Design Contracts for Cyber-Physical Systems: Making Timing Assumptions Explicit

Martin Toerngren
Stavros Tripakis
Patricia Derler
Edward A. Lee

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2012-191
http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-191.html

August 21, 2012

Design Contracts for Cyber-Physical Systems: Making Timing Assumptions Explicit

Martin Törngren
KTH
martin@md.kth.se

Stavros Tripakis Patricia Derler Edward Lee
University of California, Berkeley
{stavros,pd,eal}@eecs.berkeley.edu

August 21, 2012

Abstract

Building Cyber-Physical Systems (CPS) involves several teams of engineers with different backgrounds. We focus on interactions between control engineers and embedded software engineers. Lack of rigorous methodologies for communication and interaction between the two groups has been recognized as an obstacle to the development of dependable and cost-effective CPS. We advocate the use of *design contracts* as a step towards bridging this gap. Design contracts act as a medium for communication and interaction between the control and computation/communication design problems in a CPS. The contracts provide a focal point for negotiation and for decision making. Once design contracts have been established, they help to separate the global CPS design problem into two sub-problems, each of which can be tackled separately by the control-engineering and the embedded software teams, thus making the design more tractable and efficient. We propose a framework of design contracts encompassing (i) concepts relevant to timing constraints and functionality; (ii) a formulation of some popular design approaches as design contracts; and (iii) a process and guidelines on how to choose, derive and employ design contracts. The guidelines place specific emphasis on modeling and simulation support.

1 Introduction

Cyber-Physical Systems (CPS) have become present in all aspects of our lives and the increasing connectivity through networking, novel sensors and cost-price performance are driving new types of applications. Building CPS involves several teams of engineers with different backgrounds such as mechanical, software, electronics and control engineers. Successful CPS design requires bridging the gaps between the disciplines involved. This is known to be challenging since the disciplines have different worldviews, encompassing terminology, theories, techniques and design approaches.

In this paper, we focus on interactions between control engineers and embedded software engineers. A multitude of modeling, analysis and synthesis techniques that deal with codesign[1] of control functions and embedded software have been developed since the 1970s. 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 on the control engineering side and is also becoming more common on the embedded software side, for example, through schedule synthesis, see e.g. [2].

Despite these advances, gaps between the control and embedded software disciplines remain. It is commonly the case that timing and other assumptions are still implicit and there is room to improve the awareness of the required interactions. We believe this to be true both in academia and for practicing engineers.

[1] We will use the term codesign to refer to design approaches that, at a minimum, provide an awareness of constraints across the disciplines such that control and embedded software design can proceed in parallel. The term is sometimes used to refer to optimization approaches only.

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: Example concepts and concerns in control and embedded software design.

Some of the differences in concepts and concerns between control and software engineering are exemplified in Table 1. For instance, performance refers to concepts such as bandwidth and settling time in the control domain, and to response time and context switching time in the embedded software domain. Moreover, 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 [50].

The fundamental challenge is that many design aspects, including those mentioned in Table 1, will be dependent across the two domains. A decision in one domain will affect the other. Also, the dependencies are typically non-linear and, since they affect both the control and embedded software design, they are associated with trade-offs (see e.g. [3]). For example, changing the required speed/bandwidth of a feedback control system will have an impact on the sampling period required to be realized by the computer system and may also have a significant impact on the StA delay that can be tolerated by the closed-loop system. Likewise, the choice of the partitioning of executable code into tasks will, together with a triggering and scheduling scheme, have an impact on the characteristics of the 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 together with external constraints form boundary conditions for the design contracts. External constraints (which could correspond to other contracts) restrict the degrees of freedom available to control and embedded software engineers. For example, the choice of processing platform bounds achievable execution speeds and thus the closed-loop 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 (the plant).

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. Each group makes certain promises to the other groups, and expects something from the other groups in return. Making these elements explicit in the design contract provides a basis for proper CPS design and decision making. In this paper, we focus in particular on the expected timing behavior and functionality of the embedded control subsystem of a CPS. The design contract states the times at which certain functions must be computed by the embedded software infrastructure. The control engineers expect that these functions are computed at the right time or during the right intervals, obeying the rules specified in the contract. The embedded software engineers expect that the CPS will behave correctly as a whole, provided the above timing constraints are 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. Preliminary concepts used in the design contracts are provided in Section 3. Examples of formalized design contracts are

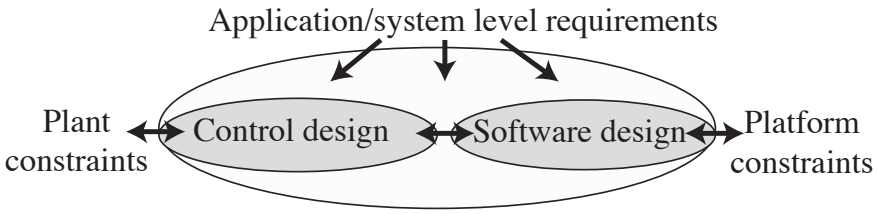


Figure 1: Design contracts illustrated as the arrow between control and software design; additional arrows illustrate the context of these design contracts.

given in Section 4. In Section 5 we describe guidelines for deriving and applying design contracts. Section 6 presents conclusions and discusses future work directions.

2 Related work

We first provide a broad overview of research areas related to the codesign of control systems and embedded software. We then provide 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 (e.g. [52, 53]). Here, 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 [9] (corresponding to the ZET contract – see Section 4.1), and the Logical Execution Time approach (LET) of Giotto [31] and its successors [39, 29, 28]. As part of this approach, we also consider robust control design, which makes the control system robust to some degree of imperfection in the implemented timing behavior introduced by computation and communication, and thus enabling partial decoupling. See for example [53, 27, 49].

Optimization and synthesis of timing parameters. This is a popular research strand, where the idea is to define a cost function that can be used, for example, to optimize the control system performance of one or more controllers by adequate choice of controller periods, and using available limited resources as constraints. See [43] for early work in this area. The optimization often refers to the choice of periods for controllers and processor utilization. It is rare that more than one parameter is considered in this work. One exception is [10] which considers both periods and delays. Recent efforts encompassing task and message scheduling in a distributed system setting include [19, 34, 41, 54, 30].

Run-time adaptation. This is another popular research direction encompassing the provisioning of online compensation as well as online optimization. The idea with online controller compensation is that while the computer system (e.g. due to legacy parts) cannot provide an easily characterized timing behavior, it might still be possible to provide run-time measurements of the actual timing behavior. The control system can then be designed to use this online information and compensate for the imperfect timing. The topic has been studied both for wired and for wireless networks. A common proposed approach is the use of extended estimators that address time-varying delays and data-loss [45, 5]. Online optimization uses online resource managers that take 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. [24, 17, 4].

A common extension for the first two categories is to also consider modes of operation. For each mode, a different configuration of the system (including periods and schedules) is derived during design. For all three categories, a range of computational models have been proposed, including different schemes for how to map control functionalities to task and execution strategies (see [2] for an overview).

