Ordering Events Based on Intentionality in Cyber-Physical Systems

Wajeb Saab, Maaz Mohiuddin, Simon Bliudze, Jean-Yves Le Boudec
École Polytechnique Fédérale de Lausanne, Switzerland
{firstname.lastname}@epfl.ch

***Abstract*—We consider cyber-physical systems (CPSs) comprising a central controller that might be replicated for high-reliability, and one or more process agents. The controller receives measurements from process agents, causing it to compute and issue setpoints that are sent back to process agents. The implementation of these setpoints causes a change in the state of the controlled physical process, and the new state is communicated to the controllers through resulting measurements. To ensure correct operation, the process agents must implement only those setpoints that were caused by their most recent measurements. However, in the presence of replication of the controller, network or computation delays, setpoints and measurements do not necessarily succeed in causing the intended behavior. To capture the dependencies among events associated with measurements and setpoints, we introduce the intentionality relation among such events in a CPS and illustrate its differences with respect to the happened-before relation. We propose a mechanism, intentionality clocks, and the design of controllers and process agents that can be used to guarantee the strong clock-consistency condition under the intentionality relation. Moreover, we prove that our design ensures correct operation despite crash, delay, and network faults. We also demonstrate the practical application of our abstraction through an illustration with a real-world CPS for electrical vehicles.**

I. Introduction

A. Motivation

We consider cyber-physical systems (CPSs) comprising software agents, namely controllers and process agents (PAs), that coordinate, as shown in Fig. 1, to maintain a physical process in a desirable state. A PA is a software agent that interfaces with sensors and actuators, and controls a sub-process. The controller receives measurements from PAs, computes and issues setpoints that are implemented by the PAs through actuators. The communication network might drop, delay, reorder or retransmit messages. The controller, being susceptible to crash and delay faults, is a single point of failure and is usually replicated for high-reliability. The same general architecture of CPSs is considered in recent papers [1]–[6].

On implementation of a setpoint by a PA, the state of the sub-process is altered and the new state is communicated to the controller via a measurement. The controller uses these measurements to recreate the new state of the entire process that it then uses to compute setpoints. The setpoints computed with this state are only valid as long as the recreated state reflects the actual state of the process. Any subsequent setpoint implementations change the process state, making the former setpoints *unsafe* for implementation. Therefore, to achieve the desired control, the state of the process at the time of setpoint implementation must be the same as that used by the controller for computing the corresponding setpoints. Thus, before implementing a setpoint, the PA must be able to ascertain whether the setpoint reflects the state it last advertised. In other words, it must be able to infer if a received setpoint was *caused by* the measurement it last advertised.

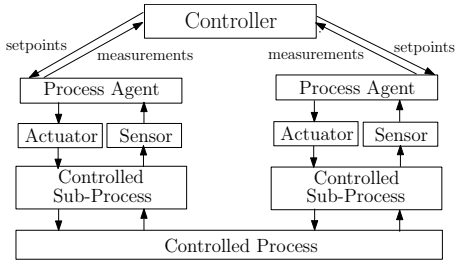

Fig. 1. CPS architecture with a central controller and one or more PAs

Similarly, a controller must be able to ascertain if a measurement received from a PA represents the most recent-state of that sub-process or the state corresponding to earlier setpoint-implementations. This causal relationship can be better understood using the notion of control rounds. Software agents must be able to attribute a round number to received messages, to compare it with the round number they are currently executing, and to treat the message appropriately.

Note that, we use the term "state of the process" as a proxy for the state of the PAs. While the state of the physical process is continuous and evolving, the state of the PAs is discrete and only changes upon a setpoint implementation. In real-time control, the CPS issues setpoints at a rate faster than the dynamics of the underlying process. Thus, the evolution of the state of the process between two setpoint implementations is minimal, thereby justifying our usage of the term.

B. Need for a New Relation

When the controller is replicated, assigning a consistent round number to events requires consensus between the replicas. Due to network losses and delays, and due to software faults, consensus might require unbounded time [7], making it unsuitable for real-time systems such as CPSs.

In literature, this problem is circumvented by using message labels (that represent the causal order between the messages) to infer the round number. The causal order between the messages is derived using the happened-before relation [8]. In the presence of replication of the controller, or random network or computation delays, messages that intend to cause

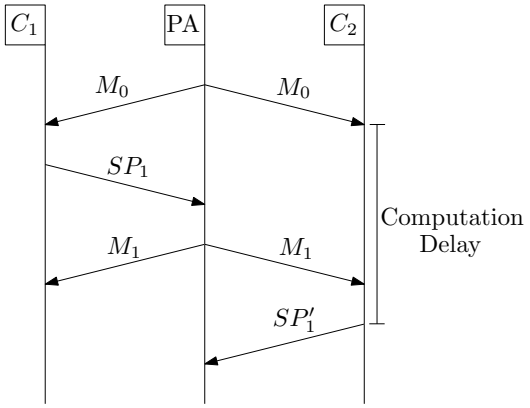

Fig. 2. Illustration of why the happened-before relation is not suitable for ordering events in a CPS in the presence of replication

a certain effect do not necessarily succeed, due to competing messages. This is illustrated through the example in Fig. 2.

In Fig. 2, the controller is replicated, and its two replicas C_1 and C_2 receive the measurement M_0 sent by the PA. This measurement is used by each replica in the computation of a setpoint, resulting in SP_1 and SP_1' in C_1 and C_2, respectively. In such a scenario, SP_1 and SP_1' belong to the same "generation" or the same "control round", and are said to be equivalent. SP_1 is received by the PA, and its implementation results in M_1. However, due to a delay at C_2, M_1 is received before SP_1' is issued. Although M_1 can be said to have happened before SP_1', M_1 is nonetheless caused by an equivalent of SP_1', namely SP_1.

In the previous example, we say that SP_1' intends to have caused M_1, but it did not succeed because a competing equivalent event (SP_1) was received earlier. In order to formally capture this phenomenon, we introduce the *intentionality* relation. This relation enables an implementation that provides a solution to the ordering problem.

C. Contributions

First, in Section III, we describe an abstraction of CPSs with one, possibly replicated, central controller, and one or more local PAs. This abstraction models the execution trace of a CPS by using two inherent relations: (1) the networking relation used to represent message exchange and (2) the computation relation used to represent the computation by software agents.

Second, in Section IV, we formally define the intentionality relation ($\rightarrow$, read intends) by using the networking relation and the computation relation. We also give an intuition for competing events and formally define this notion under the name, *intentional equivalence*. Moreover, we show that physical time, on its own, is inadequate in providing strong clock-consistency under the intentionality relation [8]. In other words, if physical time is used to timestamp and to order events a and b, where $TS(a)$ is the timestamp of event a, then $TS(a) < TS(b) \not\Leftrightarrow a \rightarrow b$.

Third, in Section V, we describe a mechanism, intentionality clocks, that uses logical clocks to attribute a round number to messages and can be used to guarantee strong clock-consistency under the intentionality relation. We formalize the desired behavior in terms of two correctness properties: *safety* and *optimal selection*. These properties specify which messages can be used in computation by software agents, and which can be discarded. We present the design of a controller and a PA that use intentionality clocks to guarantee these correctness properties. All guarantees are formally proven.

Lastly, we demonstrate a practical case study of the concepts developed throughout the paper by using, as an example, a real-world CPS for real-time control of electric vehicles (EVs) [4]. We analyze their design in the light of the intentionality relation and show that when the controller is replicated, it violates the optimal selection property. Consequently, the CPS can enter a deadlock situation which can be avoided by using the CPS design that we proposed.

II. Related Work

Here, we summarize the different bodies of work that addressed the problem of ordering events in distributed systems, and we note their shortcomings with respect to the intentionality relation. The detailed discussion of the shortcomings, along with examples, can be found in Sections I-B, IV-E and V-A. We also relate the CPS design we presented to other CPS designs found in literature.

Ordering events in distributed systems is traditionally done through a causal order, captured by the *happened-before* relation [8]. Providing a causal order adheres to what is referred to as the real-time causal consistency semantic [9]. This is achieved through one of several mechanisms such as timestamps, Lamport clocks [8], or vector clocks [10].

As previously mentioned (Section I-B) causal order and the happened-before relation fail to capture the inherent "round number" in CPSs. Therefore, we introduce the *intentionality* relation. We elaborate, in Section IV-E, on the issues that prevent physical time and timestamping-based labeling schemes from guaranteeing the strong clock-consistency condition under the intentionality relation.

