delay, state updates and related computations can be executed after the outputs are computed and written. This practice is, however, not made part of an explicit contract.

Following the above approach, the implementation of a ZET contract results in a sequence of actions as shown here in pseudo code:

```
initialize state;
set periodic event H;
while (true) {
    await event H;
    sample inputs;
    compute and write outputs;
    compute and update state;
}
```

3.1.2 ZET – multi-periodic version

A generalization of the single-period ZET contract is a multi-periodic ZET contract, where the control engineer designs a set of machines M_i, for $i = 1, ..., n$. Each machine M_i generally needs to execute at a different period h_i and machines generally need to communicate. Block-diagram formalisms such as Simulink (c.f., Figure 3) or synchronous languages such as Lustre [9] can be used to specify the communication semantics.

Control engineers can design multi-periodic ZET contracts based on sampled data theory [5]. A common approach is to identify a basic sampling period h^0 such that all other rates are multiples of it and assume that all samplings are synchronized. The complete system is then resampled with period h^0.

Multi-periodic ZET contracts are, again, impossible to implement if taken literally. Moreover, the fact that there are multiple state machines to be executed instead of just one, and the fact that these machines communicate with each other, add complexity to the implementation problem. Some execution-platform specific approaches have been developed to deal with this complexity. A method for the single-processor platform case is described as follows. First, each state machine M_i is implemented as a separate task; i.e. a sequential program. Scheduling theory can be used to order task executions using WCET information. Although naive inter-task communication schemes do not work (i.e. do not preserve the functional ZET semantics), semantics-preserving protocols that address this problem exist [10].

3.1.3 ZET – event-triggered version

A further generalization of the ZET contract allows machines to be triggered sporadically by external triggers (i.e., events coming from the environment) or internal triggers (i.e., events sent by other machines). Assumptions on the timing of external events (e.g., how frequently they may occur) can be captured explicitly as part of the contract. Real-time scheduling theory and semantics-preserving implementation techniques can implement this type of ZET contract and in implementations constraints can be approximated.

3.2 The BET and DET (Bounded Execution Time) Contracts

The *Bounded Execution Time* (BET) contract weakens the requirement of the ZET contract that outputs must be produced at the same time as inputs are sampled. Instead, outputs can be produced at any time until the end of the period. In its single-period version, the BET contract can be stated as follows:

> **BET**: A BET contract is specified by a tuple (M, h) where M and h are as in a ZET contract. The contract

[3] http://www.mathworks.com/products/simulink/
[4] This is true in the general case, where outputs instantaneously depend on the inputs, as in Figure 3. If M is a machine of type Moore rather than Mealy, then outputs only depend on the state of the machine. In that case, a ZET contract is implementable.

states that the inputs are sampled at times $t_k^s = k \cdot h$, that the outputs are computed and are written at some point t_k^a in the interval $[t_k^s, t_{k+1}^s)$, i.e., $J_k^s = 0$ and $\tau_k < h$, and that the machine performs a state update at some point before t_{k+1}^s.

How control engineers can derive a BET contract: BET contracts are harder to derive for control engineers, because the promises made by the software engineers are weaker. In particular, the timing of the outputs is non-deterministic: the output can be written at any time within the period. This results in a time-varying control system, whereas standard control design is based upon the assumption of time-invariance. Control engineers must take this into account to make sure that the controllers they design work under all possible scenarios. Simulations, tests and analytical methods can support such reasoning [14, 22, 4, 11, 6]. For example, the Jitter-Margin approach can be used to assess the stability of a feedback control system with time-varying delays [14].

If the control performance degradation is deemed insignificant, control design can proceed using standard techniques. Otherwise, the contract must be modified or delay compensation and robustness mechanisms must be introduced. This may also involve changes of the sampling periods. Delay compensation can utilize an average delay estimations, or online estimation assuming more knowledge will be available at run-time. The final options are to renegotiate the setting for the control design, e.g. by reducing the controller bandwidth.

How software engineers can implement a BET contract: Strictly speaking, BET contracts are still impossible to implement, since it is impossible to sample inputs *precisely* at the beginning of each period. In practice, however, this is not a big concern since the sampling period is chosen with respect to the system dynamics. As long as the variations in the sampling period are small, deviations from the true values are minimal. Moreover, BET contracts are an improvement over ZET contracts from the software engineer's perspective, since the outputs have a non-zero deadline. Software engineers can use the same techniques as described above for approximating ZET contracts. BET contracts closely correspond to a scheduling scheme where a high priority task provides close to jitter-free sampling, and the scheduling guarantees that tasks complete by the end of the period.