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 [14]. Event-triggered control clearly poses challenges for scheduling, motivating the need for further work on codesign.

Regardless of the design approach taken, it is beneficial to make the timing assumptions explicit, and design contracts can be used for this purpose. 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, after completion, 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

Over the years, a number of proposals have been made that 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 [20]. 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 the assumptions that the component makes on the behavior of the environment (e.g., that a certain input will never exceed a certain value) as well as the guarantees that the component provides (e.g., that a certain output will never exceed a certain value).

Abstracting components in some mathematical framework that offers stepwise refinement and compositionality guarantees is an idea that goes back to the work of Floyd and Hoare on proving program correctness using pre- and post-condititions [26, 32]. 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[2] paradigm implemented in the Eiffel programming language [35].

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 a number of formalisms have been developed to deal with real-time and continuous dynamics aspects (e.g., see [21, 48, 8]). Particularly relevant to our study is the work on scheduling interfaces [1], which are used to specify the set of all admissible schedules of a control component. Scheduling interfaces assume a time-driven scheduling context, where time is divided in slots, and slots need to be allocated to different control tasks. LET is used to specify timing within a slot.

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 [42]. In particular, design contracts could be used as 'vertical contracts', using the terminology of [42]. Whereas 'horizontal contracts' focus on the relationships of different components at the same level of abstraction, and 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, work has been greatly inspired by the presentation of the evolution of real-time programming given in [33]. In particular, part of our terminology, such as the terms ZET and BET, is borrowed from there.

[2] Although related, the term 'design contract' is not to be confused with 'design by contract'.

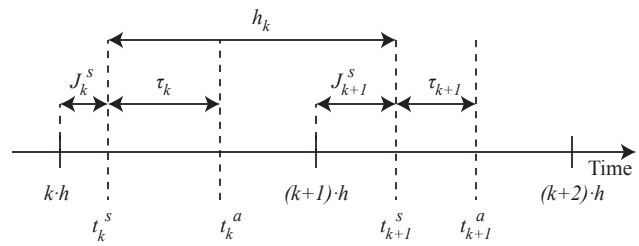


Figure 2: Timing variables in periodic control systems.

3 Preliminaries

The design contracts presented in Section 4 focus on functional and timing aspects. Let us first discuss here some preliminary concepts related to these aspects.

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.

Regarding timing, Figure 2 presents relevant 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.[3] 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: $h_k = t_{k+1}^s - t_k^s = h + J_{k+1}^s - J_k^s$.

4 Design Contracts

In this section we 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. We briefly discuss, for each design contract, how it can be derived and implemented. We conclude the section by listing a number of other approaches which could also be formalized as design contracts.

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). Instead, design contracts emphasize the importance of interaction and negotiation between the teams, and are as such bidirectional. Contracts typically include conditional, or assume/guarantee types of 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 obligation of the control engineers is to ensure the correct behavior of the closed-loop system (provided the timing requirements are met).

[3] For simplicity, clock drift is neglected in this formalization. We will also generally assume that $k \cdot h \leq t_k^s \leq t_k^a \leq (k+1) \cdot h$.

4.1 The ZET (Zero Execution Time) Contract

4.1.1 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, measured in some time unit, e.g., seconds. 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 machine performs a state update.

How control engineers can derive a ZET contract: A typical control design process naturally results in a ZET contract. Control engineers can use standard results from control theory which assume that inputs are sampled periodically and outputs are computed and written instantaneously at the beginning of each period [6].

Figure 3 shows a Simulink[4] model of a simple CPS consisting of a plant (a DC motor) and a controller. The controller uses sensors to measure the motor angle and angular velocity, and computes a requested torque (abbreviated as torque in the Figure) as controller output. This example controller has no state. This controller model captures a ZET contract, as it computes and writes outputs in zero time.

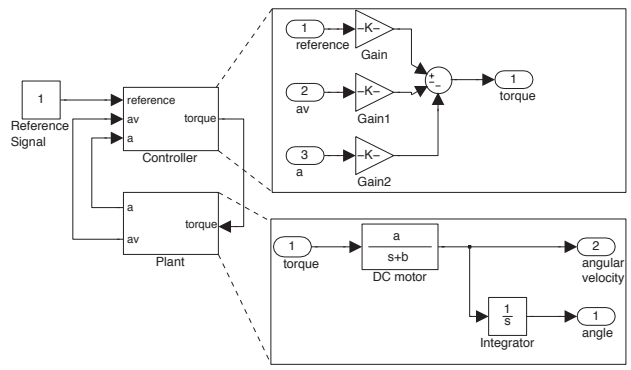


Figure 3: A ZET model of a controlled DC motor in Simulink.

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 not only hard, it is generally *impossible* to implement a ZET contract in the strict sense. This is because computation always takes time, therefore, the outputs cannot be written at exactly the same time when the inputs are read.[5] 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 in a given programming language (say, in C), 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. It should be noted that this practice is, however, not made part of an explicit contract.

Following the above approach, the implementation of a ZET contract results in the following sequence of actions (pseudo code):

[4] http://www.mathworks.com/products/simulink/

[5] 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.

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

4.1.2 ZET – multi-periodic version

A generalization of the single-period ZET contract is a multi-periodic ZET contract, where, instead of a single machine M, 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. The machines generally also need to communicate. Block-diagram formalisms such as Simulink (c.f., Figure 3) or synchronous languages such as Lustre [**?**] can be used to specify the communication semantics. Also note that a single set of I/O interface functions is needed in this case, for the external inputs and outputs

Control engineers can design multi-periodic ZET contracts based on sampled data theory [6]. There are several approaches and settings for multirate control system design. A common approach is to identify a basic sampling period h^0 such that all other rates are integer 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. Different approaches have been developed to deal with this complexity, depending on the target execution platform. In the case of a single-processor platform, one method that can be used is the following. First, each state machine M_i can be implemented as a separate task. Each task is a sequential program following the pseudo-code template presented for the single-period case above. Individual WCETs for each task can be computed. Real-time scheduling theory, e.g., rate-monotonic analysis (RMA), can be used to schedule the tasks. Note that although naive inter-task communication schemes do not work, in the sense that they do not preserve the functional ZET semantics, semantics-preserving protocols that address this problem exist [12].

4.1.3 ZET – event-triggered version

A further generalization of the ZET contract is to lift the restriction that the triggering of each machine M_i occurs periodically and to allow machines to be triggered sporadically, according to 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. Again, real-time scheduling theory and semantics-preserving implementation techniques can be used to implement this type of ZET contract. The zero-time constraint is again impossible to meet, but can be approximated.

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

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 much weaker. In particular, the timing of the outputs is non-deterministic: the output can be written to the actuator 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 be used to support such reasoning [37, 16, 27, 36, 5, 13, 7, 47]. For example, the JitterMargin approach can be used to assess the stability of a feedback control system with time-varying delays [16].