Lamport clocks and vector clocks are complementary under the happened-before relation, as Lamport clocks describe this relation, and vector clocks infer it. Intentionality clocks use a scalar clock inspired from Lamport clocks. In Section V-A, we discuss the differences between intentionality clocks and Lamport clocks in greater detail. Vector clocks were not considered as an avenue because they only provide a partial ordering between events in general distributed systems, as not all labels are comparable.

We use the same model of the CPS as previous work [1]–[5], with a central, possibly replicated, controller and several PAs. Some previous works that use the same model [2], [4] assume the existence of a labeling scheme to achieve their goal. By providing one such labeling scheme in Section V-A, this paper complements the existing work in CPS design. We explore the CPS design in [4] in greater detail in Section VI; and we show that our system model, formalism, and mechanism apply to such a system. We also expose the possible issues in a naive extension of its design, which arise when the controller is replicated, and that can be avoided with our CPS design.

Several industrial solutions circumvent the ordering problem by using frameworks, such as the time-triggered architecture

(TTA) [11], which provide synchrony guarantees. However, using such frameworks requires specialized hardware, in addition to a complete redesigning of the application to fit the framework. In contrast, we propose a solution that requires neither. This facilitates deployment in existing CPSs.

III. CPS Model

In this section, we describe the model of the CPS we consider for defining the intentionality relation and for designing intentionality clocks. CPSs consists of four types of software agents: *sensors, actuators, controllers* and *PAs*. We consider CPSs with one central controller and one or more PAs, as shown in Fig. 1. The sensors and actuators interact with the physical process, by reading and altering its state, respectively. The software agents, namely controller and PAs, together achieve the desired control of the physical process. This model of the CPS is in agreement with the general model of CPSs considered in literature [1]–[6], and applies to a wide range of applications such as real-time control of smart grids, autonomous vehicles, and manufacturing processes.

PAs are low-level software agents responsible for controlling a sub-process, i.e., one part of the controlled process. PAs interface with the sensors and actuators as shown in Fig. 1. They implement the setpoints received from the controller through their actuators, read the state of the resources through sensors and send them as measurements to the controller. The number of PAs in a CPS is a constant and each PA is denoted with a unique identifier.

A controller performs high-level control of the physical process by receiving measurements from PAs and by sending setpoints to PAs.

We consider crash and delay faults [1] in all the software agents, namely controller and PAs. Byzantine faults are not considered. Being a single point of failure, the controller is often replicated for high-reliability. We assume that the PAs are not replicated. Hence, a software agent is either a controller replica or a (non-replicated) PA. All replicas of the controller have different identifiers. Furthermore, we assume that the communication network between the software agents might drop, delay, reorder or duplicate messages. Byzantine contamination of messages is not considered.

We abstract the execution of a CPS as a trace of events occurring on different software agents. We define three types of events that can occur on a software agent: *sending event*, *reception event*, and *timeout event*. When software agent A sends a message to agent B, we say that A experiences a sending event. Upon successful reception of the message by B, we say that B experiences a reception event. Whereas, if the message is lost or is delayed beyond a deadline, B experiences a timeout event. A timeout event could also be caused by the internal logic of a software agent, such as firing of a timer (as seen in Section VI) or a response to system state.

An event is represented by the 4-tuple (sa, pa, m, l) where (1) sa is the identifier of the agent on which the event occurs, (2) pa is the identifier of either the PA on which the event occurred, or the PA for which the event is intended, (3) m is the message encapsulated in the event, and (4) l is an event label given by the software agent on which the event occurs. For sending or reception events, the encapsulated message is the measurement or setpoint exchanged, and timeout events encapsulate $\perp$, representing the absence of a message.

Let $\mathcal{C}, \mathcal{P}$ be the set of identifiers of all controller replicas and PAs, respectively. Then, the set of identifiers of all software agents is $\mathcal{S} = \mathcal{C} \cup \mathcal{P}$. Let $\mathcal{M}$ be the set of all messages such that $\perp \in \mathcal{M}$, and $\mathcal{L}$ be a partially ordered set of labels. Then, we denote the set of all events that occur in an execution trace by $\mathcal{E} \subset \mathcal{S} \times \mathcal{P} \times \mathcal{M} \times \mathcal{L}$. No two distinct events have the same 4-tuple (sa, pa, m, l). Furthermore, we require that the abstract labeling scheme used to obtain $\mathcal{L}$ ensures that labels of events occurring on the same software agent, for the same PA, are different. In practice, such a labeling of events is achieved through physical timestamps, a permanent sequence numbering scheme, Lamport clocks [8], Vector clocks [10], etc. Also, for CPSs that do not implement any labeling mechanism on events, the model still applies by successively numbering all events of each software agent with increasing integers.

Sending events are considered as *output events*, as they are the output of a computation at a software agent. Reception and timeout events are considered *input events*[1]. The set of input events $\mathcal{E}^i$, which includes reception and timeout events, and the set of output events $\mathcal{E}^o$, which includes sending events, are such that $\mathcal{E} = \mathcal{E}^i \cup \mathcal{E}^o$ and $\mathcal{E}^i \cap \mathcal{E}^o = \emptyset$. Note that, due to network retransmissions, a single output event can result in different input events at the same PA, as each of these input events will have different labels l.

We consider the following computation model of a controller. In each computation, the controller uses exactly one input event from each PA and produces exactly one output event for each PA. Moreover, when the controller computes by using timeout events for one or more PAs, it is able to appropriately account for the missing information that it would have received from the corresponding reception events. Else, the controller refrains from computation of setpoints until more reception events occur (measurements are received).

To bootstrap the CPS, we assume that a controller starts with p sending events, one for each PA. These events are called *initial sending events*. The set of all initial sending events is represented by $\mathcal{I}$.

IV. Relations Between Events in a CPS

In this section, we formalize the notion of intentionality, an intrinsic relation between events in a CPS which captures the order between measurements and setpoints. First, we define the sub-relations that constitute intentionality, namely, the *network relation* and the *computation relation*. Then, in Section IV-C, we define an equivalence relation between events, called as *intentional equivalence*. Finally, we define the intentionality relation in Section IV-D.

For a relation $\xrightarrow{\mathrm{r}}$ and an event a, we denote by $r(a)$ and $r^{-1}(a)$ the image and pre-image of a by $\xrightarrow{\mathrm{r}}$, respectively.

[1]This dichotomy of input-output events is similar to that of sending-receiving events used in classic distributed systems literature. We use a different name because we also have timeout events in our model.

A. Network Relation

Software agents exchange messages using a communication network. Thus, a network relation ($\xrightarrow{\text{n}}$) exists between events at different agents. This relation maps an output event (sending event) at one agent to an input event (reception/timeout event) at another agent. Formally, we abstract the properties of a network relation as follows.

Definition 1 (Network Relation)**.** *$\xrightarrow{n}$ is a network relation, iff $\xrightarrow{n} \subset \mathcal{E}^o \times \mathcal{E}^i$ and*

- *for any $a \in \mathcal{E}^o$, there exists $b \in \mathcal{E}^i$ s.t.*
  1) $a \xrightarrow{n} b, b.pa = a.pa$, *and* $n^{-1}(b) = \{a\}$
  2) *If* $a.sa \in \mathcal{C}$, *then* $b.sa = a.pa$
  3) *If* $a.sa \in \mathcal{P}$, *then* $b.sa \in \mathcal{C}$
- *for any $b \in \mathcal{E}^i$, there exists $a \in \mathcal{E}^o$ s.t. $n^{-1}(b) = \{a\}$*

Intuitively, $a \xrightarrow{\text{n}} b$ if a is a sending event and b is its corresponding reception event or the corresponding timeout event that occurs on the intended destination. Notice that for a sending event that occurs on a controller, the corresponding input event occurs on a PA, and for a sending event that occurs on a PA, the corresponding input event occurs on a controller.

B. Computation Relation

A computation performed by a software agent can be represented as a mapping from a set of input events to a set of output events. In each computation, a controller uses p measurements, one from each PA and computes p setpoints, one for each PA. Upon reception of a setpoint, a PA implements it through the actuator, then reads the state of the sensor and sends the new state as a measurement. The set of input events used by a PA for computation is a singleton set. We abstract the properties of a computation relation ($\xrightarrow{\text{c}}$) as follows.