DET - a generalization of BET with smaller deadlines

The *Deadline Execution Time* (DET) contract is a generalization of BET where the deadline can be smaller than the period:

> **DET**: A DET contract is specified by a tuple (M, h, d) where M and h are as in a BET contract and d is a deadline measured in the same time unit as the period h, such that $d < h$. The contract states that the inputs are sampled at times $t_k^s = k \cdot h$, that the outputs are computed and written at some point t_k^a in the interval $[t_k^s, t_k^s + d]$, i.e., $J_k^s = 0$ and $\tau_k \le d$, and that the machine performs a state update at some point before t_{k+1}^s.

Deriving and implementing a DET contract raises similar issues as in the case of BET. DET allows for improved control performance by reducing the delay in the control loop.

3.3 The LET (Logical Execution Time) Contract

ZET gives guarantees to the control engineer but is impossible to implement in the strict sense. BET is implementable but only provides non-deterministic guarantees, which make control design more difficult. The *Logical Execution Time* (LET) approach tries to reconcile both worlds. In this contract, outputs are written *precisely* at the end of a period. In the simple, single-period case, the LET contract can be stated as follows:

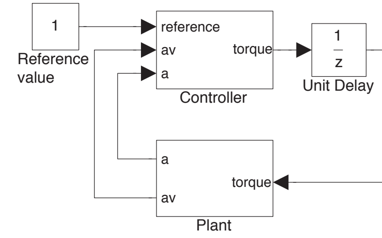

Figure 4: The LET version of the model of Figure 3

> **LET**: A LET contract is specified by a tuple (M, h) where M and h are as in a ZET contract. The LET contract states that the inputs are sampled at times $t_k^s = k \cdot h$, that the outputs are computed and the machine performs a state update at some point in the interval $[t_k^s, t_{k+1}^s)$, and that the outputs are written at $t_k^a = t_{k+1}^s = t_k^s + h$, i.e., $J_k^s = 0$ and $\tau_k = h$.

How control engineers can derive a LET contract: LET makes deterministic guarantees about the I/O timing and thus conforms to standard control theory. It is easier to use for control design than BET. However, LET introduces a one period delay which degrades the performance of the control system. This performance degradation can be partly compensated for [5].

Figure 4 shows the LET version of the Simulink model of Figure 3. The unit-delay block added between the controller and the plant provides a delay of one sampling period, causing the model to exhibit LET behavior.

How software engineers can implement a LET contract: Implementation of LET contracts is similar to the BET case. Using the techniques for BET, the software engineer can guarantee that output deadlines are met. It is (conceptually) easy to delay an early output until the end of the period.

3.4 The TOL (Timing Tolerances) Contract

Building upon the inherent robustness of feedback control systems, it is natural to introduce a relaxation of the tolerances associated with periods and delays. The *Timing Tolerances* (TOL) contract captures such relaxations. TOL can be seen as a generalization of LET with the differences that the StA delay is constant but smaller than the period, and that tolerances for allowable deviations from nominal specifications of the period and StA delay are specified:

> **TOL**: A TOL contract is specified by a tuple $(M, h, \tau, J^h, J^\tau)$ where M is a state machine, h and τ are the nominal period and StA delay, and J^h and J^τ are bounds on admissible variations of period and StA delay. All parameters are assumed to be non-negative and to satisfy the constraints $J^\tau \le \tau$ and $J^h + \tau + J^\tau < h$. The contract states that $t_k^s \in [k \cdot h, k \cdot h + J^h]$, $t_k^a \in [t_k^s + \tau - J^\tau, t_k^s + \tau + J^\tau]$, and that the k-th state update happens before t_{k+1}^s, for all k.

Figure 5 illustrates the time variables and two possible sensing and actuation times. Note that $t_k^a \in [t_k^s + \tau - J^\tau, t_k^s + \tau + J^\tau]$ is equivalent

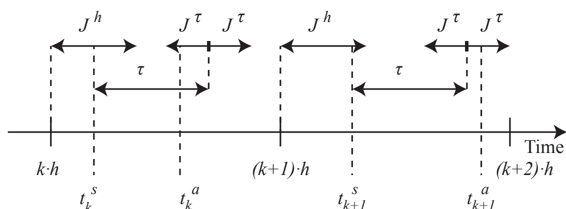

Figure 5: Timing variables of TOL contract

to $|\tau_k - \tau| \leq J^\tau$. However, $t_k^s \in [k \cdot h, k \cdot h + J^h]$ implies, but is not equal to, $|h_k - h| \leq J^h$ (this allows unbounded drift).