If the control performance degradation is deemed insignificant, control design can proceed using standard techniques. Otherwise, the options are to modify the contract allowing reduction of the delays, or to introduce delay compensation and robustness, which may also involve changes of the sampling periods. Delay compensation could be done for an average delay if such a delay can be estimated, or using 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 potentially reducing the required speed of the closed loop system.

How software engineers can implement a BET contract: Strictly speaking, BET contracts are still impossible to implement, since it is impossible to guarantee that inputs are sampled *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; thus as long as the variations in the sampling period are small they will also imply small deviations from the true values. Moreover, BET contracts are an improvement over ZET contracts from the software engineer's perspective, since the outputs have a non-zero deadline, which is feasible to meet. To implement BET contracts, software engineers can use the same techniques as those described above for approximating ZET contracts. We can note that the BET contract closely corresponds to a scheduling scheme where a high priority task provides close to jitter-free sampling, and where the scheduling guarantees that tasks complete by the end of the period.

DET - a generalization of BET with deadlines smaller than periods

The *Deadline Execution Time* (DET) contract is a simple 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 $\tau_k \leq d$, 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 those raised in the case of BET. From the point of view of control performance, DET is preferable compared to BET since it makes it possible to reduce the delay in the control loop. Reducing the delays requires specific consideration of deadlines less than the period and makes it more important to separate the output from the state update functions.

4.3 The LET (Logical Execution Time) Contract

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

> **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 timing of inputs and outputs, therefore, it conforms to standard control theory and is easier to use for control design than BET. A main problem with LET is that it introduces a delay of one period that degrades the performance of the feedback control system. This performance degradation can be partly compensated for [6]. The specific requirements for the control application at hand will determine if this is sufficient.

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.

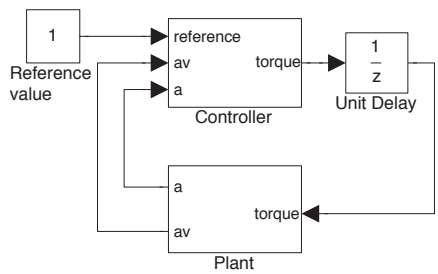


Figure 4: The LET version of the model of Figure 3.

How software engineers can implement a LET contract: LET contracts are not significantly more difficult to implement than BET contracts. Using the techniques for BET, the software engineer can guarantee that output deadlines are met. Then, it is conceptually easy to delay an early output until the end of the period.

4.4 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 a constant StA delay smaller than the period is used, 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 the admissible variations of period and StA delay respectively. All parameters are assumed to be non-negative and to satisfy the constraints $J^\tau \leq \tau$ and $J^h + \tau + J^\tau < h$. The contract states that $t^s_k \in [k \cdot h, k \cdot h + J^h]$, $t^a_k \in [t^s_k + \tau - J^\tau, t^s_k + \tau + J^\tau]$, and that the k-th state update happens before t^s_{k+1}, for all k.

Figure 5 illustrates the time variables and two possible sensing and actuation times. Note that the constraint $t^a_k \in [t^s_k + \tau - J^\tau, t^s_k + \tau + J^\tau]$ is equivalent to $|\tau_k - \tau| \leq J^\tau$. On the other hand, the constraint $t^s_k \in [k \cdot h, k \cdot h + J^h]$ implies, but is not equivalent to, $|h_k - h| \leq J^h$. Indeed, $|h_k - h| \leq J^h$ allows unbounded drift, whereas the TOL contract as formulated above does not.

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 [37, 13, 7]. It is well-known that small variations in the sampling period and StA delay normally pose only small degradation in the control performance [53, 3]. A main intention with the TOL contract is to explicitly capture this inherent robustness. By including some type of compensation (off-line or on-line) it is possible for control engineers to tailor the robustness provided according to contract negotiations.

How software engineers can implement a TOL contract and derive TOL parameters: Since the input and output events have relaxed timing constraints, it becomes easier to provide a schedule that meets the

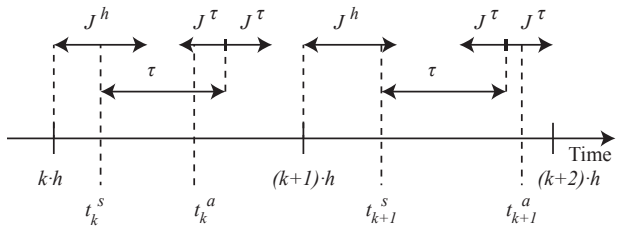


Figure 5: Timing variables of TOL contract.

constraints, e.g., taking into account blocking and interrupt disabling that could cause jitter in otherwise precisely timed interrupts. The actual scheduling is otherwise similar to the previous approaches.

Apart from implementing a contract, software engineers can also contribute to the derivation of TOL parameters, such as τ, J^τ, etc. (In fact, this is the case not just for TOL but for all contracts, which are the result of negotiation – c.f. Section 5.1.) For instance, a best case response time of the output function (if it can be determined), can be interpreted as $\tau - J^\tau$.

4.5 Other Contracts

Many variations of the above described contracts are possible. For example, it is possible to define an extended DET contract that includes tolerances on the sampling period jitter, and a new variant of the TOL contract could be provided by expressing the delay to be composed of a fixed StA delay together with an additional time-varying component.

Contracts could also be associated with the use of specific control and/or scheduling approaches. For example, a contract may specify the use of a control server [15]. The control server prescribes periodic sampling and a server-based scheduling scheme which given utilization and execution times determines a fixed StA delay and a fixed period. Cost-functions are used as part of the approach to tune the control performance vs. system utilization.

Other examples of contracts include the following.

Multimode control systems. The above mentioned contracts can be extended to multimode systems by providing subcontracts per mode and by specifying constraints and timing, for mode switches.

Periodic controllers in asynchronous distributed systems. The above introduced contracts scale to distributed settings, as long as the sampler is periodic and the output, update and output functions are event-triggered (triggered when data becomes available). However, in a multirate setting, or even for a single loop, where at least two of the parts are periodic and execute on non-synchronized computers, another type of contract is needed. Let us consider a single feedback loop, with sampling in computer A and all other functions in computer B, both executing periodically with period h. For each computer, a contract will specify the assumptions with respect to sampling. For the feedback loop, the StA delay depends on the communication delay and the actual phasing between the sampling instants of the two computers. This may result in samples arriving from A just after the start of the period of B, causing an additional time delay of h. Such considerations are part of designing the contract, see e.g. [5, 50].

Event-triggered and dynamic deadlines. Although most of the example contracts described above are based on a periodic-sampling setting, "event-triggered contracts" could also be specified. Moreover, the deadlines on the StA delay need not be statically defined, as in the examples above, but could be dynamic, e.g., the result of computation, as is the case in the Ptides model [**?**].

Probabilistic delays and data loss. The presented contracts (BET, DET, TOL) only provide bounds on maximum deviations. In some cases, probabilistic models of such variations may be attainable and can be made part of contracts. A typical case were this can make sense are delays caused by wireless communication. Apart from delays, especially wireless communication networks in addition have a higher probability for data-loss. Assumptions on probabilities or worst-case scenarios for data-loss could be formalized as part of contracts, see for example [5, 45, 38, 37].

