Formal Verification of a Mixed-Trust Synchronization Protocol

Ruben Martins
rubenm@andrew.cmu.edu
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

Michael McCall
mjmccall@cert.org
Carnegie Mellon University
Software Engineering Institute
Pittsburgh, Pennsylvania, USA

Dionisio de Niz
dionisio@sei.cmu.edu
Carnegie Mellon University
Software Engineering Institute
Pittsburgh, Pennsylvania, USA

Amit Vasudevan
avasudevan@sei.cmu.edu
Carnegie Mellon University
Software Engineering Institute
Pittsburgh, Pennsylvania, USA

Bjorn Andersson
baandersson@sei.cmu.edu
Carnegie Mellon University
Software Engineering Institute
Pittsburgh, Pennsylvania, USA

Mark Klein
mk@sei.cmu.edu
Carnegie Mellon University
Software Engineering Institute
Pittsburgh, Pennsylvania, USA

John P. Lehoczky
jl16@andrew.cmu.edu
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

Hyoseung Kim
hyoseung@ece.ucr.edu
UC Riverside
Riverside, California, USA

ABSTRACT

Cyber-Physical Systems (CPS) are becoming widespread in many safety-critical real-time applications, such as autonomous driving, medical monitoring, robotics systems, and unmanned aircraft. However, verifying these complex real-time systems remains an open challenge because traditional verification techniques are unable to verify all components.

One approach to address this challenge is to use the framework for mixed-trust computing for real-time systems where unverified (untrusted) components are constrained not to exhibit unsafe behavior by verified (trusted) components. This framework increases assurance in the CPS by verifying timing and functional properties of the trusted components. However, even though the trusted components are verified, formal verification of the synchronization protocol between trusted and untrusted components has been an open problem. If the synchronization protocol between the untrusted and trusted components is incorrect then the behavior of the entire system can be compromised.

In this paper, we present a formal model of a synchronization protocol between trusted and untrusted components using timed automata. We use temporal logic to prove the protocol satisfies properties that guarantee its correctness. The verification was used to identify and correct a critical flaw in the previous protocol implementation and increases confidence in the mixed-trust framework.

KEYWORDS

Mixed-Trust, Formal Methods, Timed Automata

RTNS'2021, April 7–9, 2021, NANTES, France

ACM ISBN 978-1-4503-9001-9/21/04…$15.00
https://doi.org/10.1145/3453417.3453431

ACM Reference Format:
Ruben Martins, Michael McCall, Dionisio de Niz, Amit Vasudevan, Bjorn Andersson, Mark Klein, John P. Lehoczky, and Hyoseung Kim. 2021. Formal Verification of a Mixed-Trust Synchronization Protocol. In *29th International Conference on Real-Time Networks and Systems (RTNS'2021), April 7–9, 2021, NANTES, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3453417.3453431

1 INTRODUCTION

Cyber-Physical Systems (CPS) are becoming more common and are often used for safety-critical real-time applications. For CPS to be safely used, they have to go through a rigorous independent verification and validation (IV&V) procedure and be certified by the authority relevant to the sector in which the CPS will be deployed (e.g., FAA [34]). One approach to certify CPS is to use formal methods to mathematically verify that the CPS components respect a set of safety properties (e.g., an unmanned aircraft will never crash). Unfortunately, the complexity of CPS surpasses the current capabilities of verification tools and techniques precluding formal verification of the entire system.

Current IV&V approaches assign different levels of criticality to each software and hardware component and provide a variety of testing and verification techniques that can be applied to each component. However, if an organization elects to formally verify critical components, at the end of the IV&V procedure, the system will consist of a set of components that are formally verified—which we denote by *trusted components*—and a set of components that may not be verified due to their complexity or low criticality—which we denote by *untrusted components*. Even though the trusted components are formally verified, this verification is often done assuming that the trusted components are isolated and cannot be compromised by the untrusted components. For instance, small verified hypervisors and microkernels such as seL4 [27], CertiKOS [22], and uberXMHF [38, 39], have only a few trusted components that are run in a protected memory space and are isolated from the untrusted components that are run within a virtual machine on

top of an operating system like Linux. Unfortunately, verification based on isolation limits the trusted components to small pieces of software that can be verified. This is not realistic for many CPS systems where the software responsible for complex real-time operations like autopilot or image recognition is beyond the reach of current verification approaches.

To address this issue, the framework for mixed-trust computing for real-time systems [16] has been recently proposed to support untrusted components that can be used in critical functions of CPS. In this framework, untrusted components are monitored by trusted components that ensure functional and timing guarantees, i.e. that the composition of an untrusted component and its associated trusted component will always output an action that is both safe and on time. When the trusted component detects any failure from the untrusted component, it overrides its output with fail-safe action. Previous work has shown how trusted components can be verified with the isolation provided by a verified hypervisor [39]. It has also been shown how the functional behavior of untrusted components can be guaranteed by using logical enforcers [3, 17] and how timing properties are guaranteed by using timing enforcers [13]. The mixed-trust framework combines these techniques by introducing the concept of synchronization between trusted and untrusted components and analyzes the correct timing behavior of these components sharing a processor. However, formally proving that the synchronization protocol between these components is correct is still an open problem.

The synchronization protocol is a cornerstone of the mixed-trust framework given that without it we cannot guarantee that the trusted components will take control of the system in time and act accordingly. A faulty synchronization protocol could compromise the entire system.

In this paper, we address this open problem by presenting a formal model of the synchronization protocol using timed automata. We formally prove temporal properties on our model that guarantee the correct behavior of the synchronization protocol between trusted and untrusted components. The verification of the protocol led us to find and correct a critical flaw in a previous implementation where trusted components were not correctly detecting the start and finish time of untrusted component execution.

In summary, this paper makes the following main contributions:

- A formal model using timed automata of the synchronization protocol between trusted and untrusted components.
- Formal verification using Timed Computational Tree Logic (TCTL) [2, 24] of properties that ensure the correct behavior of the mixed-trust framework.
- Identification and correction of a critical flaw in the previous implementation of the synchronization protocol in the real-time mixed-trust computing framework [16].

2 MIXED-TRUST FRAMEWORK

To understand the details of the mixed-trust synchronization protocol, in this section, we provide the background on the *real-time mixed-trust computing* framework (RT-MTC) [16]. This framework enables the use of untrusted components within critical CPS functionality. That is, in this framework, a critical function like autonomous navigation of a robot is allowed to use an untrusted component

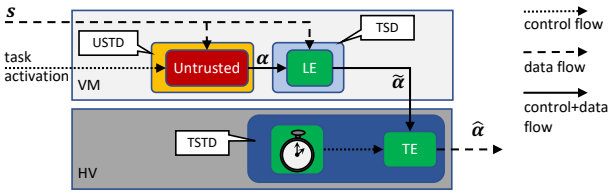


Figure 1: Mixed-Trust Task Runtime Architecture [16]

without compromising safety guarantees. This contrasts with other approaches that do not use untrusted components in critical CPS functions.

