2013 IEEE 16th International Conference on Computational Science and Engineering

# Utilizing Intervals in Component-based Design of Cyber Physical Systems

Steffen Peter and Tony Givargis
Ctr. for Embedded Computer Systems, University of California, Irvine, USA
Email: {st.peter, givargis}@uci.edu

***Abstract*—Within the design of Cyber Physical Systems, model-based approaches are powerful means to describe and test the behavior of the system. Still, a good methodology is needed to go from the idealized model environment to an implementable system architecture that is capable of dealing with uncertainties in both the physical and the cyber subsystem. This paper presents a concept that explicitly utilizes intervals to express uncertainties in the physical system, the control process and the cyber system to improve the robustness and stability of the design. This interval concept has been integrated in a component-based framework that allows one to describe properties of the components out of which the CPS is composed. With a prototype implementation of the component framework, this paper shows the usefulness of this approach for an exemplary CPS. The results indicate the practical benefits of value intervals for property assessment of composed CPSs which can be exploited at design time as well as run time.**

***Keywords*–Cyber Physical Systems, design, intervals, components, models**

## I. Introduction

The design of Cyber Physical System (CPS) is a complex task. CPSs are systems in which an integrated cyber subsystem (CS) interacts with a physical subsystem (PS). This interaction adds to the already existing challenges in the design of the embedded computation systems, on one side, and the physical system, on the other side. As an example, the PS may be as complex as the mechanical drive train in a car or the hydraulic system in a water distribution network.

The model-integrated and model-based design (MBD) approach [8, 9] is the most promising design paradigm for CPSs and has become the de-facto standard approach in many industries with capable tools like Simulink [1]. In these tools, models of the PS and the CS are simulated, interact side-by-side and allow the designer to develop and test algorithms for the CPSs. The models of the subsystems, naturally, are abstract approximations and simplifications of the real systems in order to understand and simulate them with reasonable effort. MBD-flows successively refine the CS from a high behavioral abstraction level, ideally, to a level that allows direct implementation. An important element of this refinement process is (de)composition. Already, the definition of CPS states that there is a cyber system and a physical system which are composed to work as one logical unit. Beyond that, the cyber part of the CPS can be decomposed into sub components such as sensors, actuators and the computation processes. This decomposition approach has a range of benefits. Smaller sub systems are better understood and can be verified against their specifications, previously implemented components can be reused, and parallel development of components is fostered. In this paper, we pursue the notion of composition of reusable components to define the architecture of a CPS and to analyze its properties on a high abstraction level.

Component frameworks for CPSs are not new and, for instance, have been applied to automotive [4] and space applications [14]. While existing component frameworks are capable in composing software, they mostly ignore the variability of the physical subsystem and the interfaces between CS and PS. Furthermore, the semantics of the composition in existing frameworks is limited mostly to interface- and compatibility checking. Reasoning about non-functional properties such as timing, security and dependability is very limited and usually only worst case properties are considered. For many CPSs worst case timing analysis is not sufficient because too late or too early could result in errors. For instance, the heart pacemaker presented in [12] runs one operation per millisecond, for which the authors state that a higher frequency would compromise the quality of the implemented control algorithm. Intervals provide mechanisms to cope with such upper and lower boundaries in the requirements. Specifically, intervals may be integrated as part of the system design methodology to explicitly take into account uncertainties in the models and carry this information throughout the design processes, from the ideal abstract models to an implementable architecture.

Applying such explicit value intervals, in this paper, we address the question whether a given system architecture, whose properties to a certain degree are expressed with uncertainty, still satisfies the requirements of the CPS. The main contributions of our work are:

1) We thoroughly discuss the concept of value intervals in the context of CPS and show their importance in the definition and design of the physical and the cyber subsystems.
2) We show how the interval concept can be applied within a component framework that enables the composition of a CPS system architecture out of functional building blocks, covering hardware, software and environmental properties.

We present a prototype tool that for a given system, whose component properties can be expressed with uncertainties, determines whether the system satisfies the requirements imposed by the control algorithm, or whether the system contains conflicts. We motivate the contributions and discuss their effectiveness by a novel example use case –the Falling Ball– that combines many design challenges of CPSs.

The rest of this paper is structured as follows. Section II introduces the running example and outlines its design challenges. Section III provides a brief overview of related work, Section IV discusses the role of intervals in the CPS design. The CPS component framework that processes these intervals is introduced in Section V. Results for our prototype tool and a set of design examples are presented in Section VI.

978-0-7695-5096-1/13 $31.00 © 2013 IEEE
DOI 10.1109/CSE.2013.99

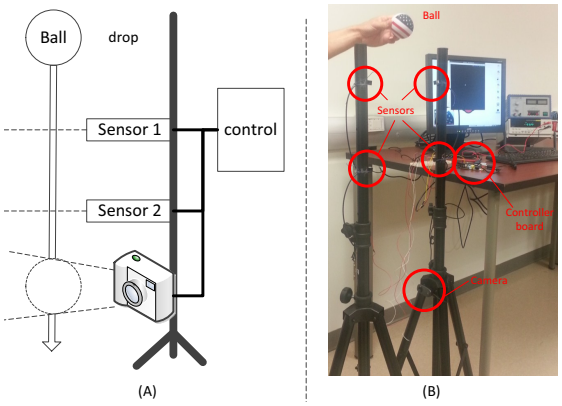


Fig. 1. Setup of the Falling Ball example: (A) as schematics, (B) in practice.