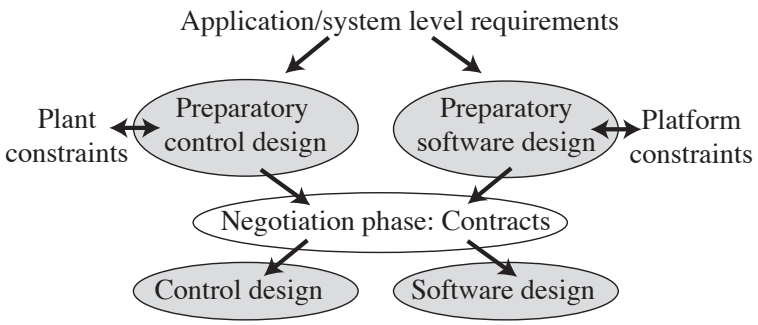


Figure 6: A process for deriving design contracts.

Deterministically varying delays and period. In some systems where time-variations are present, these variations may vary in a predictable way. One example of this is where the delays are due to pre-computed off-line schedules, where the schedule is periodic over the hyper-period (least common multiple of periods) but may vary during the hyper-period; the pre-computed schedule may also be connected to a codesign optimization procedure which results in schedule(s) [40]. Such explicit and deterministic variations can be made part of a contract.

Overruns. While a contract states a nominal situation, errors can occur not only in communication (data-loss) but also for computations. If a computation fails to complete in time, there is a need to decide on what to do; e.g., skip the cycle, provide more time, and if the problem persists, take some other action. Such considerations could be made part of contracts.

Outage. Outage refers to a situation when a computer control system suffers a complete (or partial) failure, and, for example, needs to restart. This is a well-known concept in fault-tolerant computing. A complete control system failure will in essence result in an open-loop system, in which the plant will be evolving according to its dynamics (possibly with the last control signal still being actuated) until the control system has recovered. Outage introduces "hard deadlines" [44] on recovery. From a design-contract viewpoint, the worst case is of interest, which requires consideration of critical situations (e.g., braking system outage when braking a car at high speed).

Anytime or imprecise computations. Several computational models in this category have been proposed. They all consider algorithms which refine their results where it is possible to make a distinction between a mandatory output computation and an optional one. If the latter one is allowed to execute and to complete one iteration, the results can be used to improve the output results. Examples in this category include model-predictive control which involves online optimization of a cost function in every sample, [3].

5 Support for contract based design

In this section we discuss and provide methodological guidance for using the design contract framework.

5.1 A Process for Deriving Design Contracts

In our approach, 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.

Figure 6 outlines the overall methodology. Compared to Figure 1, an explicit negotiation phase has been 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 the form of constraints (e.g. in terms of limits on desired properties such as utilization and closed-loop system bandwidth) or in terms of design goals (e.g. desire to optimize certain properties).

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

Preparatory design phase: In this phase, control and software engineers analyse the problem and propose overall strategies for their designs. From the control engineering side this includes selecting a control design approach including structure (e.g. PID, state feedback, cascaded control etc.), deriving a range of feasible sampling periods and investigating overall 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 including overall scheduling approach (static, priority based, hierarchical, etc.).

Negotiation phase: During this phase, control and software engineers meet to investigate, propose and decide on the use of one or more contracts. The considerations and trade-offs are supported by use of techniques such as heuristics/design expertise, modeling and simulation, analytic techniques and through optimization (for example to determine optimal periods).

In some cases it may be possible for the control engineers to explicitly design, or redesign, a controller to become more robust to time-variations (e.g. including delay compensation), and similarly for the software engineers to design/change the task scheduling. Such measures are part of the negotiation stage. For embedded software engineers, 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 also be in scope as design options. However, it is important to realize that such choices may also impact other system-level properties, such as separation of functions of different criticality, thus trade-offs are likely to be involved. The negotiation phase may obviously need several iterations. The trade-offs and optimizations may also require iterations back to the system-level requirements. The negotiation phase ends with an agreement to select 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 respectively. Having agreed upon contracts should also largely facilitate the function, subsystem and system verification since the contracts focus the work and can be used, e.g., to 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.

5.2 Choosing a Design Contract

Application-specific requirements play an important role in determining the actual contract. A number of desired properties influence the choice of contract, 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 will only be possible 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 [6]. 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. A LET contract is also impossible to implement in the strict sense, it is therefore reasonable to extend it to a TOL-type contract which is explicit about admissible time-variations and allows StA delays smaller than the period but still (nominally) constant.

The sensitivity of a feedback control system to (time-varying) delays can vary substantially and among other things 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 [3, 15]).[6] Because of this, performance-critical applications generally benefit from DET or BET style contracts.

[6] A well-known practice is to minimize the delay in the feedback loop by separating the necessary state updates and performing them after computing and writing a new control output.

Activity	Techniques and Tools
Simulation	Simulink, Ptolemy, Truetime, MetroII, 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 (software/hardware in the loop)

Table 2: Examples of techniques and tools supporting simulation and other activities in the control and embedded system domains (e.g., see [3, 22, 11, 46, 18]).

Developers have a range of techniques and tools at hand to support the investigation and negotiation of different contracts, e.g., see Table 2. Industrial techniques are mainly centered on testing and simulation. Modeling and simulation techniques are discussed further in Section 5.3.

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 5.3.2. (Similar results are obtained from a Ptolemy[7] model, c.f. Figure 12 of Section 5.3.1.) 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.[8] We observe the following:

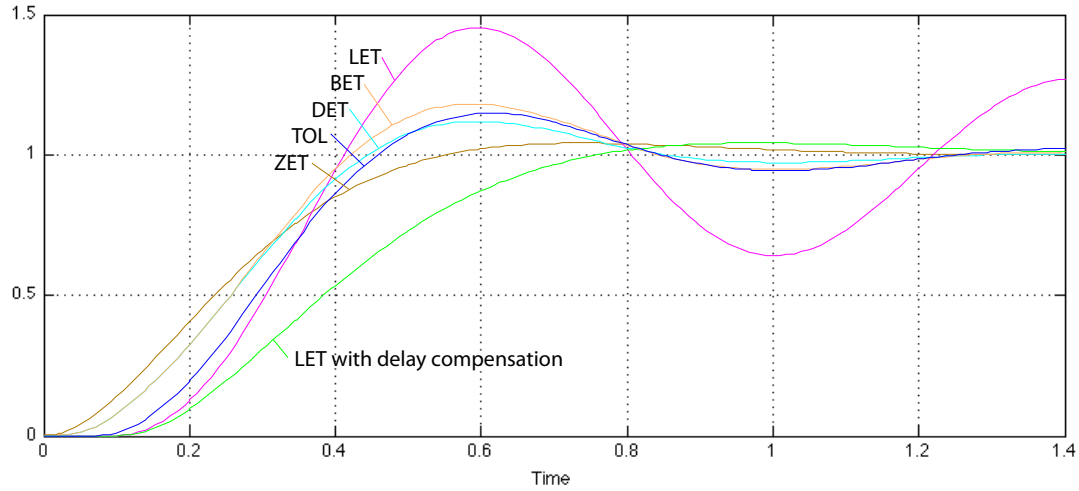


Figure 7: Control performance of various contracts on the DC motor example modeled in Simulink.