The RT-MTC framework allows the use of untrusted components in critical functions since these components are monitored by verified components that ensure that the output of the untrusted components always leads to safe states (e.g., avoiding crashes). These monitoring components are known as *logical enforcers* [3, 17] and are protected by a verified micro-hypervisor [39].

The RT-MTC framework uses temporal enforcers to preserve timing guarantees of the system. Temporal enforcers are small, self-contained code blocks that perform a default safety action (e.g., *hover* in a quadrotor) if the untrusted component has not produced a correct output by a specified time.

Temporal enforcers are contained within the verified micro-hypervisor without jeopardizing the existing level of trust (e.g., using compositional verification offered by extensible micro-hypervisors [39]). The untrusted part and the temporal enforcers (the trusted part) are scheduled as a combined task that is called *mixed-trust task*. The untrusted part is executed in a virtual machine (VM) running on top of a trusted hypervisor, which in turn executes the trusted temporal enforcer.

Figure 1 depicts the architecture of the mixed-trust task. The figure shows how the output of the untrusted component is validated by the logical enforcer (LE) to make sure the value is safe.[1] However, because the untrusted component may delay the production of the output, a timer reserved exclusively for the hypervisor (HV) is used to trigger the temporal enforcer (TE) if no output is produced by the LE. However, if the LE produces an output before the timer, then this value is used as the output (e.g., actuation in a controller).

The architecture also presents three protection domains: *Untrusted Space and Temporal Domain* that has no memory or time protection where the untrusted component lives, *Trusted Space Domain* that ensures the memory of the component is not compromised—where the LE lives—and *Trusted Space and Temporal Domain* that provides not only memory protection but also timing protection, implemented by the HV timer—this is where the TE lives.

In the RT-MTC framework, a system is composed of a set of periodic tasks where each task has an untrusted component guarded by the logical and the temporal enforcer as presented in Figure 1. The scheduling scheme in [16] models these tasks as running on a single-core processor where the set of tasks is modeled as $\Gamma = \{\mu_i | \mu_i = (T_i, D_i, \tau_i, \kappa_i)\}$.

In this task set, μ_i is a mixed-trust task with two execution segments, τ_i and κ_i, with period T_i and deadline D_i. The segment

[1] We refer the reader to the literature [3] for a formalization of this scheme.

Algorithm 1: Periodic Task Programming

```
1 while true do
2 |   jobComputation();
3 |   waitForNextPeriod();
4 end
```

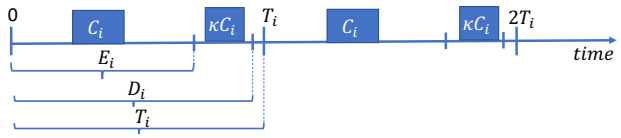


Figure 2: Mixed-Trust Task Execution Timeline Sample

τ_i is the untrusted component and runs in the untrusted OS kernel inside the VM. The segment κ_i is the trusted component and runs within the trusted HV. To represent the fact that these segments are handled by different schedulers, prior work [16] considers them to be tasks and call τ_i the *guest task* (*GT*) and κ_i the *hyper task* (*HT*). These tasks are defined by: $\tau_i = (T_i, E_i, C_i)$, $\kappa_i = (T_i, D_i, \kappa C_i)$, where T_i and D_i are the same as in μ_i, C_i is the worst-case execution time (WCET) of τ_i, κC_i is the WCET of κ_i[2], and E_i is the deadline for τ_i. While the combination of trusted and untrusted parts have some similarities to critical and non-critical components in previous mixed-criticality frameworks [10, 18], it is worth highlighting that the additional logical properties and memory protection imposes new challenges in the mixed-trust framework that are not present in mixed-criticality scheduling.

Algorithm 1 shows a sample pseudocode of a periodic task. In this case, each execution of the function `jobComputation()` is considered a job after which the task waits for the period to elapse before executing the next iteration. As can be seen in the algorithm, there can be an infinite number of executions of such a function and, hence, we say that a task is considered to be a potentially infinite sequence of jobs. The q^{th} mixed-trust job of a mixed-trust task μ_i is denoted as $\mu_{i,q}$. This mixed-trust job is composed of a guest job $\tau_{i,q}$ and a hyper job $\kappa_{i,q}$. Ideally, $\tau_{i,q}$ will execute correctly taking no more than C_i time units and finishing within E_i time units after its arrival. In this case, the job $\kappa_{i,q}$ is not activated.

The logical enforcer (*LE*) verifies the correctness of the output of $\tau_{i,q}$, while the timing enforcer (*TE*) verifies that the output was produced on time. If the logical enforcer (*LE*) does not notify the HV that $\tau_{i,q}$ finished correctly and on time, then the corresponding HT $\kappa_{i,q}$ is activated by a timer set to expire E_i time units after $\tau_{i,q}$ arrives, and runs at a higher priority than any GT.

Figure 2 shows a timeline of the execution of two jobs of a single mixed-trust task with markings for all its parameters. In the following, we will consider that the deadline is equal to the period for simplicity.

The E_i parameter of each task can be calculated by the algorithm in [16] that performs schedulability testing. If the task set is schedulable, then it provides the following guarantees:

[2]As an extension to the original RT-MTC framework, one can leverage the execution budget of hyper tasks to run alternative computations when there is no need to perform enforcement computations. Interested readers can refer to [15].

(1) **Guest task τ_i finish by E_i**. If the job of a τ_i does not execute more than C_i then it is guaranteed to finished by E_i.
(2) **Hyper task κ_i finish by D_i**. Even if a job from the corresponding guest task τ_i does not finish, if the hyper task is started by E_i then it will finish by D_i. In this case, we assume that the job of κ_i will not execute for more than κC_i.

The guarantees provided above are given for each task and hence when reasoning about the timing of an individual task, such a task, and its guarantees can be considered in isolation as if no other task exists. Note that an analysis of the concurrent execution of the system was conducted in prior work [16]. In this paper, we focus on the correctness of the interactions from both the logical and timing perspectives of one mixed-trust task's guest and hyper segments. This correctness can then be used for all tasks.

To prevent an untrusted component from causing unexpected behavior, the RT-MTC framework [16] defines that a system has *correct behavior* if the following conditions are satisfied:

- **C1.** Each mixed-trust task must produce an output every period.
- **C2.** There is only one output per period.
- **C3.** The output produced by a task in a period is either from *LE* or *TE*.
- **C4.** A guest output must be the result of a job computation that executes within a single period.
- **C5.** The *TE* of a task must execute E time units after the arrival of the job it guards and finish before the deadline.