## II. Example and Design Challenges

In this section we introduce the running example that we will use throughout this paper. Further we show a typical model-based design process and discuss the challenges in designing such a system.

### A. The Falling Ball example

The Falling Ball example, illustrated in Figure 1, is a CPS with the objective to take a picture exactly at the moment a falling ball passes a camera. The approaching time is predicted based on information from motion sensors mounted above the camera. This simple example has some interesting properties of CPSs. A proper implementation requires exact timing of various events. This is one example where faster is not necessarily better. Second, the mathematical model of the general physical process is well understood, and still, due to small variations in physical parameters (e.g. gravity, air resistance), we will not achieve perfect precision. Third, sensors and actuators are part of the process for which we can study the impact of alternatives to these components. Additionally, the example can be implemented with little manual effort so that we can observe and compare effects in the models as well as in the real world using an actual implementation.

In the following we describe a typical model-based design flow for CPSs. In each step additional details of the example will be discussed.

### B. Model-based design

The underlying principle of model-based design (MBD) is to create models for the system and simulate these models to evaluate properties of the system. We illustrate the four steps to establishing the executable simulation for the Falling Ball example. A more detailed model-driven design flow for CPSs is described in [8].

**Definition of the top-level architecture of the system:** The first step in the MBD is the definition of the main entities of the system and the interfaces and data paths between the entities. For the Falling Ball example these information are shown as block diagram in Figure 2. The diagram defines the Ball entity (the PS) the Control entity (the CS), and the two sensors and the camera components.

**The description of the PS:** The physical system is most conveniently expressed as differential equations. In case of the Falling Ball, the height of the ball is determined by the differential equation of velocity over time, while velocity is determined by the differential equation of acceleration over time. This small system of equations is initialized with a constant acceleration (gravity=$9.81m/s^2$) and a selected initial height, i.e. the drop height of the ball.

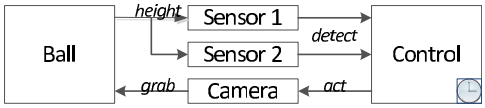


Fig. 2. System entities and high level data flow.

**The control program:** Since the control program, usually, cannot be expressed with differential equations, it is necessary to express the control logic as a sequential program. In the case of the Falling Ball example, this step can be resolved by creating a program based on Newton's free fall equations. These equations dictate the logic of the control program that first, waits for sensor reading, then computes the expected time for the ball to reach the camera, and then sends the trigger signal. As part of modern design and simulation tools, this control program can be represented as a state machine or as a sequential program captured using some imperative language.

**Simulation and test of the control algorithm (CA):** When all components are specified, the system can be simulated. This allows the designer to validate a system before implementation for a given set of properties of the components. The simulation results express whether or not the system is stable, while typically additional properties such as consumed time or energy can be evaluated as well.

### C. Problem Formulation

After the definition and test of the control algorithm (CA), suitable hardware and software components can be selected or implemented, which can be assembled to a system that satisfies the requirements of the control algorithm. The problem is that any variation of properties from the simulated system may void the results of the simulation. This is crucial as many conditions of the system are not know at design time. For instance, the computing platform or the applied sensors or actuators may be unknown before the actual installment of the system. Properties of the PS often can never be expressed with full precision due to the limitations of the underlying model.

In practice this leads to an iterative process in which the knowledge from the integration phase contributes to refine specifications of the models. The refined models then can be applied to adapt the control algorithm if necessary. Such development practice does work, but is time consuming, limits the reusablity of designed algorithms and imposes severe challenges when organizational or temporal constraints forbid a close cooperation between the designer of the algorithm and integrator of the system. Preferable, instead, is the decoupling of control design and integration in a way that the control algorithm provides the specification to the lower layers, in which the system can be designed and implemented. In order to be actually implementable, this specification has to provide space for variation of system properties. As result any higher level abstract component can be replaced by an actual component or subsystem in practice, as long as the properties of the actual system are in range of the assumed

values of the simulated component. This, at first, needs a good understanding of sources and characteristics of uncertainties in CPS design, and second, a technical framework to work with these uncertainties. We address these issues in section IV and V, respectively, after a review on related work.

## III. Related Work

Model based design approaches have been proposed in a range of related work addressing the challenges of CPS design [8, 9]. As example Wang [19] presents an integrated modeling and simulation environment for CPS and shows how integrated simulations can be used to improve the control performance. Sztipanovits [17] proposed model-driven approaches with domain-specific models and general composability across the domain barriers. These works provide valuable contributions to improve the results from modeling and simulation early in the development, but do not address the translation to a system architecture.

With the motivation to bridge the work domains of control engineers and computer engineers, Sangiovanni-Vincentelli [16] and Damm [4] proposed explicit design contracts for vertical and horizontal composition. Recently, Derler et al [5] discussed a generalization of design contracts for timing properties in CPSs. The work also considered explicit delay compensation as part of the control algorithms. While delay compensation can improve the control performance of real systems with delays, the experiments also showed that the amount of compensation has to be chosen carefully, since both too pessimistic as well as too optimistic compensation can result in control violations.