- 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; in our simulation the delay is uniformly distributed in $[0, h)$ for BET and in $[0, h/2]$ for DET. The average delay will thus be $h/2$ for BET and $h/4$ for DET resulting in a somewhat worse response compared to the ZET controller. Given the smaller average delay, the response for DET is slightly better than that for BET.

[7] http://ptolemy.org/

[8] 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 LET controller with delay compensation).

- 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, the 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.

For contracts with time-variations, such as DET and TOL, there is a need to investigate different time-variations as a basis for choosing the contract. The actual time variations to consider 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. variations where the StA delays and sampling periods alternate between maximum and minimum values).

Figure 8 study such simulations for deriving the tolerances of TOL. The simulations use the same DC motor model as before. However, the speed (bandwidth) of the closed loop system has been slightly increased to further illustrate the effects of the time variations.[9] The time variations are in the range of up to 40% of the nominal duration (period, StA).

- Figure 8(a) depicts a simulation where $J^\tau = 0, \tau = 0.06, h = 0.1$, and where J^h varies randomly with uniform distribution, such that h takes on values in the range [0.061..0.139]. This implies that actuation will always take place before the end of the period.

- Figure 8(b) depicts a simulation where $J^h = 0, \tau = 0.06, h = 0.1$, and where J^τ varies randomly, with uniform distribution, such that τ takes on values in the range [0.025..0.095].

- Figure 8(c) depicts a simulation where $\tau = 0.06, h = 0.1$, and where J^h and J^τ both vary randomly with uniform distribution in the range [0..0.019] such that h and τ take on values in the range [0.081..0.119] and [0.041..0.079] respectively.

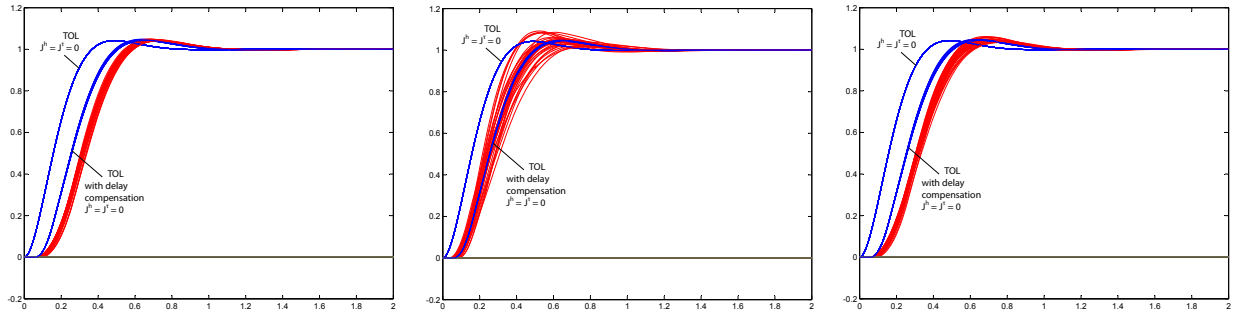
The simulations shown in Figure 8 are also relevant for a negotiation situation where LET is not considered as acceptable for control performance reasons (neither without nor with delay compensation). It can then be relevant to investigate delay compensation with TOL (assuming that controller redesign is possible).

Figure 9 further 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 compared to the uncompensated system 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). It is seen that (iii) actually 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.

5.3 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 contract framework as well. As a proof of concept, we experimented with two modeling and simulation environments, Simulink (and associated toolboxes) by the Mathworks, and Ptolemy [25], investigating how easy it would be to model a control system that uses various design contracts in each

[9] Like before the control design is based on optimal controller design (resulting in a controller using state feedback) and a discretized plant model, but with a higher closed-loop system bandwidth. The ZET and TOL controllers without delay compensation use gains $[3.97, 0.74]$ and the TOL controller with delay compensation uses gains $[3.09, 0.8, 0.6]$.



(a) Variation of J^h (b) Variation of J^τ (c) Variation of J^h and J^τ

Figure 8: Control performance on the DC motor example with a TOL contract and variations of J^h and J^τ.

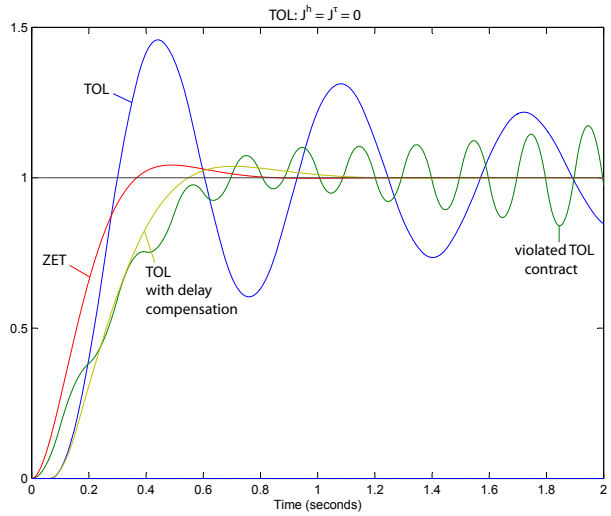


Figure 9: Performance of the DC motor example under various contracts, including illustration of contract violation.

of these tools. Here we focus the discussion on how to model TOL, which is the most general among the concrete contracts described in Section 4 (thus ZET, LET, etc., can also be modeled as special cases).

One possible conceptual model of a control system that uses the TOL contract is shown in Figure 10. 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* captures 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.[10] A 'wrapper' is used to convert the Controller into a component operating on discrete event (DE) signals, which are sequences of time-stamped events. The *Sample* component operates on the DE domain while components *Delay* and *Hold* operate on the CT and DE domain, and in effect interface between these domains. 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 both can vary according to the TOL contract. The Sample, Delay and Hold components may also be used to 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

[10] Languages such as the synchronous languages can be seen as operating in the LT domain. Note that LT is not to be confused with LET.

values to fixed point.

Figure 10 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.

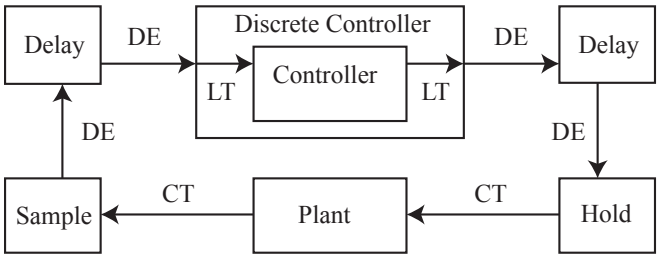


Figure 10: A conceptual model of a control system using the TOL contract.