Note that only condition **C5** was formally proven in prior work by de Niz et al. [16]. Condition **C5** is a scheduling problem that can be solved with real-time scheduling theories. This paper addresses the gap in prior work where conditions **C1** to **C4** were not formally proven. Additionally, we also prove a new condition **C6** that together with **C4** guarantees that, if there is a valid run of a guest task, then that output is used as the output of the mixed-trust task:

- **C6.** If the guest task finishes before the guest deadline, then the output of this task is used as the output of the mixed-trust task.

Before formally proving these properties, we introduce the timing semantics used in this paper and show how these relate to the properties that must be proven to establish the correctness of the synchronization protocol (Section 3). Afterward, we model the interaction between the guest task and the hyper task with a timed automata model (Section 4.2). Next, we encode these properties using Timed Computational Tree Logic (TCTL) and prove their satisfaction by our model (Section 4.3). Finally, we discuss how the verification process leads to the discovery of a critical flaw in a previous implementation of the synchronization protocol and discuss how our correction conforms to the model (Section 5).

3 TIMING SEMANTICS

The mixed-trust synchronization protocol is responsible for the correct communication between trusted and untrusted components and for guaranteeing the correct behavior of the system with respect to *time* and *functional behavior*.

In this section, we present the timing semantics used by the guest task and hyper task for a valid execution of the system. An

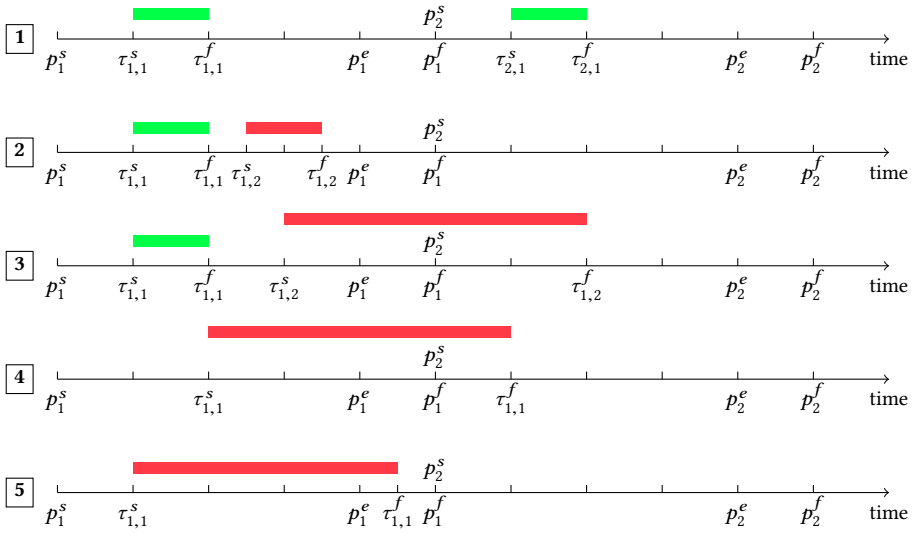


Figure 3: Example of valid (light green) and invalid guest jobs (dark red)

execution of the system consists of a sequence of instructions, each belonging to either the hyper or guest task. We decompose the execution of the system into a collection of *guest and hyper jobs* that occur in each *period*. In addition, we focus on the synchronization protocol between the guest and the hyper task and on the properties that must be satisfied to ensure the correct behavior of the RT-MTC framework. In the remainder of this paper, we assume that the resolution of the clock model is high enough to guarantee that any two consecutive timestamps will always have different values. [3]

Definition 3.1 (Guest Job and Hyper Job). A *guest job* $\tau_{i,q}$ is a job run by the guest task and defined as a tuple $\langle \tau^s_{i,q}, \tau^f_{i,q}, \tau^\alpha_{i,q} \rangle$. Similarly, a *hyper job* $\kappa_{i,q}$ is a job run by the hyper task and defined as a tuple $\langle \kappa^s_{i,q}, \kappa^f_{i,q}, \kappa^\alpha_{i,q} \rangle$. For simplicity, in the remainder of this paper we will omit the indexes i, and q from jobs $\tau_{i,q}$ and $\kappa_{i,q}$, unless we need to refer to multiple jobs in a given task.

A guest job and a hyper job have the following properties:

- $\tau^s, \tau^f, \kappa^s, \kappa^f \in \mathbb{N} \cup \{\infty\}$ correspond to the initial (τ^s, κ^s) and final timestamps (τ^f, κ^f) of a job with $\tau^s < \tau^f, \kappa^s < \kappa^f$. Note that if a guest job is still active then $\tau^f = \infty$, and if a hyper job is still active then $\kappa^f = \infty$.
- $\tau^\alpha, \kappa^\alpha$ denotes the output action of a guest job and a hyper job, respectively. For instance, an output action of a guest job for a quadrotor could be "move left", while the hyper job output action would be a default safety action such as "hover". If there is no output for a guest job then $\tau^\alpha = \perp$, and if there is no output for a hyper job then $\kappa^\alpha = \perp$, where $\perp$ denotes the empty action. For a formal definition of an output action in the context of enforcers, we refer the reader to the literature [3].

If we could trust the guest task, then we could assume that there would only be a single guest job per period, which would finish before the deadline. However, since the guest task is untrusted, its behavior cannot be predicted and we cannot make any assumptions on it. Therefore, we need to define when the guest task performs a valid guest job, i.e., when it satisfies timing and logical requirements. If a guest's job is valid then the output of the guest task can be used by the mixed-trust system.

Definition 3.2 (Logical Period). A logical period p is a tuple $\langle p^s, p^f, p^e, p^\alpha, p^{\vec{j}} \rangle$ where:

- $p^s, p^f \in \mathbb{N}$ correspond to the initial and final timestamps of period p, respectively, with $p^s < p^f$.
- $p^e \in \mathbb{N}$ is the deadline for all jobs performed by the guest task in a single period, with $p^s < p^e < p^f$.
- p^α denotes the output action of the mixed-trust framework for period p.
- $p^{\vec{j}}$ is a collection of jobs that occur in period p.

For simplicity, we will refer to logical period as period for the remainder of the paper.

Definition 3.3 (Valid Guest Job). Given a period p, we consider that a guest job $\tau \in p^{\vec{j}}$ is valid if and only if:

- $\tau^s > p^s$ and $\tau^f < p^e$, i.e., if the job started after the beginning of the current period and finished before the deadline for the guest task.
- $\forall_{\tau_{i,q} \in p^{\vec{j}}} \tau^s \leq \tau^s_{i,q}$, i.e., if more than one guest job was run in the same period p, we only consider as valid the job that started first.

Definition 3.4 (Valid Execution). For each period p, the execution of the RT-MTC framework is valid if and only if the following conditions are met:

(1) The hyper task only runs *one hyper job* $\kappa \in p^{\vec{j}}$ with $\kappa^s \geq p^e$ and $\kappa^f < p^f$, i.e., it will always terminate before the deadline and will always have an output action $\kappa^\alpha \neq \perp$.