How control engineers can derive a TOL contract: From the viewpoint of control theory, TOL is similar to DET and BET in that it allows variations in the StA delay. In addition, TOL allows variations in the sampling period. Therefore, additional work is required to derive the tolerance parameters J^h and J^τ. This can be supported by methods such as [11, 6]. It is well-known that small variations in the sampling period and StA delay normally pose only small degradation in the control performance [37, 3]. A main intention with the TOL contract is to explicitly capture this inherent robustness. By including some type of compensation (off- or online) control engineers can tailor the robustness provided according to contract negotiations. small variations in the feedback delay can be seen to correspond to small disturbances or plant uncertainties.

How software engineers can implement a TOL contract and derive TOL parameters: Relaxed time constraints on input and output events make it easier to provide a correct schedule, e.g., taking into account blocking and interrupt disabling that could cause jitter in otherwise precisely timed interrupts. The actual scheduling is similar to approaches above.

Software engineers can contribute to the derivation of TOL parameters, such as τ or J^τ. For instance, a best case response time of the output function can be interpreted as $\tau - J^\tau$.

4. SUPPORT FOR CONTRACT BASED DESIGN

This section discusses methodological guidance for using the design contract framework. We acknowledge the individual activities of control and software designers but also the need for them to communicate, make trade-offs and agree on one or more well-defined design contracts.

4.1 A Process for Deriving Design Contracts

Figure 6 outlines the overall methodology. Compared to Figure 1, an explicit negotiation phase is added. We assume that the system-level specifications have been developed prior to the preparatory design phase. Such specifications should provide metrics and criteria for the desired system qualities such as cost, extensibility, performance, robustness and safety. The criteria can be in form of constraints (e.g. limits on utilization and closed-loop system bandwidth) or of design goals (e.g. desire to optimize certain properties).

The design phases as outlined in Figure 6 are as follows:

Preparatory design phase: Control and software engineers analyze the problem and propose overall strategies for their designs. Control engineers select a control design approach (e.g. PID, state feedback, cascaded control etc.), derive feasible sampling periods and investigate the delay sensitivity of the closed-loop system. From the software engineering side this includes investigating platform constraints, preparing I/O and communication functions and investigating possible software architectures as well as the scheduling approach.

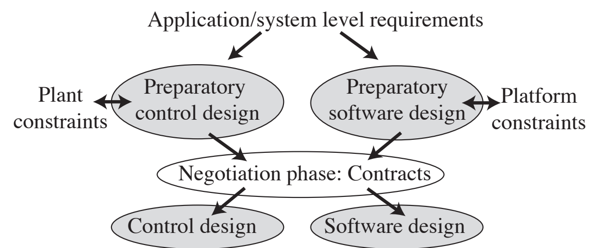

Figure 6: A process for deriving design contracts

Negotiation phase: Control and software engineers meet to investigate, propose and decide on the contracts. The considerations and trade-offs are supported by heuristics/design expertise, simulation, analytic and optimization techniques. This will, for example, help to determine optimal periods. In some cases control engineers may be able to explicitly design, or redesign, a controller to become more robust to time-variations (e.g. including delay compensation). Similarly, software engineers may be able to design or change the task scheduling. Such measures are part of the negotiation stage. There may be degrees of freedom in choosing the task and scheduling model including triggering. Options for allocation of control functionalities to tasks and computing nodes may be other design options. However, it is important to realize that such choices may impact other system-level properties, such as separation of functions of different criticality. Trade-offs are likely to be involved. The negotiation phase may obviously need several iterations. Trade-offs and optimizations may also require iterations back to the system-level requirements. The negotiation phase ends with a selection of one or more contracts with fixed contract parameters (e.g., M, h, d for DET).

Detailed disciplinary design phase: Having established the contracts enables the control and software engineers to proceed individually with the detailed design of the control and embedded software design. This also facilitates the function, subsystem and system verification, since the contracts focus the work and can be used to, for instance, generate invariants and test cases. The outputs of this phase are fed into the subsequent phases of system integration and testing which are not shown in Figure 6 for simplicity reasons.

4.2 Choosing a Design Contract

Application-specific requirements are important in determining the actual contract. A number of desired properties influence the choice, including *control performance, control robustness, system extensibility, ease of control design, ease of software design, level of determinism and resource utilization*. Many of these properties have inherent conflicts, for example control performance vs. level of determinism. Contracts which eliminate time-variations, such as LET, maximize the temporal determinism, facilitating verification and making LET contracts attractive for safety-critical applications. However, a LET contract is only applicable if the closed-loop system can tolerate the introduced delays, which generally deteriorate control performance and may even lead to instability. Standard control theory provides methods to partly compensate for known constant delays [5]. However, the inherent response delay cannot be removed and will result in inferior responses to disturbances and unsynchronized set-point changes. Using more expensive hardware to reduce the delays is one option to pave way for a LET contract. Because a LET contract is impossible to implement in the strict