Definition 2 (Computation Relation)**.** *$\xrightarrow{c}$ is a computation relation, iff $\xrightarrow{c} \subset \mathcal{E}^i \times \mathcal{E}^o$*

1) *for any $a \in \mathcal{E}^i$*
   a) *If $a.sa \in \mathcal{P}$, then $\exists b \in \mathcal{E}^o : a.sa = b.sa$, $a.pa = b.pa, c(a) = \{b\}$, and $c^{-1}(b) = \{a\}$*
   b) *If $a.sa \in \mathcal{C}$, then*
      - $\forall\ i \in \mathcal{P},\ \exists!\ b \in c(a) : b.pa = i$
      - $\forall\ b \in c(a), b.sa = a.sa$
      - $\forall\ b, b' \in c(a), c^{-1}(b) = c^{-1}(b')$
      - $\forall\ b \in c(a),\ \forall\ i \in \mathcal{P},\ \exists!\, a' \in c^{-1}(b) : a'.pa = i$
2) *for any $a \in \mathcal{E}^o \setminus \mathcal{I}$, there exists $b \in \mathcal{E}^i$ s.t. $b.pa = a.pa$ and $b \in c^{-1}(a)$.*

C. Intentional Equivalence Relation

In the presence of controller replication and message retransmission, certain events that occur in the same "control round" in a CPS are functionally the same, i.e. they steer the physical process to a similar state. The *intentional equivalence* relation ($\equiv$) captures this.

We list the properties that will be used to define this relation. The intentional equivalence relation is defined as the smallest relation satisfying the following properties.

1) If a and b are the initial sending events at controller replicas, and $a.pa = b.pa$ then $a \equiv b$
2) If $a \xrightarrow{\text{n}} b$ and $a \xrightarrow{\text{n}} c$, then $b \equiv c$
3) If $a \equiv \tilde{a}$, $a \xrightarrow{\text{n}} b$ and $\tilde{a} \xrightarrow{\text{n}} c \implies b \equiv c$
4) If $a \xrightarrow{\text{c}} b$, $a \xrightarrow{\text{c}} c$ and $b.pa = c.pa$, then $b \equiv c$
5) If $a \equiv \tilde{a}$, $a \xrightarrow{\text{c}} b$, $\tilde{a} \xrightarrow{\text{c}} c$ and $b.pa = c.pa \implies b \equiv c$

The intuition behind rule (1) is that initial sending events are computed without any knowledge of the state of the system. They are polling events and do not steer the system in any direction. Thus, those sent to the same PA are equivalent.

For rule (2), the underlying intuition is that reception events and their corresponding timeout events convey similar information to the controller, as do multiple reception events corresponding to the same sending events (i.e., retransmission). A reception event informs the controller of the state of the process, whereas the corresponding timeout event forces the controller to estimate the missing state before computation. Recall from Section III that, when a controller of a CPS computes using timeout events, it accounts for the missing information, in order to ensure correctness. Thus, reception events and corresponding timeout events are equivalent.

From Definition 2, there exists a single output event resulting from a computation relation for each PA. Rule (4) states that if an event causes, after computation, two events for the same PA, then the resulting events are equivalent. In fact, the resulting events are the same event, and are therefore equivalent by the reflexive property of the intentional equivalence relation (Theorem IV.1). Finally, rules (3) and (5) mean that equivalent events, when subject to the same relation, result in equivalent events.

To better understand the intuition behind equivalent events resulting in similar changes to the state of a CPS, consider a controller that receives partial information from its PAs. A well-designed controller would compute only if it can reconstruct the missing information from the partially received information, thus resulting in safe setpoints. For example, a controller with two PAs, one with a 1 Hz update rate and another with a 10 Hz update rate, would require a measurement from the latter every 100 ms, whereas from the former only once every 1 s as it knows that the state of the slow PA has not changed during that second. Thus, a timeout event on the measurement from that PA is equivalent to a reception event.

Events that result from retransmissions encapsulate the same message verbatim, and are thus deemed equivalent. In the presence of replication, however, different replicas might send different sets of setpoints in a given "control round". This might be due to different sets of measurements received, a different internal state, or a non-deterministic computation. In other words, CPS equivalent events can be different and can lead to different outcomes. Therefore, in practice, CPSs must be able to live with this. This can be done, for instance, with an agreement on input as shown in our companion work, Quarts [2]. Alternatively, this is achieved if the setpoints are idempotent and the CPS logic permits such deviations [12].

The following theorem states that the intentional equivalence is, indeed, an equivalence relation.

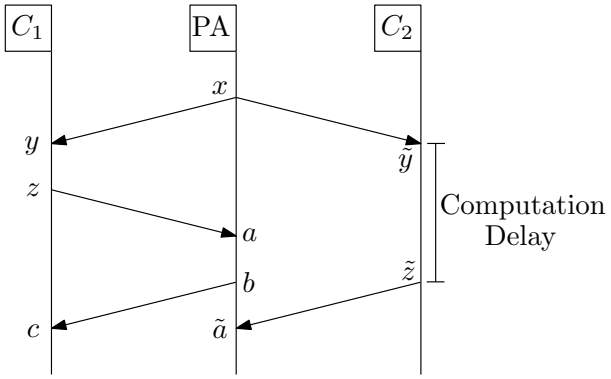

Fig. 3. Difference between temporal order and intentionality due to replication of the controller

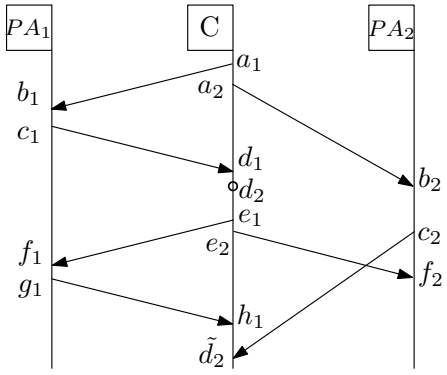

Fig. 4. Difference between temporal order and intentionality due to delays

Theorem IV.1. *"Intentional equivalence" is reflexive, symmetric and transitive.*

D. Intentionality Relation

We define the intentionality relation ($\rightarrow$), where $a \rightarrow b$ is read as "a intends b", by using the relations defined in the previous sections. It is the smallest relation satisfying the following properties:

1) If $a \xrightarrow{\text{n}} b$, then $a \rightarrow b$
2) If $a \xrightarrow{\text{n}} b$ and $\tilde{a} \equiv a$, then $\tilde{a} \rightarrow b$
3) If $a \xrightarrow{\text{c}} b$, then $a \rightarrow b$
4) If $a \xrightarrow{\text{c}} b$ and $\tilde{a} \equiv a$, then $\tilde{a} \rightarrow b$
5) If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$

Based on these properties, the following theorems hold.

Theorem IV.2. *For any two events a, b:* $a \rightarrow b \implies b \not\rightarrow a$.

Theorem IV.3. *For any two events a, b, such that* $a.pa = b.pa$*:* $(a \not\rightarrow b \wedge b \not\rightarrow a) \iff a \equiv b$.

The proofs of Theorems IV.1, IV.2, and IV.3 are not central to the discussion, and are thus not included for spatial consideration. We discuss them further in the technical report [13].

Let $\mathcal{E}_p = \{e \in \mathcal{E} | e.pa = p\}$. Recall that $\mathcal{E}_p/\equiv$ represents the factorization of the set $\mathcal{E}_p$ by the relation $\equiv$. Then, as a consequence of Theorem IV.3, the intentionality relation induces a total order on $\mathcal{E}_p/\equiv$, for any $p \in \mathcal{P}$. In other words, the intentionality relation induces a total order on any set of events concerning the same PA and belonging to different equivalence classes.

In Section V, we use the definition of intentionality to formally specify the desirable correctness properties of a CPS. We also present a CPS design and prove that it guarantees the said properties.

E. Intentionality and Physical Time

In this section, we answer the question "Is physical time sufficient to guarantee the strong clock-consistency condition under the intentionality relation?"

As CPSs are real-time systems, they generally keep track of physical time. To maintain synchronized global time on all software agents, their physical clocks are synchronized either using GPS-based clock-synchronization or network-based clock-synchronization (e.g., PTP [14], NTP [15]). These time-synchronization solutions provide a synchronization accuracy δ that ranges from sub-microsecond to one millisecond.