[3] An example where this assumption holds is described in the literature for an Intel architecture [13].

(2) Let $valid : jobs \rightarrow \{0, 1\}$ be a predicate that takes as input a guest job $\tau \in p^{\vec{j}}$ and returns 1 if τ is valid and 0 otherwise. The output of the each period is determined as follows:

$$p^{\alpha} = \begin{cases} \kappa^{\alpha}, & \text{if } \forall_{\tau \in p^{\vec{j}}} \neg valid(\tau) \\ \tau^{\alpha}, & \text{otherwise} \end{cases}$$

If there are no valid guest jobs, then the output action of the period is the output action of the hyper job κ^{α}. Otherwise, it is the output action of a valid guest job τ^{α}.

To verify that a synchronization protocol is correct, we need to prove that *all periods have a valid execution*. Note that a valid execution satisfies the conditions **C1** to **C4**, and **C6** described in Section 2. In other words, for an execution to be valid, we need to verify that there is exactly one output per period (**C1**, **C2**) and this output is either from the guest task or the hyper task (**C3**). Moreover, if the guest has a valid job (**C4**) then we need to verify that the output of the period is the one from the guest task (**C6**).

Example 3.5 (Valid and invalid guest jobs). Figure 3 shows different executions scenarios of the guest task. The segments highlighted in light green (dark red) correspond to valid (invalid) guest job tasks.

In Figure 3-1, we can observe valid executions of the guest task since the job $\tau_{i,q}$ always starts and ends before p_i^e, i.e. $\tau_{i,q}^s < \tau_{i,q}^f < p_i^e$. In Figure 3-2, even though both guest jobs $\tau_{1,1}$ and $\tau_{1,2}$ start and finish before p_1^e, we only consider one valid guest job per period. Therefore, $\tau_{1,1}$ is a valid guest job because it was run first, while $\tau_{1,2}$ is an invalid guest job because another valid guest job already exists in that period. Figure 3-3, shows a valid guest job with beginning and end before the enforcement deadline p_1^e. However, it also shows another job $\tau_{1,2}$ that starts in period p_1 but only ends in period p_2. This computation was not done in a single period and did not finish before p_1^e, it is considered invalid. Figure 3-4 and Figure 3-5, depict invalid guest job executions because in both cases the guest job finishes after p_1^e.

4 MODELING AND VERIFICATION OF A MIXED-TRUST SYNCHRONIZATION PROTOCOL USING TIMED AUTOMATA

We model and verify the mixed-trust synchronization protocol using Uppaal [30]. The Uppaal toolbox allows us to model this protocol with timed automata [2] extended with additional features such as integer variables and channel synchronization. In this section, we start by briefly reviewing the Uppaal toolbox with respect to timed automata modeling and verification of properties using a subset of TCTL [2, 24]. Next, we show how we can model the mixed-trust synchronization protocol using timed automata. Finally, we show how we can prove properties that imply the satisfaction of conditions **C1** to **C4**, and **C6**.

4.1 Primer on Uppaal

The Uppaal modeling language is an extension of timed automata [5]. A timed automaton is a finite-state machine extended with clock variables where all clocks progress synchronously. Further details on time automata can be found in [2].

An Uppaal model is composed of a network of multiple timed automata. Uppaal supports extensions of timed automata that are useful to model the synchronization protocol, namely, it allows for bounded discrete variables for additional state information and broadcast channels that synchronize edge firing for multiple automata. The state of a network of timed automata is defined by the locations of all automata, the clock values, and the values of the discrete variables. Each automaton may fire an edge if the conditions on that edge are satisfied, e.g. if clock conditions are met (written in green in the automata shown in this section) or if it receives a signal that was broadcast by another automaton (written in light blue). Additionally, edges can also encode assignments to discrete variables (written in blue) as post-conditions of firing an edge. The automata in the network are synchronized using channels. A channel is a label on an edge that synchronizes a discrete transition with a corresponding label on an edge in another automaton. In this paper, we restrict ourselves to the usage of broadcast channels (c). In a broadcast synchronization, one sender (c!) can synchronize with any number of receivers (c?). Note that a broadcast action is non-blocking and can execute a c! action with no receivers.

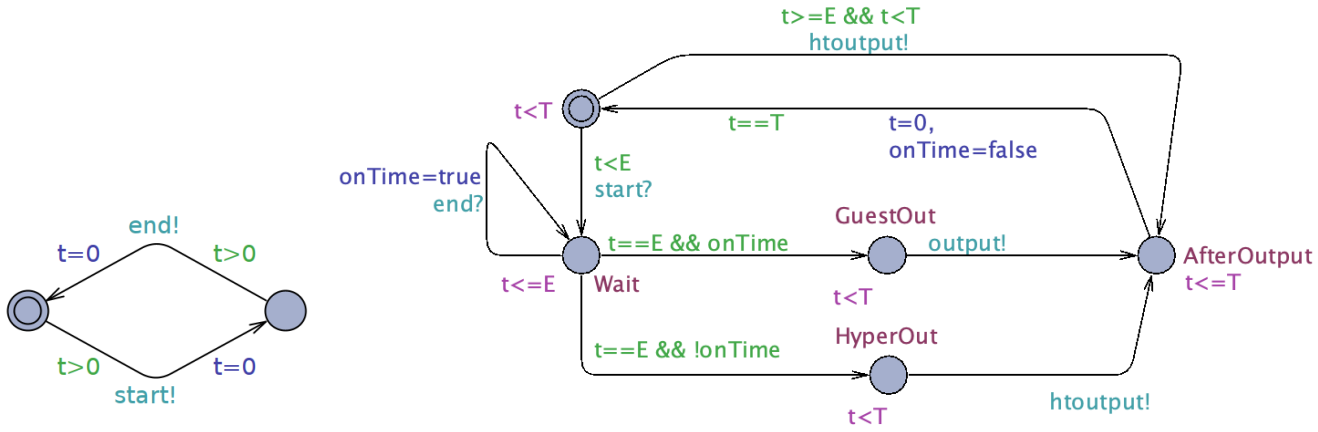
Uppaal uses a simplified version of TCTL [5] to write temporal properties for verification. In this paper, all our temporal formulas prove safety properties, i.e. that nothing bad will happen. Therefore, our temporal formulas will have the form $AG\neg\varphi$, which denotes that the formula $\neg\varphi$ holds in the current state and in all execution paths, i.e. φ will never be true in any state.

4.2 Modeling using Timed Automata

The mixed-trust protocol is represented by two automata in our Uppaal model, one for the guest task and the other for the hyper task. We model this protocol to determine whether every execution is valid per Section 3. Even though this model appears simple, it is in fact the product of many iterations, with many fault discoveries and resolutions. Note that this process of refining a simple model is more efficient than fixing implementation defects resulting from an unclear or incorrect design [35].