A formal approach facilitating state machines and concurrency analysis tools is discussed by Lee in [11]. The approach advocates for simulations with deterministic timing and is valuable for verification of systems, while it has limitations with respect to uncertain properties of the system. Even though not further evaluated in our paper, formal methods to check interoperability and correctness of interface compositions are important for system composition, and we see interval analysis of component properties as extension to those approaches.

A variety of component frameworks for CPSs have been proposed, while we highlight REMORA [18] and REMES [3]. REMORA proposed reconfigurable components that can be parameterized at run time, provided by high-level and event-driven programming in CPSs through a component-based abstraction. While runtime adaptation of CPSs exemplary motivates decoupling of control algorithms and implementation, REMORA does not address selection of proper components. REMES [3] is an approach to reasoning about service behaviors and their compositions, using annotated state machines that can be applied sequentially and hierarchically to assess properties of the composed systems. A formal verification tool, as part of REMES, centers around data flows and their pre- and post-conditions to evaluate system attributes such as consumption of time or memory. The approach was extended [2] to add support for the assessment of composed worst case execution time. So far REMES does not address physical properties of CPSs and does not support the definition or validation of properties expressed as intervals.

Interval analysis in CPS has been proposed to detect faults during runtime by Sainz [15]. Based on measurements at run time and a system model they determine intervals of future sensor readings in the system. In case the values are out of the predicted intervals, a fault will be reported. This approach only focuses on measured values processed during runtime and does not cover design decisions.

The application of interval arithmetic for constraint reasoning has been discussed by Hyvonen [6]. While his work only partly overlaps with the challenges of CPS design, the arithmetic definitions and rules find direct application in our work.

## IV. Intervals in CPS

An interval, in general, is a subset of a totally ordered set of elements, and is denoted by $[a, b]$ while $a$ and $b$ are the end points of the interval, so we can write:

$$[a, b] = \{x \in \mathbb{R} | a \leq x \leq b\}.$$

The end points belong to the interval. In most cases $\mathbb{R}$ represents the set of real numbers, while in general it can be any ordered set. In CPSs we find the following intervals:

**Definition Intervals of the Physical System (IV_P)** are intervals imposed by the natural boundaries of the PS and the physical laws that govern its behavior. Most processes in the real world are bound in some way, which is not reflected by simple simulation models. For instance, the ball in our example can only be dropped from a certain range of heights, which affects the speed the ball has at and between the interface elements. This determines the minimal and maximal response time for a system, directly influencing the choice of hardware (sensors, actuators) and software components in a final implementation.

**Uncertainties of the Physical System (IV_U)** are caused by the deviations between the real physical system and its model, which applies simplifications and approximations. As we cannot express these deviations exactly, such uncertainties extend the range of values the system has to cover. In the case of the Falling Ball example the gravimetric acceleration is influenced by these uncertainties. As we will see in Section VI, uncertainty in the gravity constant may extend the properties of the design to such a degree that the eventual system no longer may satisfy the original design requirements.

**Uncertainties of the Cyber System (IV_C)** reflect the fact that the models of the CS and its components are approximations. The timing behavior of many hardware components, such as sensors and actuators, is explicitly specified by intervals. In the Falling Ball example, this uncertainty may reflect system runtime variation due to memory delay, cache misses, other system processes, or system interrupts. Extended to the interface components, depending on the quality of the sensors, we have uncertainties in the detection and propagation time of the sensor.

**Definition space of the Control Algorithm (IV_D)** reflect the fact that the control algorithms of CPSs usually are developed with a certain environment and system in mind. Hence, the resulting algorithm is only specified for this range of values. If the PS or the CS is out of this verified range, proper functioning of the algorithm is no longer guarantied.

To summarize, each subsystem of a CPS is subject to a range of uncertainties at design time. Figure 3 illustrates the top level components of a CPS with their sources of uncertainties. The PS, which describes the properties of the

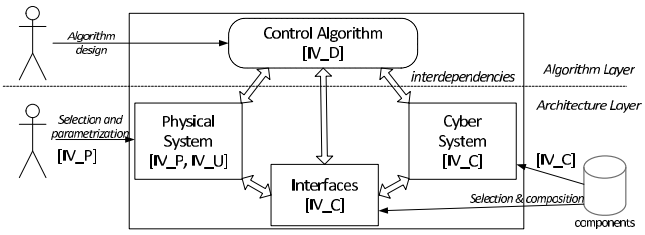


Fig. 3. Top-level CPS design problem: The abstract Control Algorithm component logically connects PS and CS which are physically connected by interfaces. Initially the CPS and the PS are defined by the developer, while the CS and Interfaces may be assembled from a set of technical components. Each subsystem is exposed to uncertainties that have to be addressed with intervals [IV_*].

environment, is exposed to its definition intervals (IV_P) and model uncertainties (IV_U). The cyber subsystem and the interfaces, which are physically connecting the PS and the CS, are assembled from a set of components. For the individual components as well as for the composed system the uncertainty IV_C applies. The CPS component, which is an abstract component that logically connects PS and CS and represents the requirements and characteristics of the underlying control algorithm, is exposed to IV_D.