Activity	Techniques and Tools
Simulation	Simulink, Ptolemy, Truetime, SystemC , ...
Static analysis	Stability and performance analysis (e.g. Matlab toolboxes, Jitterbug); Schedulability/timing analysis (e.g. SymtaVision, MAST); Hybrid systems analysis (e.g. Hytech, d/dt, SpaceEx)
Synthesis	Code generation (e.g. Targetlink); Scheduler synthesis (e.g., TTA, Giotto, Rubus)
Testing	Rapid control prototyping; SW development environments; SIL and HIL (sw/hw in the loop)

Table 2: Support for control and embedded system design

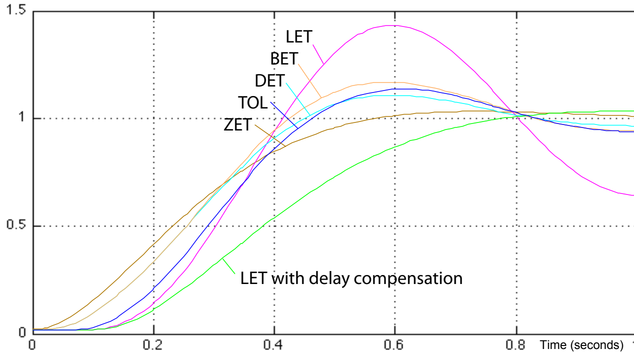

Figure 7: Control performance of contracts on the DC motor example

sense, it is reasonable to extend it to a TOL contract which is explicit about admissible time-variations.

The sensitivity of a feedback control system to (time-varying) delays can vary substantially. Among other things, it depends on the bandwidth of the controlled system, the type of variation and average delay to sampling period ratio. However, control performance generally gains from reducing the delays, and even varying delays are normally better compared to longer constant delays (see for example [13]). Performance-critical applications generally benefit from DET or BET contracts.

Table 2 lists techniques and tools that support the investigation and negotiation of contracts. Industrial techniques are mainly centered on testing and simulation.

Figure 7 illustrates the behavior of a closed-loop system with controllers obeying different design contracts. The results have been obtained from a Simulink model, with the controllers modeled as described in Section 4.3. The same value for the period parameter is used in all the controllers: $h = 0.1$. The plant is the simplified DC motor model shown in Figure 3 with requested torque as input and angular velocity and angle as outputs.[5] We observe the following:

ZET: The ZET controller provides a good response to the set point change due to zero delay and sampling jitter.

BET and DET: In both cases there is a time-varying StA delay which is uniformly distributed in $[0,h)$ for BET and in $[0,h/2]$ for DET. The average delay thus is $h/2$ for BET and $h/4$ for DET resulting in a worse response compared to the ZET controller. Given the smaller average delay, the response for DET is slightly better than that for BET.

LET with and without delay compensation: The LET controller has a StA delay of h. In the given example, this results in large oscillations and unsatisfactory performance. The LET controller can be redesigned to include compensation for the (known) constant delay. In that case, oscillations are cancelled but the inherent response delay is not removed. The response with such a controller is also shown in Figure 7.

TOL without delay compensation: the StA delay and tolerance parameters are set as follows: $\tau = h/2$, $J^h = 0.1h$ and $J^\tau = 0.4\tau$. In this case, the periods and delays are defined to vary randomly with a uniform distribution, although other variations may also be relevant to investigate (further discussed below). It can be seen that the behavior is similar to that of the BET and DET simulations.

In contracts such as DET and TOL, time-variations must be investigated as a basis for choosing the contract. The actual variations could be based on knowledge of the actual scheduling scheme and/or reasoning about relevant variations. The evaluations should consider average (e.g. uniform distribution) and assumed worst-case scenarios (e.g. StA delays and sampling periods alternating between max and min values).

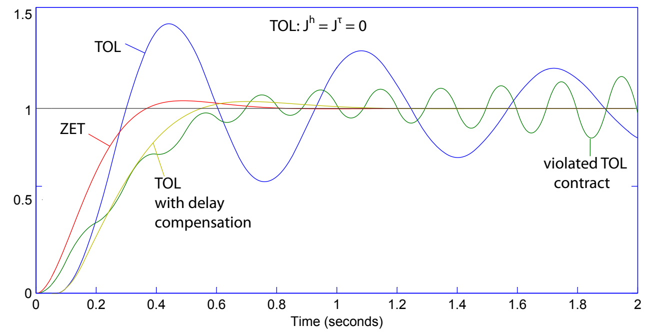

Figure 8: Performance of the DC motor example under various contracts, including illustration of contract violation

Figure 8 studies the effects of constant delay compensation in TOL, using a ZET system as a reference. For all plots $J^\tau = J^h = 0$, $h = 0.1$ and $\tau = 0.07$ (nominal StA delay for TOL in this case). The plot illustrates that StA delay compensation is necessary since the closed loop system is close to instability. The plot also illustrates a controller based on TOL, with compensation for $\tau = 0.07$, but where the actual delay is much smaller (actually set to zero). In that case, the system becomes unstable, emphasizing the importance of establishing contracts (e.g. for reuse) and in complying with agreed contracts. Even small delays can thus cause problems if they deviate from what has been assumed.