4.2.1 Guest Task Automaton. Since we do not trust the guest's task, we model its behavior with an automaton that represents all possible behaviors. Figure 4a shows our modeling of the guest task, where we assume that only one guest job can be run at each time, and each job communicates a start and end. Starting from the initial state, the guest task broadcasts to the other automata the start of the job using the *start* channel. When the guest's job finishes, it broadcasts this information using the *end* channel.

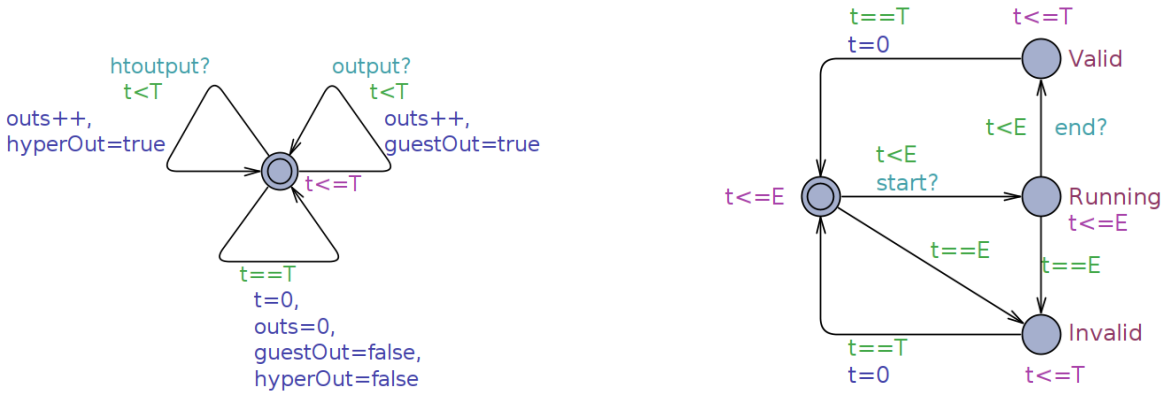
4.2.2 Hyper Task Automaton. The hyper task automaton is described in Figure 4b. An *output* signal is broadcast if the guest's job execution is valid, and an *htoutput* otherwise, representing a guest and hyper task output, respectively. From the initial location (represented by the node with the inner circle), if the hyper task receives a *start* signal before the deadline for the guest task (E), then it transitions to the *Wait* location where the hyper task waits for an *end* signal. If an *end* signal is received before the clock reaches E, the edge to *GuestOut* will be taken, and then an edge to *AfterOut* is taken, broadcasting an *output* signal. In this case, the guest's task had a valid job. If the hyper task remains in either the initial or *Wait*



(a) Guest task automaton

(b) Hyper task automaton

Figure 4: Modeling the mixed-trust synchronization protocol in UPPAAL



(a) Output Observer

(b) Job Observer

Figure 5: Observer automata

location and the clock reaches E, then the guest task did not have a valid job, the *AfterOutput* location is reached and the *htoutput* signal is broadcast. In all cases, we end up at *AfterOuput* to wait for the end of the period to transition back to the initial state and reset the clock.

4.2.3 Modeling Challenges. The automata presented in this section are simple, but this simplicity resulted from an iterative process that addressed several modeling challenges. For instance, the conditions and invariants for each edge must be carefully chosen to avoid multiple events from occurring simultaneously or having infinite discrete events occurring in finite time (known as Zeno executions). For example, in Figure 4a the clock condition for the guest task to start a job ($t > 0$) prevents the issue of simultaneity where we could potentially have a *start* and *end* in the same instant. Additionally, the properties could be proven to be vacuously true due to modeling mistakes such as certain locations being unreachable. To guarantee that this was not the case, we verified auxiliary properties to validate model behavior such as all locations being reachable, all clocks being synchronized, and that the model will never enter a deadlock. A lot of time was spent modifying the model to improve readability and clarity while still preserving its validity and ensuring it satisfies the specification. Modifying the model for clarity or to satisfy a property can easily invalidate it, so it can save time to have auxiliary properties to check the validity of the model automatically rather than by manual simulation.

4.3 Verification of Timing Properties

The synchronization protocol is correct if it satisfies conditions **C1** to **C4**, and **C6** described in Section 2. We show that these conditions are true by proving the following **P1-P4** properties that connect the conditions in Section 2 with the timing semantics presented in Section 3:

- **P1.** Each mixed-trust task must produce an output every period.

- **P2.** There is only one output per period.
- **P3.** If the guest job execution is valid, the output is from the guest task.
- **P4.** If the guest job execution is invalid, the output is from the hyper task.

Since Uppaal's specification language is restricted to a subset of TCTL, certain properties cannot be written directly (e.g., properties that involve alternating temporal operators). Therefore, we constructed two observer automata that observe the behavior of the guest and hyper tasks to make it easier to write the properties **P1**-**P4** as TCTL formulas.

4.3.1 Output Observer Automaton. Behaviors related to **P1** and **P2** are tracked using the observer automaton shown in Figure 5a. This automaton has a clock t that corresponds to the clock in the hyper task. Transitions are used to receive *output* and *htoutput* signals that each increment the integer *outs* to count the number of outputs. At the end of the period, t and *outs* are reset to zero.

This automaton makes it easier to write TCTL formulas to prove properties **P1** (Equation 1) and **P2** (Equations 1 and 2):

$$AG \neg (outputObserver.t == T \wedge outputObserver.outs == 0) \quad (1)$$

$$AG \neg (outputObserver.outs > 1 \wedge outputObserver.t < T) \quad (2)$$

If these formulas are valid then the protocol never finishes a period with zero outputs (Equation 1) or with more than one output (Equation 2).

4.3.2 Job Observer Automaton. Additional behavior related to **P3** and **P4** is tracked by the observer automaton shown in Figure 5b. The signals *start* and *end* from the guest task are tracked to determine if the guest task job execution is valid. If *start* and *end* signals are received (in that order) within one period before E, the job observer reaches the *Valid* location, otherwise it will reach the *Invalid* location. Upon reaching the end of the period, T, the automaton returns to the initial location. Once it reaches the *Valid* location, it must be the case that a guest output is used, otherwise **P3** is violated. Therefore, we can check that **P3** is satisfied by checking the validity of the following formula:

$$AG \neg (jobObserver.Valid \wedge hyperOut) \quad (3)$$

This formula states that it is never the case that a state is reached in which a guest job is valid and the output is from the hyper task. This property corresponds to the desired condition from the semantics because we will always have exactly one output in a period, according to **P1** and **P2**, so a hyper task output being sent means no guest output will be sent in the same period. A similar formula can be written to prove property **P4** as described in Equation 4.

$$AG \neg (jobObserver.Invalid \wedge guestOut) \quad (4)$$

4.3.3 Verification in Uppaal. Table 1 shows the time in seconds taken by Uppaal to verify each of the properties discussed in this section. All experiments were run inside of an Ubuntu VM allowed 4 processors and 16 GB RAM on a host machine with an Intel(R) Core(TM) i7-7820HQ CPU at 2.9GHz with 32 GB RAM. We used Uppaal's select statement to choose a non-deterministic value for E and T as follows. Let i and j be two integers from 1 to 256 and let $E = i$ and $T = i + j$. Uppaal will prove that these properties hold for any E and T within these conditions. Note that these formulas are valid for any values of E and T. However, the verification time will depend on the range of these numbers. For this range of values, we can to verify these properties in less than a minute. The verification cost is largely due to checking the model for so many combinations of period lengths. Note that if we consider fixed values of E and T, then we can prove each property in less than 0.01 seconds.