Between the entities of the system exist a range of dependencies and connections. For instance, the CA has been developed with a certain physical environment in mind. Hence, we have to validate that the properties of the PS with its uncertainties (expressed with IF_P and IV_U), are within the range of the supported values of the CPS. Similarly, the properties of the CS and the Interfaces have to be in line with the assumptions of the CA. Beside vertical interdependencies, which in our example range from the algorithm layer to the architecture layer in Figure 3, we also have to respect horizontal connections within the architecture layer. For instance, the properties of the applied interfaces influence the properties of CS and PS. As example, the delay of the interfaces may change timing attributes in the CS, which eventually have to be validated against assumptions imposed by the CA.

The result of this view on the system is a network of properties, which cannot be expressed as fixed values. For this network and under consideration of individual uncertainties, a developer is interested in whether or not this network contains any potential conflicts. To answer this question, in the following section we introduce a framework that enables the developer to compose a system out of components and to evaluate the correctness of this composed system applying intervals.

## V. Components in the CPS design

As already introduced in Section I, using components to assemble the system is a reasonable and promising concept in coping with the design challenges of CPSs. Related work has addressed a variety of component models [3, 4, 18]. The work in our paper is mostly related to the concept of rich components as proposed by Damm [4] and further refined by Sangiovanni-Vincentelli et al [16]. Rich components extend classic component models (e.g. from UML) by meta information that express functional and non-functional aspects of the components in clear semantics. As result any component $C$ can be represented by the triple:

$$C = (B, I, M),$$

while $B$ represents the behavior, $I$ the interfaces, and $M$ the meta information of $C$. We discuss the role of the behavior and the interfaces in the following, and the meta information in subsection V-B.

### A. Component Model

Components are the building blocks of the system. Each component represents a certain characteristic behavior, which may be the behavior of a software module, a hardware modules, or a physical subsystem. As an example, the high-level system architecture in the Falling Ball, discussed in Section II and shown in Figure 2, contains building blocks (ball, control, sensors, camera), which we consider as components.

Components provide their functions to other components via interfaces. Reciprocally, interfaces are used to connect to other interface in order to use their functions, so that we can express the interfaces as

$$I = (I_P, I_U),$$

while $I_P$ is the set of interfaces provided by the component, and $I_U$. During the validation of composition, which is outlined in subsection V-C, it is assured that each needed interface $I_U$ of the components in the systems is provided by one interface $I_P$.

An integral part of our component model are abstract functions, such as algorithms or protocols, which are expressed as components. Having components of different abstractions in the framework enables vertical decomposition and validation of the behavior across layers. The Control Algorithm block, depicted in Figure 3, is such an abstract component. Another example for the need for vertical composition is the CS block (in Figure 3), which can be further decomposed and refined according to the well-studied practices of embedded system design. Figure 4 shows a process state machine as exemplary decomposition of the CS for the Falling Ball. The components in this diagram are processes that can be implemented as software running on a microcontroller. Properties of each of the processes contributes to the aggregated behavioral properties of the system. The semantics of the properties and an approach for their aggregation is discussed in the following.

### B. Properties and Assumptions

In addition to the structural information, each component $C$ contains meta-information $M$, expressing the properties $P$

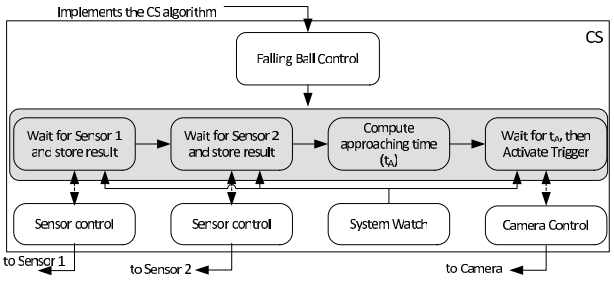


Fig. 4. Decomposition of the Falling Ball CS. The Falling Ball program can be decomposed as sensing, prediction and actuation. The sensing and actuation blocks need control drivers for the interfaces.

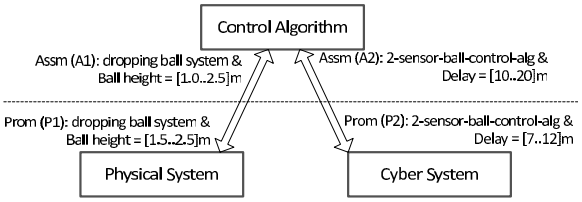


Fig. 5. Example for vertical contracts between Control Algorithm and PS and CS. In this case $A1 \subseteq P1$, but $A2 \not\subseteq P2$.

of the component, and the assumptions $A$ about properties of other components in the system. This concept is related to design contracts [4, 16, 5], in which it is assumed that two components $C_1$ and $C_2$ with their contract data as part of the information $M_i = (P_i, A_i)$ can be assembled if $A_1 \subseteq P_2$ and $A_2 \subseteq P_1$, meaning that all assumptions of a component are delivered by promised properties by the peer. A system $S$ of components is free of conflicts if the aggregated set of assumptions is satisfied by the aggregated set of proposed properties, i.e.

$$\bigcup_{component\ c \in S} A_c \subseteq \bigcup_{component\ c \in S} P_c.$$

An example for contracts applying intervals in the Falling Ball example is illustrated in Figure 5. In this example the CA has the assumptions, first, to have a (A1) dropping ball system that supports dropping height from 1.0 to 2.5m, and second, to have (A2) a two-sensor ball control algorithm with a delay from 10 to 20msec. The PS promises to (P1) provide the ball system and supports a dropping height from 1.5m to 2.5m. The CS promises (P2) the two-sensor ball control algorithm with a delay from 7 to 10ms. We can see that $A1 \subseteq P1$, but $A2 \not\subseteq P2$, because the delay promised by the CS partly lies outside the assumed interval. In this case A2 is not satisfied, and therefore the composed system has conflicts that have to be resolved.