CPSs often leverage the availability of synchronized physical clocks to reason about the temporal ordering of events. However, as we will see in the following examples, the temporal ordering of events does not coincide with intentionality.

Consider a CPS with two controller replicas and a single PA, as shown in Fig. 3. We will consider a perfect time-synchronization ($\delta = 0$). Here, $a \rightarrow b, a \equiv \tilde{a} \implies \tilde{a} \rightarrow b$. However, the time of occurrence of the effect b, is less than that of the "cause" $\tilde{a}$.

Another example emphasizing the difference between temporal order and intentionality, shown in Fig. 4, concerns a CPS with one non-replicated controller and two non-replicated PAs. Due to network delay, the reception event b_2 occurs on PA_2 much later than the reception event b_1 at PA_1. As a result, the reception event d_1 from PA_1 occurs much earlier than the corresponding reception event $\tilde{d}_2$ from PA_2. Instead, the controller moves on with a timeout event d_2 for PA_2. Consequently, the events e_1, f_1, g_1 and h_1 take place. Then, we have $a_2 \rightarrow d_2 \rightarrow e_1 \rightarrow h_1$. But, $d_2 \equiv \tilde{d}_2 \implies \tilde{d}_2 \rightarrow h_1$. However, on the controller, the time of occurrence of h_1 occurred is less than that of $\tilde{d}_2$. Therefore, as the correct temporal order is h_1 before $\tilde{d}_2$, it does not coincide with intentionality.

Hence, we conclude that on their own, physical clocks are not sufficient to reason about intentionality and require an additional mechanism to do so. In Section V-A, we present a mechanism that describes the intentionality relation by using logical clocks instead of physical clocks. Note that, such a mechanism does not require synchronized physical time.

V. CPS Design

As discussed in Section I, the controller and PAs must ensure that the events used in their computation reflect the most recent state of the process. The state of the process changes with each setpoint implementation, i.e., a computation event by a PA. Thus, when a controller uses an input event $e : (sa_1, pa_1, m_1, l_1)$ and there exists another event $e' : (sa_1, pa_1, m_2, l_2)$ such that $e \rightarrow e'$, then the message m_2 reflects a more recent state of the sub-process controlled by PA_1 than the message m_1. If the controller uses the "old" event e in the computation of setpoints, then the resulting

setpoints might not be compatible with the most-recent state of the process, thereby causing incorrect control of the CPS.

Definition 3 formally specifies the *safety* property that the software agents in a CPS must satisfy in order to guarantee correct control.

Definition 3 (Safety)**.** *If a software agent uses an event a for computation, then the last sending event b that occurred on this software agent, where $b.pa = a.pa$, is such that $b \rightarrow a$.*

The safety property requires that (1) a PA must use only those events in computation that have accounted for its most recent state, captured by its last measurement sending event, and (2) a controller must use only those events in computation that reflect the state change caused by its last setpoint sending events.

Notice that discarding all messages can trivially satisfy the safety property. However, this will render the physical process uncontrolled by a complete loss of availability of the control. Therefore, the selection of events must be optimal, i.e., only those events that violate the safety property must be discarded. This property is formally defined as follows.

Definition 4 (Optimal Selection)**.** *An event a must only be discarded if there exists another event b, such that $b \not\rightarrow a$ that: (1) occurred on this software agent, or (2) this software agent was informed that b occurred on its replica.*

The first part of the optimal selection property, on its own, states that an event a must be accepted by a software agent if the last sending event b on this software agent intended to cause a. Recall that if $b \rightarrow a$, then all events that occurred on this software agent before b also intend to cause a. Thus, a presents new information, i.e., information about the state of the process after the implementation of the last-sent setpoint. In the presence of replication, however, an event that was intended by the last sending event, does not necessarily present new information, as seen in the following example.

A controller replica that last computed in round l might receive two measurements from the same PA: one from round $l{+}1$ and another from round $l{+}2$, before it is ready to compute. This can occur due to another controller replica driving the CPS into round $l{+}2$. In this scenario, the first controller might ignore the message from round $l{+}1$ as the message from round $l+2$ supersedes it. The controller will, therefore, compute using reception events from round $l+2$, and declare timeout events for PAs from which it has not received measurements from that round. This condition is captured by the second part of the optimal selection property.

Next, we describe the design of a CPS in which the software agents satisfy the safety and optimal selection properties. Our design uses a label scheme, intentionality clocks, that is adapted from Lamport clocks to guarantee strong clock-consistency under the intentionality relation. We describe the design of intentionality clocks in Section V-A; and the design of controllers and PAs that use the intentionality clocks in Sections V-B and V-C, respectively. We present the formal guarantees in Section V-D.

A. Intentionality clocks

Intentionality clocks is a mechanism to maintain, update and synchronize logical clocks across all software agents in a CPS. It is adapted from the Lamport clocks abstraction that was designed for general distributed systems, to accommodate the specificities of events in CPSs.

Each agent maintains and updates a local logical clock that is used to set the event label of sending events on that agent. These labels are communicated along with the message encapsulated in the event and are used to obtain the event labels of the corresponding reception and timeout events.

The labels are assigned such that they guarantee the strong clock-consistency condition [8]. In other words, for two events a and b, we say that a clock mechanism *describes* the intentionality relation if the event labels are such that $a.l < b.l \implies a \rightarrow b$ and $a.l = b.l \implies a \equiv b$. We say that the clock mechanism *infers* the intentionality relation if $a \rightarrow b \implies a.l < b.l$ and $a \equiv b \implies a.l = b.l$. The strong clock-consistency condition is satisfied if a clock mechanism both describes and infers the intentionality relation.

The design of intentionality clocks is given by the following rules (for details, see Sections V-B and V-C):

1) The event label of a sending event at an agent is the value of its logical clock, right before the sending event.
2) If a is a sending event and b is its corresponding reception or timeout event, then $b.l = a.l + 1$.
3) The logical clock of an agent is only incremented before a computation. It is never decremented.
4) The logical clock of delayed software agents is resynchronized using the labels of the received events.

There are two main distinctions in the design of intentionality clocks when compared to Lamport clocks [8]. First, in our solution, the logical clock at an agent is incremented only when the agent performs a computation. This enables the controller to infer the intentionality relation by using its local logical clock, and to have a notion of the "control round". In contrast, the Lamport clock at an agent is updated after every event that occurs at that software agent Thus, the agents lose the information required to infer the intentionality relation and to have a notion of the "control round".

Second, in intentionality clocks, the label of a reception event is one more than the label of the corresponding sending event. In contrast, in Lamport clocks, for a reception event b that occurs when the value of the logical clock of the agent is C is $b.l = \max(a.l, C) + 1$, where a is the corresponding sending event. Due to the presence of delayed controller replicas or message retransmissions by the network, reception events from previous "control rounds" might have a higher label than reception events from current "control round". Consequently, it cannot describe or infer the intentionality relation, i.e., cannot guarantee the strong clock-consistency condition.

The formal guarantees of intentionality clocks are presented in Section V-D and proven in the Appendix.

B. Controller Design

Algorithms 1 and 2 describe the design of a controller with intentionality clocks; this design satisfies the safety and optimal selection properties. The model of the controller is the same as that used in our previous work [1], [2]. The parts in red are our modifications for satisfying the aforementioned properties (together with the implementation of Algorithm 3).

Each controller maintains a logical clock C, a list of input events (i.e., reception and timeout events) $\mathbf{Z}$, and a list of their corresponding event labels $\mathbf{L}$.

Upon receiving a measurement from a PA, the controller declares a reception event with a label one more than the label of the received message (Algorithm 1, line 9). The controller adds the measurement to $\mathbf{Z}$, and adds the label of the reception event to $\mathbf{L}$.

The controller also occasionally checks if it has accumulated enough information about the state of the physical process, through the measurements, required to compute setpoints. To this end, it uses the `ready_to_compute()` function. We make no assumptions on this function or the frequency with which the controller invokes it. The controller can choose to start a computation of setpoints by considering any form of information provided by measurements from $\mathbf{Z}$. Moreover, the labels in $\mathbf{L}$ provided by intentionality clocks expose additional information to the `ready_to_compute()` function, by giving insight into the intentionality relation between the events.