4.3 Modeling and Simulation Support

Modeling and simulation are important tools in designing CPS and we expect these tools to play an essential role in the design contracts framework. As a proof of concept, we experimented with two modeling and simulation environments: Simulink (and associated toolboxes) by the Mathworks, and Ptolemy [20]. We investigate how easy it is to model various design contracts in these tools. In particular we focus on TOL, which is the most general among the concrete contracts described in Section 3. ZET, LET, etc., can be modeled as special cases of TOL.

A possible conceptual model of a control system that uses the

[5] The plant transfer function, from requested torque to angular velocity, is given by $G(s) = 11.8/(s+1.2)$. The control design is based on optimal controller design (resulting in a controller using state feedback) and a discretized plant model. The controllers use gains $[2.2, 0.5]$ and$[1.6, 0.46, 0.06]$. (the latter for the controller with delay compensation.)

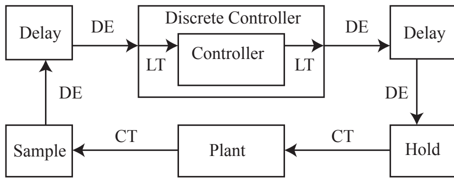

Figure 9: Control system model with TOL contract

TOL contract is shown in Figure 9. The model includes a *Plant* which operates on a continuous-time (CT) domain, taking CT signals as inputs and producing CT signals as outputs. The remaining components in the model aim to capture the TOL contract. The *Controller* contains the functionality, i.e., the state machine M: it can be modeled in a *logical time (LT)* domain, where input and output signals are ordered sequences of values, without a quantitative notion of time. A 'wrapper' converts the Controller into a component operating on discrete event (DE) signals, which are sequences of time-stamped events. The *Delay* component operates on the DE domain while *Sample* and *Hold* interface between CT and DE. Actuation values from Controller to Plant are converted from DE to CT signals via the Hold block which keeps the value constant until it receives a new event. The Sample and Delay components govern the variables t_k^s and t_k^a which can vary according to the TOL contract. The Sample, Delay and Hold components may also model parts of the sampling and actuation functions of M. For instance, quantization effects during sensing can be captured in the Sample block, which may convert floating point values to fixed point values.

Figure 9 illustrates a modular style of modeling where the relative delays caused by both delay blocks can be added without changing the control system behavior. This style might be advantageous because the delays represent different phenomena, such as communication vs. processor scheduling.

Capturing a conceptual model such as the one of Figure 9 in a concrete tool raises the following concerns:

Heterogeneity: being able to seamlessly compose models with different syntactic or semantic domains, e.g., CT, DE and LT models. Ptolemy supports many domains and achieves interoperability via a notion of 'abstract semantics' [20]. In Simulink, all blocks operate in the CT domain. Nevertheless, DE and LT signals can also be captured as piecewise-constant CT signals. A need for more explicit support was recognized by the Mathworks and various extensions to Simulink (such as SimEvents for DE) were implemented.

Mechanisms: for capturing plant behavior such as time variations or data loss. The tool must offer mechanisms for instrumenting time-variations into simulation models, as illustrated by the (time-varying) Sample and Delay blocks in Figure 9. In our experiments we use common components offered by many tools such as fixed and variable delay blocks as well as random number generators. In Simulink we also used triggered subsystems and SimEvents. Care is needed in aligning the mechanisms with the operation of the simulation environment.

Level of abstraction: at which phenomena such as time delays are modeled. These can be modeled at one extreme as full-blown architecture/execution platform models, or at the other extreme as basic delay blocks with only a couple of parameters. These parameters could be derived in different ways, e.g. based on worst-case assumptions, measurements or by simulating the embedded software and platform at some suitable level of abstraction. The structure of the conceptual model of Figure 9 allows to tailor the level of abstraction by making the Delay and other blocks arbitrarily complex or simple. Both simulation environments also offer rich opportunities for tailoring the level of abstraction, with for example SimEvents and Truetime [13] for Simulink, and quantity managers and integration with Metropolis for Ptolemy [18]. We use worst-case assumptions, random variations and more elaborate scheduling models captured with the above tools.