In the previous section we already discussed the example of the delay of the CS, which is composed of the delay of the computation blocks (see Figure 4) and the delay of the interfaces. Static contracts cover such composed properties only partly. As option, one can decompose the timing budget from the top to the bottom, so that the CS expresses clear timing assumptions to the interfaces, as well as to each of the computation blocks. The top-down budget decomposition approach needs significant in-detail knowledge about the applied components, and indicates conflicts already for small violations which could be compensated by budgets from neighbored components in the system.

To cope with this issue in our framework, we extend the concept of static contracts by allowing symbolic definitions of properties. Properties then can be defined in context of the composition as functional mapping of other properties in the system. We call this functional mapping between properties as Relation. The relations then extend the meta-information for a component to the triple

$$M = (P, R, A),$$

with the properties $P$, the relations $R$ and the assumptions $A$.

A Property $p \in P$ is a characteristic or quality that describes an attribute of the system. Properties comprise all aspects of a system that can be expressed by a value, including all sorts of attributes to describe the system, its components, its requirements, or its environment.

Properties influence and depend on other system attributes. In the Falling Ball example, the selection of the camera, having some viewing angle characteristics, affects properties in the PS. To express these dependencies, we use Relations $R$. A Relation $r \in R$ is a triple $r = (f_r, p^*_{in}, p_{out})$, while $f_r$ is a function over a set of input properties ($p^*_{in} \subseteq P$) that assigns the result to a property ($p_{out} \in P$), so that we can write

$$f_r(p^*_{in}) \Rightarrow p_{out}$$

Figure 6 (A) illustrates an example relation, representing the equation `system_delay:=sensor_Delay+computation_delay`, which defines the property `system_delay` as summation of `sensor_delay` and `computation_delay`. In this example $f_r = add$, $p_{out}$ =`system_delay`, and $p^*_{in}$ =(`sensor_delay, computation_delay`).

Assumptions $A$ are similar to relations as they compute a function over a set of input properties. However, the result of the assumption is a boolean which is not assigned to a property, so that an assumption $a \in A$ is a tuple $a = (f_a, p^*_{in})$ and

$$f_a(p^*_{in}) \Rightarrow \{true|false\}$$

Assumptions in our model are treated like assumptions in design contracts in the way that all assumptions of components that are part of the system under evaluation have to be satisfied, i.e. must be $true$.

$$\forall a \in A : f_a(p^*_{in\,a}) = true,$$

The Assumption in the example of Figure 6(B) is a boolean functions that is *true* if `system_delay` is within ($\subseteq$) the `allowed_system_delay`. The $\subseteq$ operation is resolved dynamically based on the type of the properties. Assumed, the `allowed_system_delay` is a fixed number (e.g. 20 msec) the assumption is satisfied if `system_delay` is smaller or equal 20 msec. If `allowed_system_delay` is expressed as interval (e.g. [10..20]ms) the assumption is satisfied if `system_delay` is within that range [10..20]ms.

Applying the concepts of interval arithmetic [6], relations and assumptions can process intervals (e.g. $[a_1, a_2]+[b_1, b_2] = [a_1 + b_1,\ a_2 + b_2]$). In the example of Figure 6, assumed the `sensor_delay` is specified as the range between 3 and 5 msec and the `actuator_delay` is between 15 and 18 msec, the resulting `interface_delay` is [18..23]msec, and the constraint of 20 msec is not satisfied.

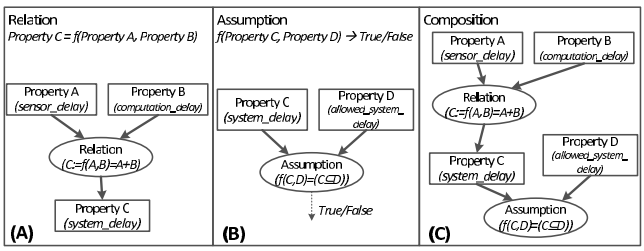


Fig. 6. Graphical representation for a Relation (A), a Assumption on Properties (B), and a composition of both (C).

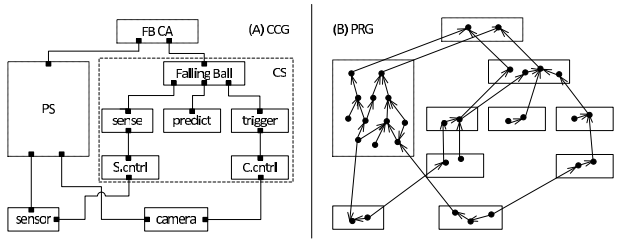


Fig. 7. On the left, a possible composition of the Falling Ball CPS is shown. On the right, the behavioral model of this composition is illustrated. Each dot in this graph represents a variable, condition, or constraint of the system. The system on the left is valid if the representation on the right contains no constraint violation.