Then, the controller computes setpoints by calling the `compute()` function that uses $\mathbf{Z}$, $\mathbf{L}$ and C as shown in Algorithm 2. In this function, we specify how the controller must use the logical clock and the labels in order to satisfy the safety and optimal selection properties. First, the controller computes the highest label of the events it has seen: C'. This represents the most recent events the controller has encountered. Then, for all PAs from which the highest label of received events is less than C', the controller declares a timeout event (line 4), thereby explicitly acknowledging that it lacks the most recent information from this PA. The controller can then account for this missing information in the subsequent computations. The logical clock C is set as $C'+1$ to mark the computation operation and the resulting setpoints are issued by way of sending events with the label being the current logical clock.

Note that, in Algorithm 2, the computation of setpoints takes as input a set of input events comprising exactly one reception or timeout event from each PA. This is in accordance with the computation relation ($\xrightarrow{\mathrm{C}}$) described in Section IV-B.

The controller is designed to be soft state [16]. When a controller boots or reboots after a crash, its logical clock is set to zero, and the lists $\mathbf{Z}$ and $\mathbf{L}$ are reinitialized. Thus, it would use all subsequent reception or timeout events for computation, because their labels would be ≥ 0. This behavior is in accordance with the safety property, as a freshly rebooted controller de-facto has no last setpoint sending event. However, as described in Section V-C, these sending events would be disregarded by the PAs as they do not reflect their most recent state. Note that, upon booting or rebooting, the controller sends setpoints corresponding to the initial sending events, indicated by S_0, with the label 0.

Algorithm 1: Abstract model of a controller

```
1  on boot or reboot
2      C ← 0;  // CPS clock on this controller
3      Z ← {}; // List of input events
4      S ← S_0; // List of setpoints to be sent
5      L ← {}; // List of labels of input events
6      Issue S with label C
7  end;
8  on reception of a message m with label l from a PA i
9      Declare reception event a : (sa, i, m, l + 1);
10     Add m to Z;
11     Add a.l to L;
12 end;
13 repeat
14     decision ← ready_to_compute(C, Z, L);
15     if decision then
16         S, C, Z, L ← compute(C, Z, L);
17         Issue S with label C; // Sending events
18     end
19 forever;
```

Algorithm 2: Function: compute($C, \mathbf{Z}, \mathbf{L}$)

```
1  C' ← max(C + 3, max(L));
2  for each PA i do
3      if the maximum label in L from PA i is not equal to C' then
4          Declare timeout event a : (sa, i, ⊥, C');
5          Add ⊥ to Z;
6          Add a.l to L;
7      end
8  end
9  C ← C' + 1;
10 S ← setpoints computed using measurements with label C';
11 Return S, C, Z, L;
```

From lines 1 and 9 in Algorithm 2, we see that the logical clock of a newly booted controller is re-synchronized with that of the other software agents before computation, by taking the maximum of the labels of the received measurements. This is discussed further, in Section V-C.

C. PA Design

Algorithm 3 describes the design of a PA with intentionality clocks; the design complements Algorithms 1 and 2 in satisfying the safety and optimal selection properties. Each PA maintains a clock C that is initialized with 0 upon booting.

Upon reception of a setpoint from a controller, the PA declares a reception event with a label one more than that received in the message. Then, the PA compares the label of the event with its local logical clock. If the reception event has a higher label, then the PA implements the setpoint, else it is discarded because it violates the safety property. In other words, a reception event with a label less than the logical clock of the PA means that the corresponding sending event was not computed with the most recent state of this PA. In this way, setpoints from delay-faulty controllers or freshly booted controllers are not implemented, thereby upholding the safety property.

After implementing the setpoint through an actuator, the PA increments its logical clock to mark the computation of a new measurement. The PA computes a new measurement through a sensor and sends the measurement to the controller by a sending event labeled with the current logical clock.

Each PA stores its logical clock before computing measurements. When a PA recovers after a crash, it initializes

Algorithm 3: Model of PA with causal clocks.

```
1  on boot
2  |  C ← 0;
3  end;
4  on reboot
5  |  C ←stored C;
6  end;
7  on reception of a message m with a label l from a controller
8  |  Declare reception event a : (pa, pa, m, l + 1);
9  |  if C < a.l then
10 |  |  C ← a.l;
11 |  |  Implement setpoint;
12 |  |  C ← C + 1;
13 |  |  Store C;
14 |  |  Compute measurement;
15 |  end
16 |  Send measurement to the controller with label C;
17 end;
```

its logical clock to the last stored value[2]. In this way, the PA keeps track of the last state it advertised to the controller.

Controllers and PAs update their local logical clocks to reflect the labels they observe (Algorithm 2 line 1, Algorithm 3 line 10). Notice that, upon receiving a setpoint with a label lower than its logical clock, a PA also sends both the latest computed measurement and the current value of the local clock. This serves to re-synchronize the software agents that miss some control rounds due to messages losses, crashes and recoveries, or delays.

D. Formal Guarantees

We formally prove that our mechanism of intentionality clocks and Algorithms 1, 2, and 3 guarantees safety and optimal selection. The first step lies in proving that intentionality clocks under our mechanism infer and describe the intentionality relation.

Theorem V.1 (Strong Clock-Consistency)**.** *In a CPS that implements Algorithms 1, 2, 3: for any two events a and b,*

$$C(a) < C(b) \iff a \to b$$

$$C(a) = C(b) \text{ and } a.pa = b.pa \iff a \equiv b$$

Proof. The proof can be found in Appendix A. □

Theorem V.2 (Safety & Optimal Selection)**.** *A CPS that implements Algorithms 1, 2 and 3 guarantees safety and optimal selection.*

Proof. The proof can be found in Appendix B. □

VI. Case Study: CPS for Scheduling Electric Vehicle Charging

We present, via a case study, a practical application of the intentionality relation and the intentionality clocks mechanism. We take the example of a CPS for scheduling the charging of a fleet of electric vehicles (EVs), which provides a schedule that accounts for both the vehicles' demand and the vehicle-to-grid regulation services [4].

[2]Constantly increasing counters might cause a counter overflow. However, a 64-bit counter incremented once every millisecond takes much longer than the lifetime of any CPS to wrap-around.

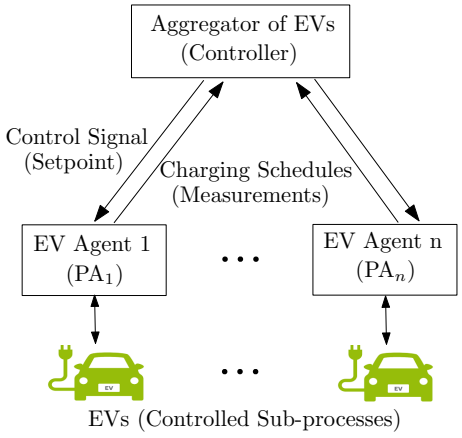

Fig. 5. Architecture and information flow of a CPS for EV charging [4]

One of the goals of the CPS is to charge each vehicle based on the vehicle's demand, and the other goal is to provide frequency support for the grid. Modulating the charging schedule of the EVs, by charging at a higher or lower rate, or even discharging into the grid for some time in cases of downward excursion of frequency, can provide frequency support. In practice, the EVs must respond to regulation requests every two to four seconds [17].

The paper [4] presents a solution to the problem assuming an ideal communication, and without considering the failure of the software agents. However, such a mission-critical control requires high levels of reliability in a real deployment. It is, therefore, desirable to replicate the controller.

In Section VI-B, we use the intentionality relation to analyze their CPS design for possible issues that could arise when the controller is replicated. We find that a naive extension to their design violates the optimal selection property. Consequently, due to software and network faults, the CPS can encounter a deadlock situation, whereby frequency support will no longer be achieved. This might result in the instability of the underlying electrical grid.

A. CPS Model

In this section, we analyze the system model of [4] and show how our system model, presented in Section III, applies.

Figure 5 presents an architectural view of the system consisting of EVs and the aggregator, as shown in [4]. It comprises EVs that represent the controlled sub-process from our model, the EV agents labeled 1 through n that correspond to the PAs from our model and the central aggregator that is the controller of the CPS. The controller receives a charging/discharging (henceforth referred to as charging) schedule from each PA, which represents the measurements in our model. The controller computes control signals (setpoints) for each of the EVs, using the received charging schedules.