Capturing a conceptual model such as the one of Figure 10 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' [25]. 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 as a result Simulink was extended with libraries and functionality for other domains. For example, SimEvents supports the DE domain.

Mechanisms: for capturing time variations, data loss etc. In order to assess the impact of using different contracts such as TOL, there is a need for basic mechanisms for instrumenting time-variations into simulation models, as illustrated by the (time-varying) Sample and Delay blocks in Figure 10. In our experiments we made use of fixed and variable delay blocks and random number generators to simulate time delay variation, time-varying sampling, and data loss. In Simulink we also used triggered subsystems and SimEvents. Our experiments show that care is needed in aligning the mechanisms with the detailed operation of the simulation environment.

Level of abstraction: at which phenomena such as time delays are modeled. Such phenomena are the result of complex behaviors and 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 10 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 [15] for Simulink, and quantity managers and integration with Metropolis for Ptolemy [23]. We used worst-case assumptions, random variations as well as more elaborate scheduling models captured with the above tools.

In the rest of this section we discuss in further detail the experiments we performed using Ptolemy and Simulink, showing how we addressed the above concerns.

5.3.1 Modeling and Simulating Design Contracts in Ptolemy

Figure 11 shows the DC motor example described in Section 5.2 modeled in Ptolemy II [25]. The goal is to illustrate how the TOL contract can be modeled in Ptolemy. Ptolemy allows for hierarchical composition of different models of computation (MoCs). Examples of MoCs are DE (discrete-event), CT (continuous-time), synchronous reactive (SR), dataflow and others. 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 might be influenced

by various factors. For instance, when using CT as the top level, the step size of the simulation chosen by the solver might be very small and thus lead to slow simulation speed.

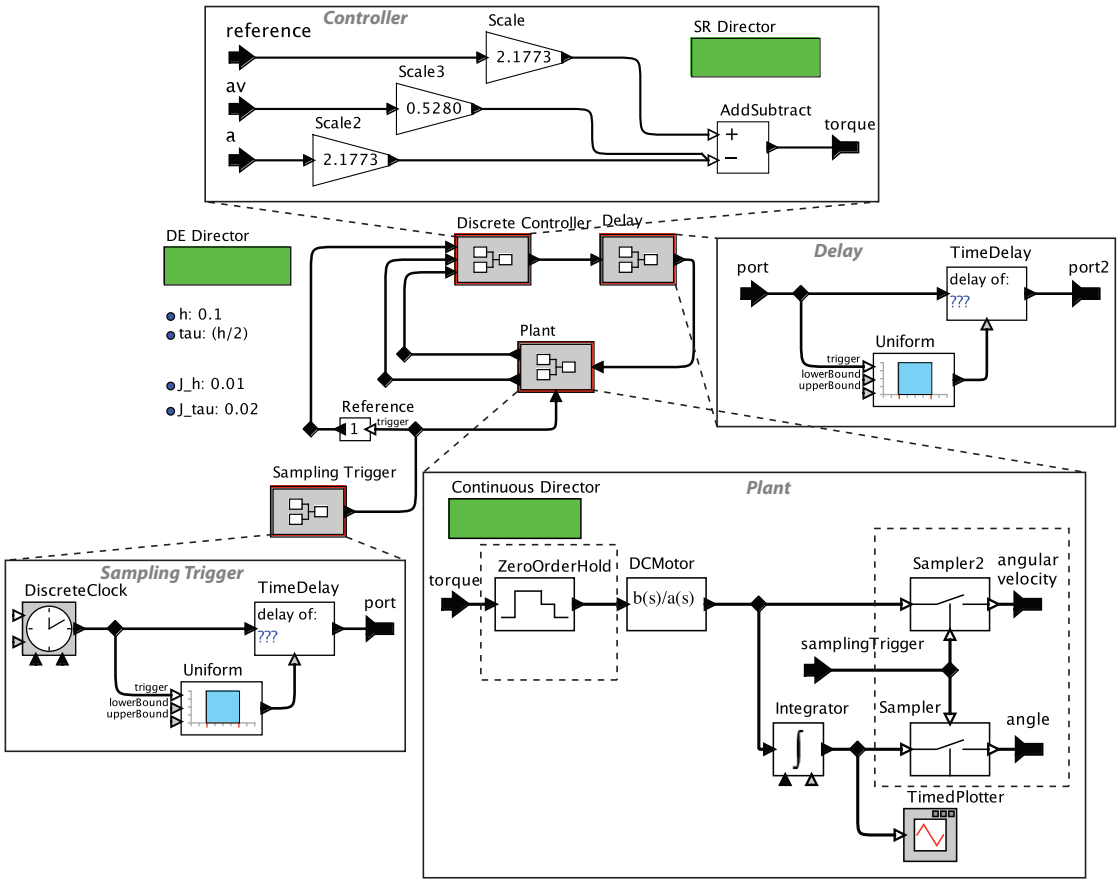


Figure 11: Ptolemy model of the DC motor example with a TOL contract.

The top level model of Figure 11 is a DE model.[11] 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 are not explicitly modeled here but an extension of the model could contain analog-to-digital converters at the Sampler actor and digital-to-analog converters before the Hold actor to represent the I/O functions.

The controller is implemented as a synchronous reactive model. SR roughly corresponds to the LT domain described above. The controller is embedded in the DE domain and thus computes the control output every time events are received on the input. The controller computes outputs even in case not all inputs have input events (in this case some inputs are *absent*). In Ptolemy, different actors treat absent inputs differently. In this example, absent inputs are treated as having value 0. The control output is incorrect if value 0 is used, therefore we must make sure that every time the controller is executed, all inputs are present. In the model of Figure 11 this is achieved by triggering the constant actor with the same trigger signal as the one which

[11] One could also use the continuous director at the top level. This would allow a more natural restructuring of the model, for instance, by removing the Sampler and ZeroOrderHold blocks from inside the Plant. On the other hand, using CT in the entire model generally slows down simulation. This is not an issue for the simple model of Figure 11, however.

defines the sampling events for the plant.[12]

The timing described by the TOL contract is implemented in the Sampling Trigger and the Delay actor. These actors have no explicit director specified inside them: this means that they implicitly "inherit" the director of their parent, that is, DE in this case. 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 (in the example $\tau = h/2$, $J^\tau = 0.02$).

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 fixed instead of variable delays (or no delays at all). Performing this type of experiment in Ptolemy we obtain the simulation results shown in Figure 12.