5 PROTOCOL IMPLEMENTATION CORRECTIONS

The modeling and verification of the synchronization protocol for the mixed-trust framework allowed us to identify a critical flaw in the original implementation [16]. The mixed-trust framework implementation (Figure 6) consists of the ZSRM scheduler [18] running the untrusted components (guest tasks), and uberXMHF [4] hypervisor [39] running the trusted components (hyper tasks) [16].

For the remainder of this section, we focus on the interaction between these two components and the functions `createhyptask()`, `disablehyptask()`, and `guestjobstart()` (see Figure 6). More specifically, we first discuss the flaw in the previous implementation that did not have the `guestjobstart()` call, and then describe the corresponding corrections to the implementation to adhere to the verified model.

5.1 Identifying the flaw

Our methodology to identify this flaw was the following. We started by modeling the original implementation using timed automata and realized that we were not able to prove all properties. We then modified the timed automata model and extend it in a way that all properties were proven. Finally, we used the insights from modeling to correct the flaw as described in Section 5.2.

Our modeling and verification properties are tightly connected with detecting a *start* and *end* of a guest job. However, the original implementation of the mixed-trust framework [16] used a different methodology to detect the completion of a job. Namely, it used a single call from the *LE* to the hypervisor to let it know when the job had been completed and produced the output with the correct value. The timing of this call is then evaluated by the hypervisor to judge whether it was completed on time or not.

The original implementation was able to determine if a job is valid in cases 1, 2, and 5 represented in Figure 3. However, it was not able to detect the invalid guest jobs shown in cases 3 and 4 in Figure 3 where the job starts in one period and finished in the next period. This happens because, when we observe the end of the job, we do not know if it started in that period or the previous one. If the job started in the previous period, it violates the condition **C4** and should be marked as invalid. However, since the previous implementation was only observing when a job ends, it was not possible to detect this case.

5.2 Correcting the flaw

The modeling and verification of the protocol allowed us to discover that it was not possible to distinguish cases 3 and 4 of Figure 3

[4] https://uberxmhf.org/

Table 1: Time in seconds to verify the different properties in UPPAAL

Property	TCTL Formula	Time (s)
P1	$AG \neg (outputObserver.t == P \wedge outputObserver.outs == 0)$	57
P2	$AG \neg (outputObserver.outs > 1 \wedge outputObserver.t < P)$	54
P3	$AG \neg (jobObserver.Valid \wedge hyperOut)$	52
P4	$AG \neg (jobObserver.Invalid \wedge guestOut)$	24

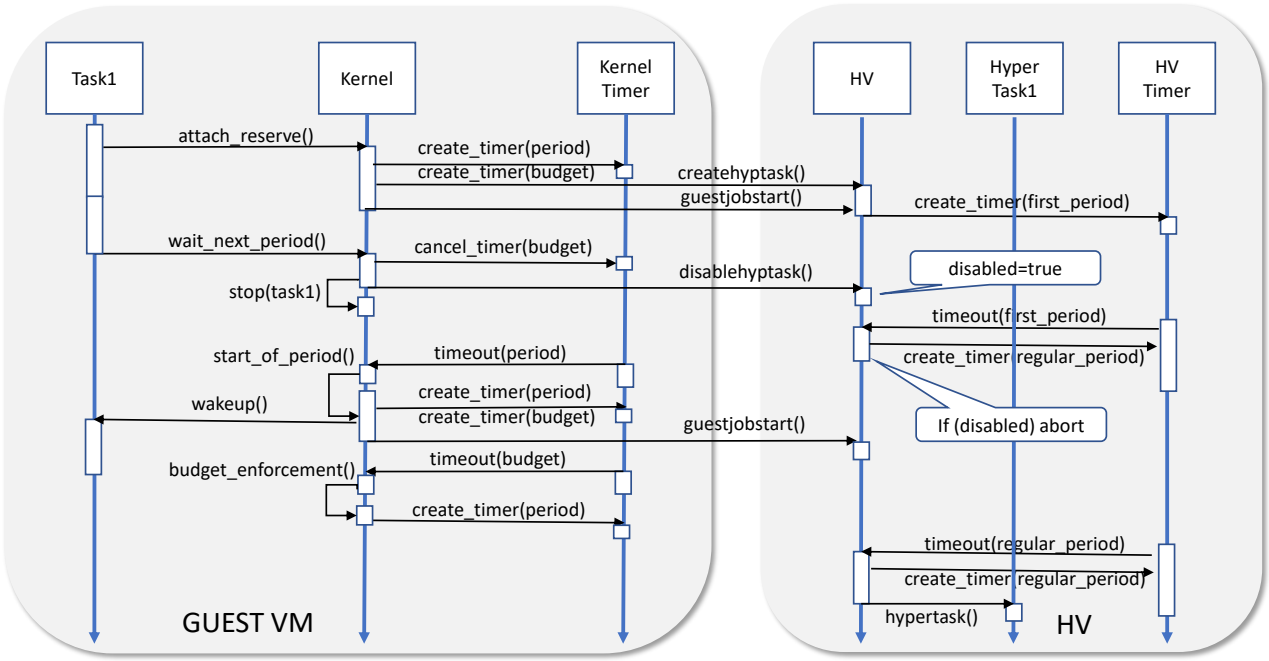


Figure 6: High-level scheme of the implementation and interactions between untrusted (Guest VM) and trusted components (Hypervisor)

without knowing when the job started. Hence, our correction was focused on capturing this event.

5.2.1 Marking the Start of a Job. We mark whether the start of a job is valid or not based on its timestamp in the revised implementation. When the call to mark the end of the job is received, the hypervisor checks if both the timestamps of the start and end of the job are valid, i.e. are in the same period and before the deadline.

Let us discuss the start of the job validation using Listing 1. More specifically, the *LE* calls the `guestjobstart()` in the hypervisor to indicate the start of the job. Within this call, the hypervisor saves the timestamp of the call in the variable `now` (line 4). It evaluates the following two cases to determine if the start of the job is valid:

(1) *First Job.* The first job can be recognized because the `start` variable is equal to zero—meaning undefined in this context. In this case we also check the current timestamp (`now`) is less than the timestamp we recorded when the task was created (`creation`) plus the E_i parameter (recorded in `hyper->t->E`). That is, the job is starting before E_i elapses.