The algorithms of the controller and PA used for achieving the desired vehicle-to-grid regulatory service are described in [4] in Algorithms 2A and 2B. We abstract this process and summarize it, as shown in Algorithm 4. The controller periodically starts an iterative process that consists of several control rounds, subject to convergence of the algorithm. Each iterative process is independent from the previous one, as

if the controller had a fresh start. The triggering of a new iterative process (line 2) is a timeout event in our model, which further causes n sending events, one for each PA (line 4). These represent the initial sending events as they are computed without using measurements.

From Algorithm 4, we see that the CPS in [4] uses a labeling scheme with label m. In Section VI-B, we show that this labeling scheme violates optimal selection (Definition 4), consequently the system can enter a deadlock situation. In line 6, we see that the controller waits for schedules from each PA before beginning computation in line 7. Thus, the `ready_to_compute()` function of this CPS is the presence of one reception event from each PA, corresponding to the current round with label m. Timeout events do not occur within an iteration.

In lines 7-13, we see the computation relation takes as input one schedule from each PA and produces one control signal for each PA. In other words, the computation relation takes as input one reception event from each PA and outputs one sending event for each PA. This is same as our computation relation (Definition 2).

The iterative process (lines 5-13) continues until the control has converged. It terminates with the sending of control and stop signals in the setpoints of the last computation.

Each EV agent (PA) also keeps track of the on-going control round by using the indicator m. It only accepts charging schedules (setpoints) that belong to the current round (line 20). Upon receiving a setpoint from the current round, the PA checks if it is accompanied with a stop signal (line 22). The presence of a stop signal is an indication of the termination of the iterative process and results in implementation of the setpoint. Alternatively, when the stop signal is absent, the PA sends its new charging schedule as a measurement to the controller and increments m by one.

Algorithm 4: Abstraction of the iterative process of scheduling EV charging [4]

```
1  At the Aggregator (Controller)
2  repeat periodically
3      m ← 0;
4      Send a request for schedules with label m to each PA;
5      repeat
6          if schedules labeled m from each PA are received then
7              Perform computation of control signals;
8              m ← m + 1;
9              if control has not converged then
10                 Send new control signals with label m to each PA;
11             end
12         end
13     until control has converged;
14     Send control signals and stop signals with label m to each PA;
15 forever;
16
17 At the EV Agent (PA)
18 m ← 0;
19 on reception of a control signal with label k from aggregator
20     if k == m then
21         Compute new charging schedule;
22         if stop signal received then
23             m ← 0;
24             Implement charging schedule until next control signal;
25         else
26             Send schedule to aggregator with label m;
27             m ← m + 1;
28         end
29     end
30 end;
```

B. Deadlock due to Violation of Optimal Selection

Here, we describe a scenario in which the CPS in [4] can enter a deadlock situation when the controller is replicated, due to the controller replicas having different round indicators.

Consider a scenario with two replicas of the controller C_1 and C_2, with indicator m_1 and m_2, respectively. Consider a PA, PA_0 with indicator m_0. Consider a situation when both controllers sent out setpoints to PA_0 with label 5. Then, $m_1 = m_2 = 5$. PA_0 receives this setpoint, implements it, sends a measurement with label $m_0 = 5$ and increments its indicator to 6. Thus, after this round, we have $m_1 = m_2 = 5$ and $m_0 = 6$. Now, if C_2 does not receive the measurement from PA_0 and C_1 received measurements from all PAs, C_1 will begin computation and C_2 will be stalled. After this computation, we have $m_1 = 6$ and $m_2 = 5$. Moreover, PA_0 implements the setpoint and increments its indicator m_0 to 7.

Next, let C_1 crash due to a software failure and reboot. Then, it requests for schedules with $m_1 = 0$. However, as PA_0 is expecting messages from round 7, it ignores these requests. PA_0 also discards any messages from C_2, as they will have a label 5. Moreover, C_2 discards the received measurements with label 6 because its label is $m_2 = 5$. This is a violation of the optimal selection property by C_2. Hence, the CPS enters a deadlock state because there is no mechanism to resynchronize the counters.

Our design (Section V) satisfies the optimal selection property even in the presence of controller replication, and software and network faults (as shown in Theorem V.2). Therefore, the problem encountered by the design in [4] can be avoided by applying intentionality clocks and tuning the design of the software agents according to Algorithms 1, 2, and 3.

VII. Conclusion and Future Work

We address the problem of enabling software agents in a CPS, namely controller and PAs, to provide a notion of rounds of computation in presence of network losses or delays, or replication of the controller. We show, that in such settings, the causal (or happened-before) relation, traditionally used in distributed systems literature, does not enable capturing the control rounds. Instead, we introduce a new relation that we call "intentionality relation". We formally define this relation for CPSs with one, possibly replicated, central controller and one or more PAs. A possible avenue for future work is to extend the intentionality relation to a wider range of CPS, specifically, comprising a hierarchy of controllers, or spontaneous sensors that send out-of-band measurements to the controllers.

We present a clock mechanism, intentionality clocks, that can be used to both describe and infer the intentionality relation. We also formalize the correctness properties, namely safety and optimal selection, that describe how the agents must treat events in order to respect intentionality. We present the design of a controller and a PA that guarantees the correctness

properties. Lastly, through a case study of a real-world CPS for charging EVs, we demonstrate the practical relevance of the introduced concepts.

In future work, we intend to extend our design to more generic CPSs as mentioned above. Moreover, we are currently in the process of implementing our controller and PA design in a CPS for real-time control of electric grids [6]. We will study the impact of violations of the correctness properties (safety and optimal selection) on the physical process, through experiments in a virtual commissioning environment [18]. This enterprise is a preparation for the deployment of the said CPS for real-time control of a medium-voltage grid.

VIII. Acknowledgments

This research was supported by the "SNSF - NRP 70" Energy Turnaround project.

References

[1] Maaz Mohiuddin, Wajeb Saab, Simon Bliudze, and Jean-Yves Le Boudec. Axo: Masking Delay Faults in Real-Time Control Systems. In *Industrial Electronics Society, IECON 2016-42nd Annual Conference of the IEEE*, pages 4933–4940. IEEE, 2016.

[2] Wajeb Saab, Maaz Mohiuddin, Simon Bliudze, and Jean-Yves Le Boudec. Quarts: Quick Agreement for Real-Time Control Systems. In *22nd IEEE International Conference on Emerging Technologies And Factory Automation*. IEEE, 2017.

[3] Edward A Lee. Cyber Physical Systems: Design Challenges. In *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pages 363–369. IEEE, 2008.

[4] Junhao Lin, Ka-Cheong Leung, and Victor OK Li. Optimal Scheduling with Vehicle-to-Grid Regulation Service. *IEEE Internet of Things Journal*, 1(6):556–569, 2014.

[5] Andrey Bernstein, Niek Bouman, and Jean-Yves Le Boudec. Real-Time Control of an Ensemble of Heterogeneous Resources. In *Proceedings of the 56th IEEE Conference on Decision and Control*. IEEE, 2017.

[6] Andrey Bernstein, Lorenzo Reyes-Chamorro, Jean-Yves Le Boudec, and Mario Paolone. A Composable Method for Real-Time Control of Active Distribution Networks with Explicit Power Setpoints. Part I: Framework. *Electric Power Systems Research*, 125:254–264, 2015.

[7] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[8] Leslie Lamport. Time Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[9] Prince Mahajan, Lorenzo Alvisi, and Mike Dahlin. Consistency, Availability, and Convergence. *University of Texas at Austin Tech Report*, 11, 2011.

[10] Colin J Fidge. Timestamps in Message-Passing Systems that Preserve the Partial Ordering. 1987.

[11] Hermann Kopetz and Günther Bauer. The Time-Triggered Architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.

[12] Aurojit Panda, Wenting Zheng, Xiaohe Hu, Arvind Krishnamurthy, and Scott Shenker. SCL: Simplifying Distributed SDN Control Planes. In *NSDI*, pages 329–345, 2017.

[13] Wajeb Saab, Maaz Mohiuddin, Simon Bliudze, and Jean-Yves Le Boudec. Ordering Events Based on Intentionality in Cyber-Physical Systems. Technical report, EPFL, 2017.

[14] IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems . *IEEE Std 1588-2002*, pages i–144, 2002.