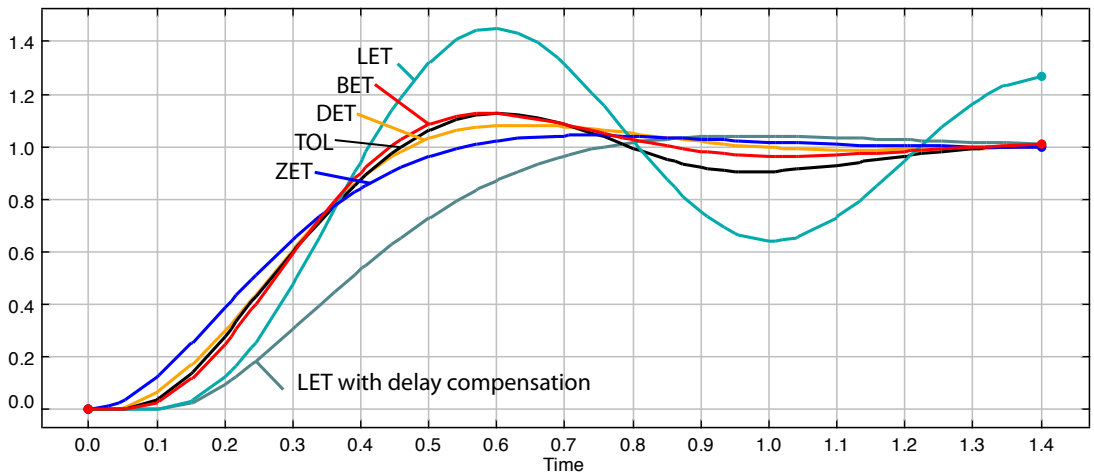


Figure 12: Control performance of various contracts on the DC motor example modeled in Ptolemy.

5.3.2 Modeling and Simulating Design Contracts in Simulink

In Simulink, the TOL contract can be modeled as shown in Figure 13. In Simulink, all signals are essentially CT signals. Discrete-time (DT) signals can be modeled as piecewise constant CT signals. The model of Figure 13 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. This mechanism is used in particular to capture times t_k^s and t_k^a: t_k^s is captured by the rising edges of the output signal of block TriggerValues, while t_k^a is captured by the falling edges of the same block.

Another difference between Simulink and Ptolemy is how to model trigger signals. In the model of Figure 13 the triggers are defined off-line (i.e., in the Matlab workspace) as vectors – c.f. label "From Workspace" on the TriggerValues block. The latter produces signals with rising and falling edges as mentioned above. These signals trigger two event-triggered subsystems, causing them both to behave as time-varying samplers; with the former acting as the real sampler and the second as a time-varying StA delay.

[12] Another way to ensure correct controller execution and avoid the issue of absent values is to use Register or Sampler actors to store previous values. A register or sampler actor stores the input value until a new value is received. Every execution of the controller would then read values from the sampler actor. The execution of the controller would then not be triggered by input events but by a separate trigger source.

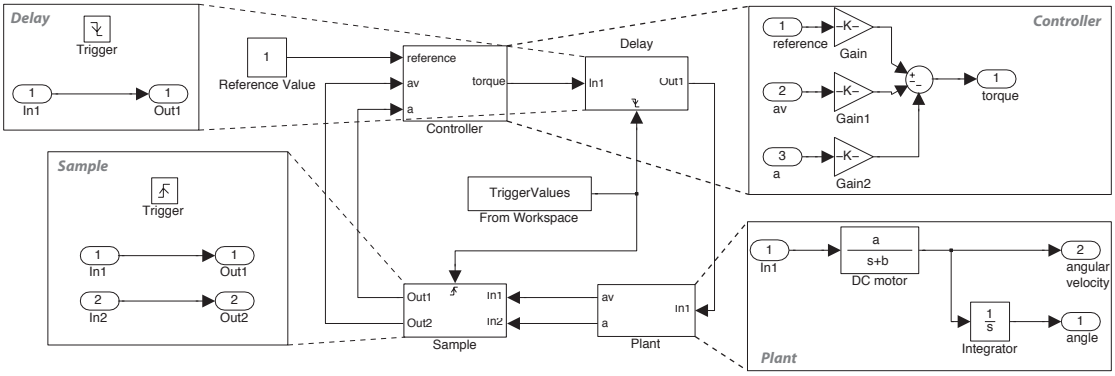


Figure 13: Simulink model of the DC motor example with a TOL contract.

5.3.3 Modeling Delay Compensation

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. For the DC motor example, this means using the control signal that is computed by the controller also as an input to the controller (this means that the controller now has state). The gains have to be recomputed to accommodate this change.

Figure 14 illustrates a model of the controller presented in the previous sections improved with delay compensation.

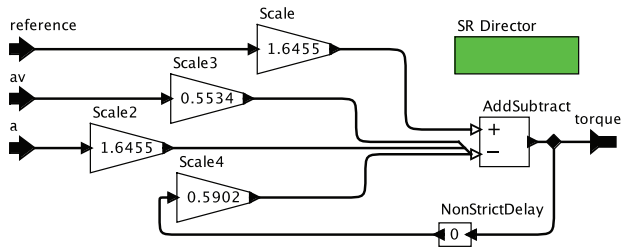


Figure 14: Controller with delay compensation in Ptolemy.

An attempt to apply a similar change to the TOL model in Simulink is illustrated in Figure 15. The unit delay block has a fixed sample time and is thus executed periodically. However, if we have 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. Therefore we must make sure that the control signal is sampled at the same time input values to the controller from the plant are sampled. This is not the case with the model of Figure 15.

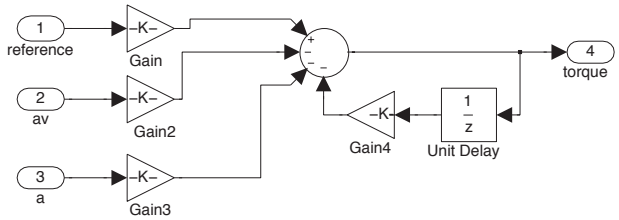


Figure 15: First attempt to model a controller with delay compensation in Simulink.

A correct implementation of the delay compensation in Simulink is presented in Figure 16. The output

of the controller is memorized and sampled by the sampling block.

As it can be seen, 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.

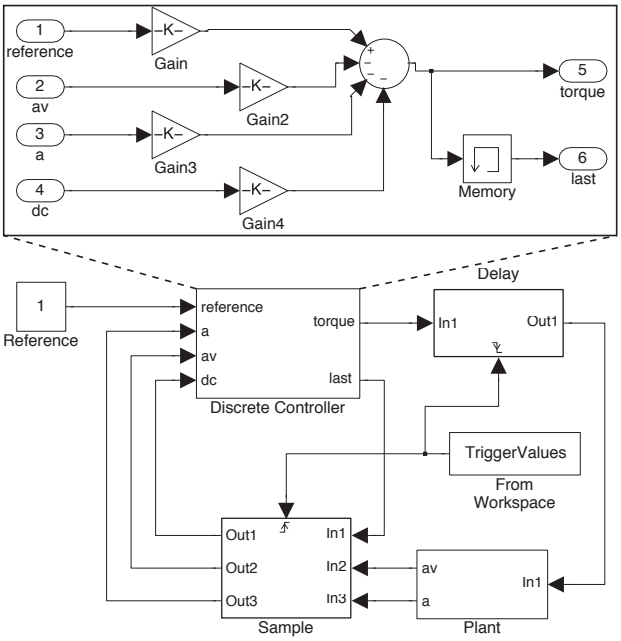


Figure 16: Modeling delay compensation in Simulink.