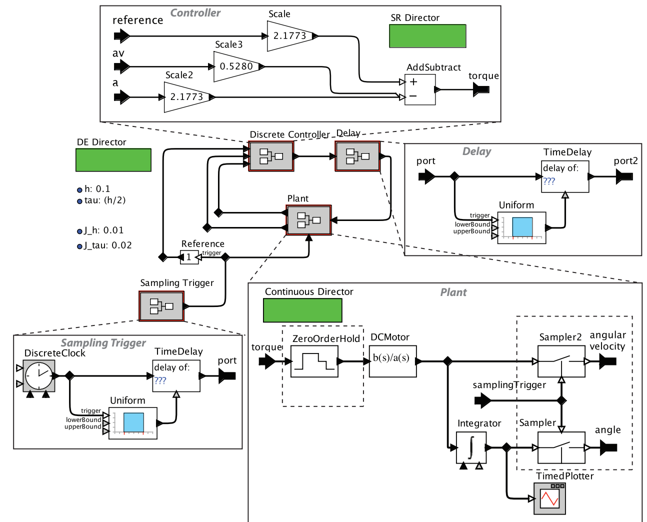

Figure 10: DC motor model with TOL contract in Ptolemy

In the rest of this section we address the above concerns in experiments using Ptolemy and Simulink.

4.3.1 Modeling and Simulating Design Contracts in Ptolemy

Figure 10 shows the TOL contract for the DC motor example described in Section 4.2 modeled in Ptolemy. Ptolemy allows for hierarchical composition of different models of computation (MoCs) such as DE, CT, synchronous reactive (SR), dataflow and others [20]. MoCs are specified by *directors*, which implement the abstract semantics mentioned above. A single MoC is allowed at any given hierarchy level. In particular, the designer must choose the top-level director (and corresponding MoC). The choice is influenced by various factors. For instance, with CT as the top level, the solver might choose a small step size which leads to slow simulation speed.

The top level model of Figure 10 is a DE model. It contains 5 actors, 4 of which are *composite* actors, meaning they are refined into sub-models (potentially described in other MoCs). The composite actors are: the plant, the controller, a delay between controller and plant and a sampling trigger block. A constant actor with value 1 is used as the reference value for the controller. The plant is a CT model. Sampler and Hold actors inside the plant actor (highlighted by dotted lines) translate between DE and CT signals. In Ptolemy, these actors have to be inside the CT MoC. Sampling and actuation functions (AD/DA converters) are not explicitly modeled here.

The controller is implemented as a synchronous reactive (SR) model which roughly corresponds to the LT domain described above. The controller is embedded in the DE domain and thus computes the control output every time one or more events are received on the inputs. Absence of events is treated as value 0 which leads to incorrect control outputs. Therefore all input sources must be triggered at the same time which is achieved by the sampling trigger.

The timing described by the TOL contract is implemented in the Sampling Trigger and the Delay actor. These actors are im-

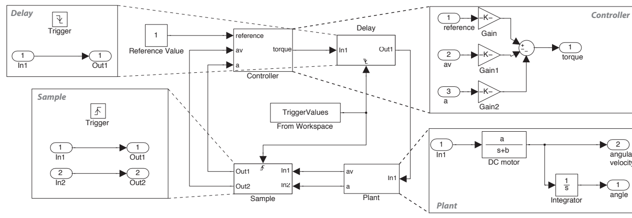

Figure 11: DC motor model with TOL contract in Simulink

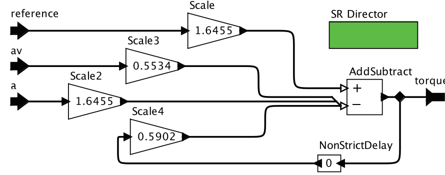

Figure 12: Controller with delay compensation in Ptolemy

plemented in the DE domain. The clock in the Sampling Trigger actor creates a periodic signal with period h, which is delayed by a random amount of time between 0 and J^h, implemented by the Uniform and the TimeDelay actors. The Delay actor between the controller and the plant implements the delay of the control signal for $\tau \pm J^\tau$ time units (here $\tau = h/2$, $J^\tau = 0.02$).

TOL is a general contract, therefore, being able to model TOL implies being able to model special cases such as ZET, LET, etc. This can be done by eliminating some random actors as well as using zero or fixed instead of variable delays.

4.3.2 *Modeling and Simulating Design Contracts in Simulink*

The TOL contract in Simulink is shown in Figure 11. All signals are CT signals. Discrete-time (DT) signals are modeled as piecewise constant CT signals. The model of Figure 11 does not use such special toolboxes such as SimEvents to model events. Instead, events are encoded in the rising and/or falling edges of DT signals and is used here to capture times t_k^s and t_k^a.