The set of relations and assumptions of a component can be intuitively represented as finite directed graph in which properties, defined by a relation are element of the input properties ($p^*_{in}$) of other relations or assumptions. Figure 6(C) illustrates the composed graph for our running example. In the system composition process, which is outlined in the following, these individual relations can be merged to the system-wide description of properties.

### C. Composition of Components and Properties

During the composition and binding of the system out of individual components we have to address two major concerns. First, the structure of the composed system, and second, the properties of the system. These two aspects are addressed by two graph structures we maintain during the composition phase: the Component Composition Graph (CCG) and the Property Relation Graph (PRG).

The CCG represents the components of the system and expresses how they connected via the interfaces. The CCG is the blueprint for the actual system integration.

The PRG combines the property description of the components in the CCG to a complete behavioral system description. Considered, for instance, one component provides the Relation of Property C as shown in Figure 6, and a connected component requires the constraint (C<=D), then the resulting PRG combines both partial graphs, so that the Assumption is always evaluated whenever one of the properties (A, B, C, or D) is changed.

As such, the PRG is the merged graph of the properties $P_S$, relations $R_S$, and assumptions $A_S$ of the individual components of the system ($S$): $PRG = (P_S, R_S, A_S)$

$$P_S = \bigcup_{c \in S} P_c, \quad R_S = \bigcup_{c \in S} R_c, \quad A_S = \bigcup_{c \in S} A_c.$$

This is a straightforward merge operation of properties, relations and assumptions that results in one holistic graph, which is structurally equivalent to the meta information $M$ of the individual components. Therefore, the composed assumptions of the systems can be evaluated just as the assumptions of the components. A system is free of conflicts if all assumptions are satisfied, i.e.

$$\forall a \in A_S : f_a(p^*_{in_a}) = true,$$

while all relations $R_S$ of the PRG have been resolved. Assumptions that remain $false$ are conflicts in the system, equal to contract violations in static design contracts, and have to be resolved by changing the components of the system. Tracing back the initial inputs of the violated assumption helps in identifying knobs to change in order to resolve the conflicts.

The PRG can grow quickly, as properties may influence all neighbored (connected) components. This complexity reflects the non-trivial system inter-dependencies, we already discussed in Section IV. In fact, this complex network of system properties illustrates the difficulty of making design decisions in CPSs in which changing the value, or the uncertainty, of one property affects many other system attributes. System properties that can be evaluated on this level of abstraction include timing [10], robustness and safety [7] memory consumption, energy consumption and qualitative security [13]. Figure 7 illustrates the CCG and the PRG for one possible composition of the Falling Ball example. Each dot in Figure 7(B) represents one property or function of the relations and assumptions. The arrows show the direction of the relations. Details of this example will be discussed in the evaluation in the next section.

## VI. Evaluation

To evaluate the suitability of intervals as part of the component-based design process for CPSs, we implemented a tool to process and analyze compositions as described in the previous section. The purpose of the evaluation tool is to establish the PRG for a given CCG, and to evaluate the properties, relations, and assumptions of the generated PRG. The output of this tool is a graphical representation of the PRG and a report on violated assumptions. A system is considered as implementable if the CCG is complete (i.e., is a single graph without unconnected interfaces), and the PRG is free of conflicts.

The primary data structure of the tool is a repository containing the component descriptions stored as XML files. As discussed in the previous section, components can represent abstract algorithms (e.g. the control algorithm), the physical subsystem, interfaces (e.g. sensors and camera) and components of the CS (computation processes). The data record for each component contain

- Key information about the component (name, id, file association),
- Used and provided interfaces ($I_U$ and $I_P$),
- Meta information $M = (P, R, A)$, expressing properties, relations, and assumptions of the component.

Properties may represent text (e.g. sensor.type=XT100), numbers (sensor.height=1.5m), or intervals (sensor.delay=[5..7]ms). The tool enables the CPS system developer to analyze any given component composition. The three main steps for this CPS analysis are a) chose the top level CA component, b) select and parametrize the physical system, and c) select components of the CS.

The choice of the top level CA component expresses the functional requirements via the used interfaces ($I_p$) and its assumptions. In the Falling Ball example, the CA system needs a PS that provides the Ball physics, and a CS that implements the control algorithm to trigger the camera and to access the sensors. In our example the CA component has only one rule: the expected timing of the CS must be within the acceptable timing deviation required by the PS.