[15] David L Mills. Internet Time Synchronization: the Network Time Protocol. *Communications, IEEE Transactions on*, 39(10):1482–1493, 1991.

[16] John CS Lui, Vishal Misra, and Dan Rubenstein. On the Robustness of Soft State Protocols. In *Network Protocols, 2004. ICNP 2004. Proceedings of the 12th IEEE International Conference on*, pages 50–60. IEEE, 2004.

[17] Sachin Kamboj, Willett Kempton, and Keith S Decker. Deploying Power Grid-Integrated Electric Vehicles as a Multi-Agent System. In *The 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 1*, pages 13–20. International Foundation for Autonomous Agents and Multiagent Systems, 2011.

[18] Jagdish Achara, Maaz Mohiuddin, Wajeb Saab, Roman Rudnik, and Jean-Yves Le Boudec. T-RECS: A Software Testbed for Multi-Agent Real-Time Control of Electric Grids. In *22nd IEEE International Conference on Emerging Technologies And Factory Automation*. IEEE, 2017.

Appendix A
Proof of Theorem V.1

In a CPS that implements Algorithms 1, 2, 3: for any two events a and b,

$$C(a) < C(b) \iff a \to b$$

$$C(a) = C(b) \text{ and } a.pa = b.pa \iff a \equiv b$$

Proof. The proof follows from Lemmas A.1, A.2, A.3, A.4. □

Lemma A.1. $a \equiv b \implies C(a) = C(b)$ *and* $a.pa = b.pa$.

Proof. If $a \equiv b$, then from the properties of the intentional equivalence relation, $a.pa = b.pa$, and one of the following five cases must hold:

1) a and b are initial sending events.
2) $\exists\ c$: $c \xrightarrow{\text{n}} a$ and $c \xrightarrow{\text{n}} b$
3) $\exists\ c, \tilde{c}$: $c \equiv \tilde{c}, c \xrightarrow{\text{n}} a, \tilde{c} \xrightarrow{\text{n}} b$
4) $\exists\ c$: $c \xrightarrow{\text{c}} a$ and $c \xrightarrow{\text{c}} b$
5) $\exists\ c, \tilde{c}$: $c \equiv \tilde{c}$, $c \xrightarrow{\text{c}} a$, $\tilde{c} \xrightarrow{\text{c}} b$

We prove the statement of the lemma by induction.

Base case: c, $\tilde{c}$ are initial sending events.
From item 1: $c \equiv \tilde{c}$.
From Algorithm 1, line 2: $C(c) = C(\tilde{c}) = 0$.
Then, $c \equiv \tilde{c} \implies C(c) = C(\tilde{c})$.

Inductive hypothesis: $C(c) = C(\tilde{c})$. (For cases 2 and 4, $c = \tilde{c}$ and this hypothesis holds trivially).

Inductive step: We show that the statement of the lemma also holds for a and b.
In cases 2 and 3, by Lemma A.6:
$C(a) = C(c) + 1 = C(\tilde{c}) + 1 = C(b)$.
In cases 4 and 5, by Lemma A.5:
$C(a) = C(c) + 1 = C(\tilde{c}) + 1 = C(b)$. □

Lemma A.2. $C(a) = C(b)$ *and* $a.pa = b.pa \implies a \equiv b$.

Proof. We prove this by induction on $l = C(a) = C(b)$.

Base case: For $l = 0$, a and b are initial sending events.
By rule 1 of intentionality: $a.pa = b.pa \implies a \equiv b$.

Inductive hypothesis: $\forall\ e,\ f : C(e) = C(f) = k - 1$ and $e.pa = f.pa \implies e \equiv f$.

Inductive step: We show that, $\forall\ a,\ b : C(a) = C(b) = k$ and $a.pa = b.pa \implies a \equiv b$.

Case 1: a and b are input events.
By Definition 1: $\exists\ e, f$: $e \xrightarrow{\text{n}} a$ and $f \xrightarrow{\text{n}} b$.
Then, from Definition 1: $e.pa = a.pa$ and $f.pa = b.pa$

Thus, $e.pa = f.pa$.
By Lemma A.6: $C(e) = C(a) - 1 = k - 1$.
By Lemma A.6: $C(f) = C(b) - 1 = k - 1$.
Thus, by the inductive hypothesis: $e \equiv f$.
Thus, by rule 3 of intentional equivalence: $a \equiv b$.

Case 2: a and b are output events.
By Definition 2: $\exists\ g, h$: $g \xrightarrow{\text{c}} a$, $h \xrightarrow{\text{c}} b$, and $g.pa = h.pa$.
Then, from Lemma A.5: $C(g) = C(a) - 1 = k - 1$.
Also, from Lemma A.5: $C(h) = C(b) - 1 = k - 1$.
Thus, by inductive hypothesis: $g \equiv h$.
Thus, by rule 5 of intentional equivalence: $a \equiv b$.

Case 3: a is an input event and b is an output event.
We prove that this is an impossible case.
By Definition 1: there exists an input event e, such that $e.pa = a.pa$ and $e \xrightarrow{\text{n}} a$.
By Definition 2: there exists an output event f, such that $f.pa = b.pa$ and $f \xrightarrow{\text{c}} b$.
Then, by Lemma A.6: $C(e) = C(a) - 1 = k - 1$.
Also, from Lemma A.5: $C(f) = C(b) - 1 = k - 1$.
Then, $C(e) = C(f)$ and $e.pa = f.pa$.
From the induction hypothesis: $e \equiv f$.
By the properties of the intentional equivalence relation, e and f are either both input events or both output events.
Contradiction. □

Lemma A.3. $a \to b \implies C(a) < C(b)$.

Proof. By the definition of the intentionality relation (Section IV-D), $a \to b$ has 5 possible cases.

Case 1: $a \xrightarrow{\text{n}} b$.
Then, by Lemma A.6: $C(b) = C(a) + 1$.
Thus, $C(a) < C(b)$.
Case 2: $a \equiv \tilde{a}$ and $\tilde{a} \xrightarrow{\text{n}} b$.
From Lemma A.1: $a \equiv \tilde{a} \implies C(a) = C(\tilde{a})$.
By Lemma A.6: $C(b) = C(\tilde{a}) + 1$.
Thus, $C(a) < C(b)$.
Case 3: $a \xrightarrow{\text{c}} b$
From Lemma A.5: $C(b) = C(a) + 1$.
Thus, $C(a) < C(b)$.
Case 4: $a \equiv \tilde{a}$ and $\tilde{a} \xrightarrow{\text{c}} b$.
From Lemma A.1, $C(a) = C(\tilde{a})$.
From Lemma A.5: $C(b) = C(\tilde{a}) + 1$.
Thus, $C(a) < C(b)$.
Case 5: $a \to c$ and $c \to b$.
From cases 1-4: $C(a) < C(c)$ and $C(c) < C(b)$.
Therefore, $C(a) < C(b)$. □

Lemma A.4. $C(a) < C(b) \implies a \to b$.

Proof. $C(a) < C(b) \implies C(b) = C(a) + k,\ k > 0$.
We prove the lemma by induction on k.

Base case: For $k = 1$, $C(b) = C(a) + 1$

Case 1: b is an input event.
By Definition 1: $\exists\ c : c \xrightarrow{\text{n}} b$.
By Lemma A.6: $C(b) = C(c) + 1$.
Thus, $C(c) = C(a)$.
From Lemma A.2: $c \equiv a$.
Hence, from rule 2 of intentionality: $a \to b$.

Case 2: b is an output event.
By Definition 2: $\exists\ c : c \xrightarrow{\text{c}} b$.
From Lemma A.5: $C(b) = C(c) + 1$.
Thus, $C(c) = C(a)$.
From Lemma A.2: $c \equiv a$.
Hence, from rule 4 of intentionality: $a \to b$.

Inductive hypothesis: Let, for some $k > 1$
$\forall\ e, f : C(f) = C(e) + k \implies e \to f$.