6 Conclusions and Future Directions

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. First, it would be worth investigating 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.

Second, there is a need to further develop the formalization of existing contract frameworks into complete contract 'algebras', e.g., along the lines of [20, 42]. 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 [51], while the compositionality of LET contracts is unclear [33].

Third, apart from functionality and timing, contracts must be developed for other aspects, including those outlined in Section 4.5.

Fourth, for controllers with more complex structures, such as in multiple-input multiple-output systems involving multiple rates, the design contracts will relate both to the interactions between subcontrollers as

well as to the interaction with the embedded software design. We believe that our approach is very relevant here, especially since such systems often involve multiple suppliers and complicated systems integration.

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 among others platform-space exploration, mapping, model transformations and compiler optimizations).

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.-E. Årzén, A. Robertsson, D. Henriksson, M. Johansson, H. Hjalmarsson, and K. H. Johansson. Conclusions of the artist2 roadmap on control of computing systems. *SIGBED Rev.*, 3(3):11–20, July 2006.

[5] K.E. Årzén, A. Bicchi, S. Hailes, K.H. 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*, October 2006.

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

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

[8] A. Benveniste, B. Caillaud, and R. Passerone. Multi-viewpoint state machines for rich component models. In P. Mosterman and G. Nicolescu, editors, *Model-Based Design for Embedded Systems.* CRC Press, 2009.

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

[10] E. Bini and A. Cervin. Delay-aware period assignment in control systems. In *Proceedings of the 2008 Real-Time Systems Symposium*, RTSS'08, pages 291–300, Washington, DC, USA, 2008. IEEE Computer Society.

[11] L. P. Carloni, R. Passerone, A. Pinto, and A. L. Sangiovanni-Vincentelli. Languages and tools for hybrid systems design. *Found. Trends Electron. Des. Autom.*, 1(1/2), 2006.

[12] 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, February 2008.

[13] A. Cervin, K.E. 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*, October 2006.

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

[15] A. Cervin and J. Eker. Control-scheduling codesign of real-time systems: The control server approach. *Journal of Embedded Computing*, 1(2):209224, 2005.

[16] A. Cervin, B. Lincoln, J. Eker, K.E. Årzén, and G. Buttazzo. The jitter margin and its application in the design of real-time control systems. In *Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applications*, Goeteborg, Sweden, August 2004.

[17] A. Cervin, M. Velasco, P. Marti, and A. Camacho. Optimal online sampling period assignment: Theory and experiments. *IEEE Transactions on Control Systems Technology*, 19(4):902–910, 2011.

[18] EU Project CESAR. Survey report on modeling languages, components technologies and validation technologies for real-time safety critical systems – v 2.0, 2010. Available as http://www.cesarproject.eu/fileadmin/user_upload/CESAR_D_SP3_R1.2_M2_v1.000.pdf.

[19] A. Davare, Q. Zhu, M. Di 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.

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

[21] 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.

[22] D. Densmore, R. Passerone, and A. Sangiovanni-Vincentelli. A platform-based taxonomy for esl design. *IEEE Design & Test of Computers*, 23:359–374, 2006.

[23] 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.

[24] J. Eker, P. Hagander, and K.-E. Årzén. A feedback scheduler for real-time controller tasks. *Control Engineering Practice*, 8(12):13691378, 2000.

[25] 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.

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

[27] 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 Conference on Decision and Control*, pages 199–204, 2006.

[28] A. Ghosal, T. Henzinger, C. Kirsch, and M. Sanvido. Event-driven programming with logical execution times. In *HSCC'04*, LNCS 2993, pages 357–371. Springer, 2004.

[29] 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.

[30] D. Goswami, R. Schneider, and S. Chakraborty. Co-design of cyber-physical systems via controllers with flexible delay constraints. In *16th Asia and South Pacific Design Automation Conference*, ASPDAC'11, pages 225–230. IEEE, 2011.

[31] 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.

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

[33] C. M. Kirsch and R. Sengupta. The evolution of real-time programming. In *Handbook of Real-Time and Embedded Systems*, pages 11.1–11.15, 2006.

[34] R. Majumdar, I. Saha, and M. Zamani. Performance-aware scheduler synthesis for control systems. In *EMSOFT'11*, pages 299–308. ACM, 2011.

[35] B. Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, 1992.

[36] L.A. Montestruque and P. Antsaklis. Stability of model-based networked control systems with time-varying transmission times. *Automatic Control IEEE Transactions on*, 49(9):1562–1572, 2004.

[37] J. Nilsson. *Real-Time Control Systems with Delays*. PhD thesis, Department of Automatic Control, Lund Institute of Technology, 1998.

[38] M. Pajic, S. Sundaram, G.J. Pappas, and R. Mangharam. The wireless control network: A new approach for control over networks. *Automatic Control, IEEE Transactions on*, 56(10):2305 –2318, October 2011.

[39] 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.

[40] H. Rehbinder and M. Sanfridson. Scheduling of a limited communication channel for optimal control. In *39th IEEE Conf. on Decision and Control*, pages 1011 –1016, 2000.

[41] 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.

[42] 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.

[43] D. Seto, J. P. Lehoczky, L. Sha, and K. G. Shin. On task schedulability in real-time control systems. In *17th IEEE Real-Time Systems Symposium*, RTSS'96, pages 13–. IEEE, 1996.

[44] K. G. Shin and H. Kim. Derivation of hard deadlines for real-time control systems. *IEEE Transactions on System Man and Cybernetics*, 22(6):1403–1413, 1992.

[45] 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.

[46] C.-J. Sjoestedt. *Modeling and Simulation of Physical Systems in a Mechatronic Context*. PhD thesis, KTH, 2009.

[47] K. Smeds and X. Lu. Effect of sampling jitter and control jitter on positioning error in motion control systems. In *Precision Engineering*, volume 36, pages 175–192, April 2011.

[48] 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.

[49] J. Sztipanovits, X. D. Koutsoukos, G. Karsai, N. Kottenstette, P. J. Antsaklis, V. Gupta, B. Goodwine, J. S. Baras, and S. Wang. Toward a science of cyber-physical system integration. *Proc. IEEE*, 100(1):29–44, 2012.

[50] M. Törngren. Fundamentals of implementing real-time control applications in distributed computer systems. *J. of Real-Time Systems*, 14:219–250, 1998.

[51] 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.

[52] S. Vestal. Integrating control and software views in a cace/case toolset. In *Computer-Aided Control System Design, 1994. Proceedings., IEEE/IFAC Joint Symposium on Computer-Aided Control System Design*, pages 353 –358, 1994.

[53] 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.

[54] Y. Wu, G. Buttazzo, E. Bini, and A. Cervin. Parameter selection for real-time controllers in resource-constrained systems. *IEEE Transactions on Industrial Informatics*, 6(4):610–620, November 2010.