The selection and parametrization of the physical system, which, as discussed above is considered as one component,

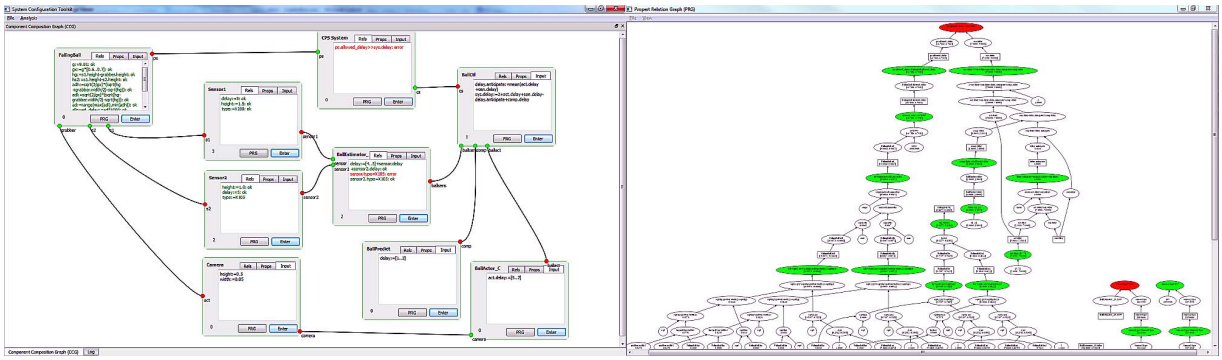

Fig. 8. Screenshot of our configuration tool. On the left, the user can edit the CCG (editing and binding of the components). On the right is the PRG with its satisfied or unsatisfied assumptions. Highlighted in red are violated assumptions.

are derived from the inputs set by the designer. These properties are forwarded to the top CPS component for which these variables define the requirements for the CS. The PS component of the Falling Ball example is parametrized by the dropping height of the ball, and the height of the sensors and the camera. The distances between these heights are processed in the property equations of the PS component to compute the timing requirements that are imposed as a requirement to the CPS system. With the CA and the PS set, components from the component repository can be selected that can be attached to the CCG to define the CS and interfaces.

We applied this design methodology for the Falling Ball example. For illustration purpose, Figure 8 shows a screen shot of the tool, with the CCG on the left and the resolved PRG on the right. A condensed version has already shown in Figure 7, where Figure 7 (A) shows the architecture of the system. In the following we discuss four use cases for this system. The results for these cases are summarized in Table I.

**Case 1:** In this first case we assume a perfect Newton acceleration. The ball will be dropped from a height of 1.0 meters above the camera, the second sensor will be placed 0.5 m above the camera. For this case, the PS computes an ideal time for the ball to travel from sensor 2 to the camera of 0.133 sec. In this case we chose a camera with an image width of 0.05 meters, this results in a precision requirement of [-5.7..5.6]msec. For the CS we assume all components to be perfectly specified and working according to this specification. This indeed leads to a perfect timing.

**Case 2:** In this case, we specify the components of the CS as well as the interfaces with more realistic values. The sensors have an assumed delay of [1..2]msec, while the camera is assumed to have a delay of [20..22]msec. The aggregated time for the computation is assumed to be 2 msec. The 'wait' component of the CS takes into account the average delays for the interfaces (22.5 msec), which results in a derivation of the CS from the ideal timing of [0.5..3.0]msec. This is well in line with the requirements.

In **Case 3:** In this case, we increase the dropping height of the all to 5m. This results in an allowed delay for the processing of 52 msec. With the components, selected so far, this delay requirement can not be satisfied, because in worst case the picture is taken 1 msec too late. To reduce this delay, a designer could select interface components with less jitter, or refine the specification of the processing tasks to obtain a better estimation of the timing.

**Case 4:** So far the physical system has been considered as ideal. Air resistance, however, limits the acceleration. This deviation depends on the density of the ball. If we add a simple model to the PS and assume 30 to 40% reduction of acceleration, the resulting approaching time of the ball will vary between 157 and 171 msec, while the allowed deviation increases due to the lower speed. The uncertainty of the PS in this scenario is too big to be covered by the CS. We would need more knowledge about the PS to have the confidence that the system works as intended.

This small example can be easily extended with additional rules. For instance, we can add the diameter of the ball as part of the PS. If the complete ball should be depicted, the size of the ball reduces the deviation the ball may have from the center of the picture to be taken. With these examples, we could demonstrate how small variations of properties, which we can easily describe for individual components, sum up to a complex network of properties. We could also learn that in this network, small uncertainties may aggregate to severe variations of system properties that cannot be controlled by any CS. Using our prototype tool, we can effectively set up and evaluate system configurations and their property networks. As an application of the tool, it can automatically iterate through a design space of CS compositions to identify setups that have a CCGs and PRGs that are free of conflicts.

TABLE I. SUMMARY OF RESULTS FOR THE FOUR TEST CASES.

| case | drop [m] | allowed delay[ms] | allowed dev.[ms] | CS delay[ms] | conflict |
|---|---|---|---|---|---|
| 1 | 1 | 113 | [-5.7..5.6] | 113 | no |
| 2 | 1 | 113 | [-5.7..5.6] | [113.5..116.0] | no |
| 3 | 5 | 52 | [-2.5..2.5] | [52.5..55.5] | yes |
| 4 | 1 | [157..171] | [-7.3..7.2] | [164.5..167.0] | yes |

## VII. CONCLUSIONS

In this paper we have shown that integrating uncertainties, expressed as value intervals, in the design of Cyber Physical Systems (CPS) is essential in order to meet design constraints, robustness requirements, and performance objectives. We use intervals to capture three kinds of uncertainties in the design of CPS: (1) implementation uncertainties, (2) the physical system uncertainties, and (3) control algorithm uncertainties. Unlike an over engineered system, modeling of uncertainties

in the form of intervals allows the designer to customize a control algorithm to operate correctly within its environment and reduce overall system cost.