Inductive step: We show that
$\forall\ a, b : C(b) = C(a) + k + 1,\ a \to b$.
$\exists\ c : C(b) = C(c) + 1, C(c) = C(a) + k$.
From the inductive hypothesis: $a \to c$.
Also, from the base case: $c \to b$.
Therefore, from rule 5 of intentionality: $a \to b$. □

Lemma A.5. $a \xrightarrow{c} b \implies C(b) = C(a) + 1$

Proof. In Algorithm 2, a is either from a reception event from a PA with label C' or a timeout event with a label C' (line 6). In line 8, we have $C = C' + 1$ and the sending event b has a label C. Thus, $C(b) = C(a) + 1$.
In Algorithm 3, the sending event b has a label C that is updated in lines 10 and 12. In line 10, we have $C = C(a)$ and in line 12, we have $C = C + 1$. Thus, $C(b) = C(a) + 1$. □

Lemma A.6. $a \xrightarrow{n} b \implies C(b) = C(a) + 1$

Proof. **At the PA**
A sending event a at a controller replica occurs in line 17 of Algorithm 1.
The corresponding reception event b at a PA occurs at line 8 of Algorithm 3.
From Algorithm 3, line 8: $C(b) = C(a) + 1$.

At the Controller
Algorithm 1 line 9: a reception event b has a label $b.l = l + 1$, where l is the label of the corresponding sending event a at a PA (Algorithm 3, line 16).
Thus, $C(b) = C(a) + 1$.

Algorithm 2 line 4: we declare timeout events with label C', for each PA i, such that the maximum label in $\mathbf{L}$ of the events corresponding to i is different from C'.
We trace the events that caused one such timeout event b.
Since $a \xrightarrow{\text{n}} b$, it follows that a is the sending event at a PA, such that $a.pa = b.pa$, which was lost or delayed thus causing the timeout event b.
Let g be the last sending event at a controller, such that there exists a chain of events $g \xrightarrow{\text{n}} f \xrightarrow{\text{c}} d \xrightarrow{\text{n}} c$, where c is a reception event at the controller on which b occurred, and $c.l = C'$. (Definitions 1 and 2).
Thus, from result of this lemma at the PA: $d.l = C' - 1$.
From Lemma A.5: $f.l = C' - 2$.
From the earlier statement at the controller: $g.l = C' - 3$.
The controller declares timeout events to acknowledge that it lacks information from some PAs, that it has from others. This information is the state of the sub-processes after the

implementation of the setpoints encapsulated in the last sending events that resulted from the same computation relation at some controller replica.
Thus, from Definitions 1 and 2: there exists another chain of events, $g \xrightarrow{\text{n}} h \xrightarrow{\text{c}} a$, such that $h.pa = a.pa$.
From result of this lemma at the PA: $h.l = C' - 2$.
From Lemma A.5: $a.l = C' - 1$. □

APPENDIX B
PROOF OF THEOREM V.2

A CPS that implements Algorithms 1, 2, 3 guarantees safety and optimal selection.

Proof. The proof has two parts: safety and optimal selection.

Safety

At the controller: Algorithm 2 line 10, the controller computes setpoints using measurements with label C'.
C' is greater than the label of the last setpoint issued at line 17 in Algorithm 1 with label C, as $C' = \max(C+3, \max(\mathbf{L}))$ (Algorithm 2, line 1).
From Theorem V.1: $C(b) < C(a) \implies b \to a$.
Thus, when an event a with label C' is used by the controller for computation, and another event b was the last event issued by the controller with label C, then $C' > C \implies b \to a$.

At the PA: The computation at PA, i.e., implementation of a setpoint followed by subsequent computation of a measurement, is triggered only if Algorithm 3 line 9 is true.
Therefore, if an event a with label $a.l$ is used in a computation by a PA when its CPS causal clock is C, then $C < a.l$.
However, C is the label of the sending event b, corresponding to the last measurement issued.
Therefore, by Theorem V.1: $C(b) < C(a) \implies b \to a$.

Optimal Selection

At the controller: Consider a reception event a with label $a.l$ at a controller, when its clock is C.

Case 1: $a.l \leq C$.
Let b is a sending event corresponding to the last setpoint sent to the same PA.
From Theorem V.1: $C(a) \leq C \implies b \not\to a$.
From Algorithm 2 line 1, we have $a.l < C + 3 < C'$.
In line 10, the controller computes with events of label C'.
Thus, the event a is not used in computation, i.e., discarded.

Case 2: $a.l > C$ and $a.l \neq \max(\mathbf{L})$.
Consider an event e such that $C(e) = \max(\mathbf{L})$.
From Theorem V.1: $C(a) < C(e) \implies a \to e$.
From Lemma B.1: $C(a) < C(e) \implies C(e) > C(a) + 4$.
Let d be the sending event on a PA, such that $d \xrightarrow{\text{n}} e$.
Then, $C(d) = C(e) - 1$.
Let c be the reception event on the PA, such that $c \xrightarrow{\text{c}} d$.
Then $C(c) = C(e) - 2$.
Let b be the sending event on a controller, such that $b \xrightarrow{\text{n}} c$.
Then $C(b) = C(e) - 3$.
Thus, $C(b) > C(a)$.
FBy Theorem V.1: $C(b) > C(a) \implies b \not\to a$.
Hence, a must be discarded.
From Algorithm 2 line 1, we have $a.l < \max(\mathbf{L}) < C'$.
In line 10, the controller computes with events of label C'.
Thus, the event a is not used in computation, i.e., discarded.

Case 3: $a.l > C$ and $a.l = \max(\mathbf{L})$
From Lemma B.1: $a.l > C + 3$.
From Algorithm 2 line 1: $a.l = C'$.
In line 10, the controller computes with events of label C'.
Thus, the event a is used in computation, i.e., not discarded.

At the PA: An event a is discarded by the PA only if $a.l \leq C$ (Algorithm 3, line 9), where C is the label of the event b corresponding to the last measurement sent.
Thus, from Theorem V.1, if a is discarded, $b \not\to a$. □

Lemma B.1. *For any two events a and b occurring at the same software agent, if a and b are both input events or both output events, then $[C(b) - C(a)] \ \% \ 4 = 0$.*

Proof. We start by proving the statement for output events at the controller, by using induction.

Base case: Let a be an initial sending event. Then, $C(a) = 0$.
Let b be an event occurring on a PA, such that $a \xrightarrow{\text{n}} b$.
Let c be a sending event on the same PA, such that $b \xrightarrow{\text{c}} c$.
Let d be an event on the controller, such that $c \xrightarrow{\text{n}} d$.
Let e be the sending event on the controller, such that $d \xrightarrow{\text{c}} e$.
Then, $C(e) = 4$ and $[C(e) - C(a)] \ \% \ 4 = 0$, where e and a are both output events. Thus, the statement is true for the first two sending events on all controllers.

Inductive hypothesis: Consider an instance of the execution trace at time t_0 at which there is no event in the CPS b, such that $C(b) \geq l$. At this instant, let $\mathcal{A}$ be the set of all sending events on all controller. Clearly, $\forall a \in \mathcal{A}, C(a) < l$.
We assume the hypothesis be true for all $a \in \mathcal{A}$.

Inductive step:
Consider $a_0 \in \mathcal{A}$, be such that $\forall\ a \in \mathcal{A}, C(a_0) \geq C(a)$.
By the induction hypothesis: $C(a_0) = 4k$, where $k \in \mathbb{N}$.
We show that the hypothesis is also true for the first sending event b that occurs at a controller after t_0 such that $a_0 \to b$.
$\forall\ c$, $(c.sa = \text{PA}_1 \wedge a_0 \xrightarrow{\text{n}} c) \implies C(c) = 4k + 1$.
$\forall\ d$, $(d.sa = \text{PA}_1 \wedge c \xrightarrow{\text{c}} d) \implies C(d) = 4k + 2$.
$\forall\ e$, $(e.sa \in \mathcal{C} \wedge\ e.pa = \text{PA}_1 \wedge d \xrightarrow{\text{n}} e) \implies C(e) = 4k + 3$.
$\forall\ f$, $(f.sa = e.sa \wedge\ e \xrightarrow{\text{c}} f) \implies C(f) = 4k + 4$.
Thus, $[C(f) - C(a_0)] \ \% \ 4 = 0$.
f is a sending event on a controller and $C(f) > C(a_0)$.
Thus, $f \notin \mathcal{A}$.
Also, f is the first sending event that occurred after t_0.
Thus, the hypothesis holds for all sending events at controllers.

Sending events at controllers have label of the form $4k$.
By Lemma A.6: input events at PAs have label of the form $4k + 1$.
From Algorithm 3: output events at the PAs have label of the form $4k + 2$.
By Lemma A.6: input events at the controller have label of the form $4k + 3$. □