(2) *Other Jobs.* We can identify that we are not validating the first job because the `start` variable is not zero, but has the timestamp of the previous job arrival. In this case we check that the current timestamp (`now`) is larger than the task creation time timestamp (`creation`) plus the T_i parameter (saved in the `hyper->t->T` variable) times the number of periods (saved in the `hyper->num_periods` variable that is incremented every time the enforcement timer in the hypervisor expires). We call this calculated timestamp, the start of the *current* period. In addition, it also checks that the `now` timestamp is smaller or equal to the start of the current period plus the E_i interval.

If either of the two previous conditions is true, the start of the period is marked as valid (line 12), otherwise, it is marked as invalid (line 15). This marking is reset (marked as invalid) whenever the enforcement timer elapses within the hypervisor. This prevents us from incorrectly identifying the end of the job as valid if its corresponding start is before the enforcement timer. This implementation-level reset follows the reset in the hyper task automaton of Figure 4b from the *AfterOutput* location to the initial

Listing 1: Job Arrival Code

```
1  void guestjobstart(ugapp_hypmtscheduler_param_t *hmtsp){
2      int start = hyper->guest_start_t;
3      int creation = hyper->guest_creation_t;
4      now = uapp_sched_read_cpucounter();
5      int current_start_t = creation + (hyper->t->T * hyper->num_periods)
6
7      if ((start == 0 && now <= creation + hyper->t->E) ||
8          (start <= hyper->last_enforcement_t &&
9          now >=  current_start_t &&
10         now <= current_start_t + hyper->t->E))
11     {
12         hyper->valid_guest_mask |= GUEST_JOB_START_VALID_MASK;
13     }
14     else {
15         hyper->valid_guest_mask &= ~GUEST_JOB_START_VALID_MASK;
16     }
17
18     hyper->guest_start_t = now;
19     hmtsp->status = 1; //success
20 }
```

Listing 2: Guest Job Termination Code

```
1  void disablehyptask(ugapp_hypmtscheduler_param_t *hmtsp){
2      int creation = hyper->guest_creation_t;
3      int end = uapp_sched_read_cpucounter();
4      int current_start_t = creation + (hyper->t->T * hyper->num_periods)
5
6      if ((hyper->valid_guest_mask & GUEST_JOB_START_VALID_MASK) &&
7          ((end <= creation + hyper->t->E) ||
8          (end >= current_start_t &&
9          end <= current_start_t + hyper->t->E)))
10     {
11         hyper->valid_guest_mask |= GUEST_JOB_END_VALID_MASK;
12     }
13     else {
14         hyper->valid_guest_mask &= ~GUEST_JOB_END_VALID_MASK;
15     }
16
17     if ((hyper->valid_guest_mask & GUEST_JOB_END_VALID_MASK)) {
18         //do not execute the hyptask function
19         hyptask_timer->disable_tfunc = TRUE;
20     }
21
22     hmtsp->status = 1; //success
23 }
```

location. Similarly, the `hyper->num_periods` variable is also incremented when the enforcement timer elapses as well. Note that the period counter is not needed in the model because clock times are not relative to absolute time read from a processor.

5.2.2 Determining the Validity of the Job. The validity of a job is determined in the hypervisor call that marks the end of the job. This is the `disablehyptask()` of Listing 2 which is intended to disable the execution of the hyper task following a valid job. In this case, we save the current timestamp in the `end` variable and then check if the job start is valid by testing the bitmap (line 6). If valid, we check the current timestamp against two cases:

(1) *First Job.* The first job is detected if the current timestamp is less than the timestamp of the creation of the task plus the E_i parameter (saved in the `hyper->t->E` variable).

(2) *Other Jobs.* In this case, we check that the current timestamp is larger than the start of the current period (`creation + hyper->t->T * hyper->num_periods`, as calculated before) and smaller than the start of the current period plus the E_i parameter.

If the job is valid, then the execution of the hyper task is disabled and the output of the guest task is accepted.

6 RELATED WORK

The mixed-trust framework [16] considers a combination of trusted and untrusted components. This system shares some of the goals of other frameworks such as mixed-criticality scheduling frameworks [10, 18] where behavior is monitored during runtime and action is taken when abnormal behavior is detected. While mixed-criticality frameworks split components into critical and non-critical, the mixed-trust framework considers additional logical and timing challenges that are not covered by mixed-criticality frameworks (Section 2). The mixed-trust framework also shares some similarities with the Simplex [36] architecture for dependable and evolvable process-control systems. In this architecture, the system is composed of a simple controller, a complex controller, and a physical plant. The simple controller is verified and it guarantees that the plant is always in a safe state. In contrast, the complex controller does not need to be verified and can be optimized for performance. One can think of the simpler controller as the hyper task in the mixed-trust framework that guarantees the correct behavior of the entire system.

Verification of timing protocols [37] is a common methodology that precedes the implementation and the creation of a complex system. This can be seen as verifying the blueprint of a system which increases confidence in its correct behavior. When verifying timing protocols, it is usual to model these protocols with formalisms that support the notion of time [8], such as timed automata [2] or timed Petri nets [33].

Timed automata have been used in a plethora of applications, such as, solving planning problems [1, 26, 31], clock synchronization of networks [4, 23, 25], verification of security protocols [28], visual e-contracts [32], and fault diagnosis [7]. When solving scheduling problems for complex systems such as embedded systems, one may consider independent modules of the system that interact with each other using a scheduling hierarchy [19, 20]. Each module consists of a real-time workload and a scheduling policy. The relationship between modules imposes restrictions on the schedulability of the different modules. In this paper, we also have different modules, but our application is unique because we divide these modules into trusted and untrusted components. This simplifies the problem since we can make assumptions on the trusted components. We also do not perform any schedulability analysis, but instead verify that the trusted component correctly observes the behavior of the untrusted component, and acts when necessary to ensure the correct behavior of the system.

Timed Petri nets is an alternative to timed automata that also allow modeling and verifying systems with respect to timing properties [6, 21]. Timed Petri nets and timed automata are closely related to each other and have the same expressiveness for several classes of problems [11, 12].

There are many tools that support the specification and verification of timing properties, such as Uppaal [30], Kronos [9], and nuXmv [14]. In this paper, we used Uppaal [30] since it supports timed automata with additional features such as bounded discrete variables and broadcast channels and which makes it easier to model and verify this problem.