In the Simulink example, triggers are defined off-line in the Matlab workspace as vectors and imported in the model by the block label "From Workspace" which produces signals with rising and falling edges. By triggering subsystems with these signals time-varying samplers and StA delays are modeled.

4.3.3 *Modeling Delay Compensation in Ptolemy and Simulink*

The TOL contract can be extended to provide delay compensation. If the StA delay τ is known, the state machine M can be modified in order to improve the control performance (the gains have to be recomputed to accommodate this change). For the DC motor example, the computed control signal is also an input to the controller (i.e., the controller now has state).

Figure 12 illustrates a Ptolemy model of the controller presented above improved with delay compensation.

Modeling delay compensation in Simulink is not as straightforward. With varying sample times (i.e. the controller is not executed periodically if $J^h > 0$ or $J^\tau > 0$, or both), we also need varying sample times for the controller output. Control signal and values from plant to controller must be sampled at the same time. An implementation of the delay compensation is presented in Figure 13. The output of the controller is memorized and sampled by the sampling

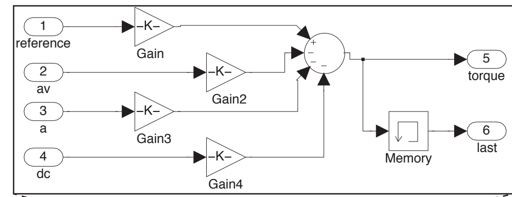

Figure 13: Modeling delay compensation in Simulink

block.The implementation of the delay compensation in Simulink involves more than just a change to the model of the controller, and is therefore less modular than that of Ptolemy. See [35] for details.

5. CONCLUSIONS AND FUTURE WORK

We proposed a framework of design contracts which relies on explicit negotiation and agreement of functionality and timing assumptions between the control and embedded software engineering teams. We believe it is essential to establish an explicit semantic mapping between the two domains to avoid potential misunderstandings, and this need will continue to grow along with the increasing scale and impact of CPS.

Future directions are numerous and include the investigation of further formalisms and languages to concretely capture design contracts. Potential candidates can be found in Section 2, but that list is by no means exhaustive. The formalization of existing contract frameworks needs to be further developed into complete contract 'algebras', e.g., along the lines of [16, 31]. In particular, compositionality of contracts is a challenging problem. For instance, a compositional formal framework for ZET contracts exists albeit with a restricted form of feedback composition [36], while the compositionality of LET contracts is unclear [26]. Apart from functionality and timing, contracts must be developed for other aspects, such as performance, reliability, failures and application modes.

Finally, we barely scratched the surface regarding questions such as: how to choose a type of contract for a given application? (a design-space exploration problem); given the type, how to fix the parameters of the contract? (a synthesis problem); how to verify the control design given a contract? (a verification problem); how to derive software implementations from contracts, ideally automatically? (a problem suite involving platform-space exploration, mapping, model transformations and compiler optimizations).

6. REFERENCES

[1] R. Alur and G. Weiss. Regular specifications of resource requirements for embedded control software. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 159–168, 2008.

[2] Artist. Artist roadmaps. `http://www.artist-embedded.org/artist/`

-Roadmaps-.html.

[3] Artist. Artist2 roadmap on real-time techniques in control. http://www.artist-embedded.org/artist/ARTIST-2-Roadmap-on-Real-Time.html.

[4] K. Årzén, A. Bicchi, S. Hailes, K. Johansson, and J. Lygeros. On the design and control of wireless networked embedded systems. In *Proceedings of the 2006 IEEE Computer Aided Control Systems Design Symposium*, Oct. 2006.

[5] K. Aström and B. Wittenmark. *Computer-Controlled Systems: Theory and Design*. Prentice-Hall, 3rd edition, 1996.

[6] I. Bates, A. Cervin, and P. Nightingale. Establishing timing requirements and control attributes for control loops in real-time systems. In *ECRTS*, 2003.

[7] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proc. IEEE*, 91(1):64–83, Jan. 2003.

[8] E. Bini and A. Cervin. Delay-aware period assignment in control systems. In *RTSS'08*, Washington, DC, USA, 2008. IEEE.

[9] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre: a declarative language for programming synchronous systems. In *14th ACM Symp. POPL*. ACM, 1987.

[10] P. Caspi, N. Scaife, C. Sofronis, and S. Tripakis. Semantics-Preserving Multitask Implementation of Synchronous Programs. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(2):1–40, Feb. 2008.

[11] A. Cervin, K. Årzén, D. Henriksson, M. Lluesma Camps, P. Balbastre, I. Ripoll, and A. Crespo. Control loop timing analysis using TrueTime and Jitterbug. In *Proceedings of the 2006 IEEE Computer Aided Control Systems Design Symposium*, Oct. 2006.