Therefore, we advocate for the explicit support of intervals in the design phase of CPSs and within tools that support design of CPSs. To demonstrate the feasibility of tool support for intervals we show how intervals can be applied as part of a component-based design process. As an integral part of the component framework, we can aggregate and evaluate the uncertainties in the system to assess the suitability of a system in accomplishing its mission. We apply a homogenous data structure that processes properties, relations and assumptions of the algorithm, the physical system and the computation system in one graph, which automatically discovers design violations in the composed system.

Our experiments demonstrate the suitability of our approach to assess the system properties. In an example, we have shown how small changes in system properties have significant ramifications in the overall system assessment. The presented approach has the potential to bridge, at least part of the way, from the behavioral description of a CPS to an implementable system architecture, and to prove valuable in the early assessment of design alternatives to support automatic design space exploration of CPSs.

As future work we intend to extend the tool to synthesize implementable systems from a component database. In case of conflicts, properties should be identified and synthesized to make the system implementable. In the current version of our tool, we can identify and highlight violated constraints in the system and trace back to the properties directly responsible for the violations, however, the elimination of conflicts requires manual effort. We plan to support runtime adaptability of components and composition, as well as probabilistic intervals in the future.

## ACKNOWLEDGMENT

This work was supported in part by the National Science Foundation under NSF grant numbers 1016789 and 1136146.

## REFERENCES

[1] *Simulink - Simulation and Model-Based Design*, 2013. http://www.mathworks.com/products/simulink/.

[2] J. Carlson. Timing analysis of component-based embedded systems. In *15th ACM SIGSOFT symposium on Component Based Software Engineering*, 2012.

[3] A. Čaušević, C. Seceleanu, and P. Pettersson. Modeling and reasoning about service behaviors and their compositions. *Leveraging Applications of Formal Methods, Verification, and Validation*, 2010.

[4] W. Damm, A. Votintseva, A. Metzner, B. Josko, T. Peikenkamp, and E. Böde. Boosting re-use of embedded automotive applications through rich components. *in Foundations of Interface Technologies*, 2005, 2005.

[5] P. Derler, E. Lee, M. Törngren, and S. Tripakis. Cyber-physical system design contracts. In *ACM/IEEE International Conference on Cyber-Physical Systems*, 2013.

[6] E. Hyvonen. Constraint reasoning based on interval arithmetic. In *Proceedings of the 11th international joint conference on Artificial intelligence*, pages 1193–1198, 1989.

[7] K. Jamboti and P. Liggesmeyer. A framework for generating integrated component fault trees from architectural views. In *14th International Symposium on High-Assurance Systems Engineering (HASE)*, pages 114–121, Oct. 2012.

[8] J. Jensen, D. Chang, and E. Lee. A model-based design methodology for cyber-physical systems. In *Wireless Communications and Mobile Computing Conference (IWCMC), 2011 7th International*, pages 1666–1671. IEEE, 2011.

[9] G. Karsai and J. Sztipanovits. Model-integrated development of cyber-physical systems. *Software Technologies for Embedded and Ubiquitous Systems*, pages 46–54, 2008.

[10] L. Lednicki, J. Carlson, and K. Sandström. Model level worst-case execution time analysis for iec 61499. In *The 16th International ACM Sigsoft Symposium on Component-Based Software Engineering (CBSE)*, 2013.

[11] E. Lee and S. Seshia. *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*. LeeShia.org, 2011.

[12] M. Pajic, Z. Jiang, I. Lee, O. Sokolsky, and R. Mangharam. From verification to implementation: A model translation tool and a pacemaker case study. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2012.

[13] S. Peter, K. Piotrowski, and P. Langendörfer. In-network-aggregation as case study for a support tool reducing the complexity of designing secure wireless sensor networks. In *Proc. of the Third IEEE International Workshop on Practical Issues in Building Sensor Network Applications (SenseApp)*, 2008.

[14] M. Prochazka, R. Ward, P. Tuma, P. Hnetynk, and J. Adamek. A component-oriented framework for spacecraft on-board software. In *DASIA 2008 Data Systems In Aerospace*, volume 665, page 39, 2008.

[15] M. Sainz, J. Armengol, and J. Vehı. Fault detection and isolation of the three-tank system using the modal interval analysis. *Journal of Process Control*, 12(2):325–338, 2002.

[16] A. Sangiovanni-Vincentelli, W. Damm, and R. Passerone. Taming dr. frankenstein: Contract-based design for cyber-physical systems. *European journal of control*, 18(3):217–238, 2012.

[17] J. Sztipanovits. Composition of cyber-physical systems. In *Engineering of Computer-Based Systems, ECBS'07. 14th Annual IEEE International Conference and Workshops on the*, pages 3–6, 2007.

[18] A. Taherkordi, F. Loiret, R. Rouvoy, F. Eliassen, et al. Optimizing sensor network reprogramming via in-situ reconfigurable components. *ACM Transactions on Sensor Networks*, 9(2):1–37, 2013.

[19] B. Wang and J. Baras. Integrated modeling and simulation framework for wireless sensor networks. In *Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, 2012.