[12] A. Cervin and K. Åström. On limit cycles in event-based control systems. In *46th IEEE Conference on Decision and Control*, New Orleans, LA, 2007.

[13] A. Cervin and J. Eker. Control-scheduling codesign of real-time systems: The control server approach. *Journal of Embedded Computing*, 1(2):209 – 224, 2005.

[14] A. Cervin, B. Lincoln, J. Eker, K. Årzén, and G. Buttazzo. The jitter margin and its application in the design of real-time control systems. In *RTCSA*, Goeteborg, Sweden, Aug. 2004.

[15] A. Davare, Q. Zhu, M. D. Natale, C. Pinello, S. Kanajan, and A. Sangiovanni-Vincentelli. Period optimization for hard real-time distributed automotive systems. In *DAC*, pages 278–283. IEEE, 2007.

[16] L. de Alfaro and T. Henzinger. Interface theories for component-based design. In *EMSOFT'01*. Springer, LNCS 2211, 2001.

[17] L. de Alfaro, T. A. Henzinger, and M. I. A. Stoelinga. Timed interfaces. In *EMSOFT'02: 2nd Intl. Workshop on Embedded Software*, LNCS, pages 108–122. Springer, 2002.

[18] P. Derler, E. A. Lee, and A. Sangiovanni-Vincentelli. Modeling cyber-physical systems. *Proceedings of the IEEE (special issue on CPS)*, 100(1):13 – 28, January 2012.

[19] J. Eker, P. Hagander, and K.-E. Årzén. A feedback scheduler for real-time controller tasks. *Control Engineering Practice*, 8(12), 2000.

[20] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE*, 91(2):127–144, 2003.

[21] R. Floyd. Assigning meanings to programs. In *In. Proc. Symp. on Appl. Math. 19*, pages 19–32. American Mathematical Society, 1967.

[22] P. Garcia, P. Castillo, R. Lozano, and P. Albertos. Robustness with respect to delay uncertainties of a predictor-observer based discrete-time controller. *Proceedings of the 45th IEEE CDC*, pages 199–204, 2006.

[23] A. Ghosal, A. Sangiovanni-Vincentelli, C. M. Kirsch, T. A. Henzinger, and D. Iercan. A hierarchical coordination language for interacting real-time tasks. In *EMSOFT'06*, pages 132–141. ACM, 2006.

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

[25] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–580, 1969.

[26] C. M. Kirsch and R. Sengupta. The evolution of real-time programming. In *Handbook of Real-Time and Embedded Systems*, 2006.

[27] R. Majumdar, I. Saha, and M. Zamani. Performance-aware scheduler synthesis for control systems. In *EMSOFT'11*. ACM, 2011.

[28] B. Meyer. Applying "Design by Contract". *Computer*, 25(10), 1992.

[29] W. Pree and J. Templ. Modeling with the Timing Definition Language (TDL). In *Automotive Software Workshop San Diego (ASWSD 2006) on Model-Driven Development of Reliable Automotive Services*, 2006.

[30] S. Samii, A. Cervin, P. Eles, and Z. Peng. Integrated scheduling and synthesis of control applications on distributed embedded systems. In *Design, Automation and Test in Europe*, DATE'09, 2009.

[31] A. Sangiovanni-Vincentelli, W. Damm, and R. Passerone. Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems. *European Journal of Control*, 2012. In press.

[32] B. Sinopoli, L. Schenato, M. Franceschetti, K. Poolla, M. I. Jordan, and S. S. Sastry. Kalman filtering with intermittent observations. *IEEE Transactions on Automatic Control*, 49:1453–1464, 2004.

[33] X. Sun, P. Nuzzo, C.-C. Wu, and A. Sangiovanni-Vincentelli. Contract-based system-level composition of analog circuits. In *DAC'09*, pages 605–610. ACM, 2009.

[34] M. Törngren. Fundamentals of implementing real-time control applications in distributed computer systems. *Real-Time Systems*, 14, 1998.

[35] M. Törngren, S. Tripakis, P. Derler, and E. A. Lee. Design contracts for cyber-physical systems: Making timing assumptions explicit. Technical Report UCB/EECS-2012-191, EECS Dept., UC Berkeley, Aug 2012. http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-191.html.

[36] S. Tripakis, B. Lickly, T. A. Henzinger, and E. A. Lee. A theory of synchronous relational interfaces. *ACM Trans. on Progr. Lang. and Sys. (TOPLAS)*, 33(4), July 2011.

[37] B. Wittenmark, J. Nilsson, and M. Törngren. Timing problems in real-time control systems. In *In Proceedings of the American Control Conference*, pages 2000–2004, 1995.