7 CONCLUSIONS AND FUTURE WORK

Complex CPS use is increasingly predominant in today's society. Since many of these systems perform safety-critical functions, it is essential to guarantee their correct behavior. Formal verification is a common approach to guarantee the correct behavior of a system, however, it does not scale to the complexity of CPS. One approach to tackle this problem is done by the mixed-trusted framework [16], where untrusted (unverified) components can be used as long as they are monitored by trusted (verified) components. However, if the synchronization protocol between trusted and untrusted components is faulty, then it can compromise the timing and trust properties of the entire system.

In this paper, we modeled the synchronization protocol in Uppaal and proved temporal properties that show the correctness of this protocol. The verification of this protocol is then used to identify and correct a critical flaw in the original implementation [16]. This work supports the claim from Leslie Lamport, that no one should build a house without first drawing a blueprint [29]. In systems where not all parts are verified, it is critical to verify the interaction between trusted and untrusted components. In this paper, we successfully verified the synchronization protocol between trusted and untrusted components that guarantees the safe use of untrusted components in critical CPS functions.

As future work, we propose to extend this framework to ensure that the protocol implementation matches the Uppaal model. Even though the verification of the current model allowed us to find and correct bugs, there are no formal guarantees that the code satisfies the properties described in Section 4.3. To achieve this goal, we will verify the same verification properties at the code level and ensure that these properties are semantically equivalent to the verification conditions in Uppaal.

ACKNOWLEDGMENTS

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

REFERENCES

[1] Yasmina Abdeddaïm, Eugene Asarin, and Oded Maler. 2006. Scheduling with timed automata. *Theor. Comput. Sci.* 354, 2 (2006), 272–300.

[2] Rajeev Alur and David L. Dill. 1990. Automata For Modeling Real-Time Systems. In *ICALP (Lecture Notes in Computer Science)*, Vol. 443. Springer, 322–335.

[3] Björn Andersson, Sagar Chaki, and Dionisio de Niz. 2017. Combining Symbolic Runtime Enforcers for Cyber-Physical Systems. In *RV (Lecture Notes in Computer Science)*, Vol. 10548. Springer, 68–84.

[4] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. 2014. Modeling Bitcoin Contracts by Timed Automata. In *FORMATS (Lecture Notes in Computer Science)*, Vol. 8711. Springer, 7–22.

[5] Gerd Behrmann, Alexandre David, and Kim Guldstrand Larsen. 2004. A Tutorial on Uppaal. In *SFM (Lecture Notes in Computer Science)*, Vol. 3185. Springer, 200–236.

[6] Bernard Berthomieu and Michel Diaz. 1991. Modeling and Verification of Time Dependent Systems Using Time Petri Nets. *IEEE Trans. Software Eng.* 17, 3 (1991), 259–273.

[7] Patricia Bouyer, Samy Jaziri, and Nicolas Markey. 2018. Efficient Timed Diagnosis Using Automata with Timed Domains. In *RV (Lecture Notes in Computer Science)*, Vol. 11237. Springer, 205–221.

[8] Patricia Bouyer, François Laroussinie, Nicolas Markey, Joël Ouaknine, and James Worrell. 2017. Timed Temporal Logics. In *Models, Algorithms, Logics and Tools (Lecture Notes in Computer Science)*, Vol. 10460. Springer, 211–230.

[9] Marius Bozga, Conrado Daws, Oded Maler, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. 1998. Kronos: A Model-Checking Tool for Real-Time Systems. In *CAV (Lecture Notes in Computer Science)*, Vol. 1427. Springer, 546–550.

[10] Alan Burns and Robert I. Davis. 2018. A Survey of Research into Mixed Criticality Systems. *ACM Comput. Surv.* 50, 6 (2018), 82:1–82:37.

[11] Joakim Byg, Morten Jacobsen, Lasse Jacobsen, Kenneth Yrke Jørgensen, Mikael Harkjær Møller, and Jirí Srba. 2014. TCTL-preserving translations from timed-arc Petri nets to networks of timed automata. *Theor. Comput. Sci.* 537 (2014), 3–28.

[12] Franck Cassez and Olivier H. Roux. 2006. Structural translation from Time Petri Nets to Timed Automata. *J. Syst. Softw.* 79, 10 (2006), 1456–1468.

[13] Sagar Chaki and Dionisio de Niz. 2017. Formal Verification of a Timing Enforcer Implementation. *ACM Trans. Embedded Comput. Syst.* 16, 5s (2017), 168:1–168:19.

[14] Alessandro Cimatti, Alberto Griggio, Enrico Magnago, Marco Roveri, and Stefano Tonetta. 2019. Extending nuXmv with Timed Transition Systems and Timed Temporal Properties. In *CAV (Lecture Notes in Computer Science)*, Vol. 11561. Springer, 376–386.

[15] Dionisio de Niz, Björn Andersson, Hyoseung Kim, Mark H. Klein, and John P. Lehoczky. 2020. Work-in-Progress: Toward Precomputation in Real-Time Mixed-Trust Scheduling. In *Brief Presentation Session of IEEE Real-Time Systems Symposium (RTSS)*. IEEE.

[16] Dionisio de Niz, Björn Andersson, Mark H. Klein, John P. Lehoczky, A. Vasudevan, Hyoseung Kim, and Gabriel A. Moreno. 2019. Mixed-Trust Computing for Real-Time Systems. In *RTCSA*. IEEE, 1–11.

[17] Dionisio de Niz, Bjorn Andersson, and Gabriel Moreno. 2018. Safety enforcement for the verification of autonomous systems. In *Autonomous Systems: Sensors, Vehicles, Security, and the Internet of Everything*, Vol. 10643. International Society for Optics and Photonics, SPIE, 1 – 10.

[18] Dionisio de Niz, Karthik Lakshmanan, and Ragunathan Rajkumar. 2009. On the Scheduling of Mixed-Criticality Real-Time Task Sets. In *RTSS*. IEEE Computer Society, 291–300.

[19] Arvind Easwaran, Insup Lee, Insik Shin, and Oleg Sokolsky. 2007. Compositional Schedulability Analysis of Hierarchical Real-Time Systems. In *ISORC*. IEEE Computer Society, 274–281.

[20] Arvind Easwaran, Insik Shin, Oleg Sokolsky, and Insup Lee. 2006. Incremental schedulability analysis of hierarchical real-time components. In *EMSOFT*. ACM, 272–281.

[21] Kumkum Garg. 1985. An Approach to Performance Specification of Communication Protocols Using Timed Petri Nets. *IEEE Trans. Software Eng.* 11, 10 (1985), 1216–1225.

[22] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *OSDI*. USENIX Association, 653–669.

[23] Faranak Heidarian, Julien Schmaltz, and Frits W. Vaandrager. 2009. Analysis of a Clock Synchronization Protocol for Wireless Sensor Networks. In *FM (Lecture Notes in Computer Science)*, Vol. 5850. Springer, 516–531.

[24] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. 1994. Symbolic Model Checking for Real-Time Systems. *Inf. Comput.* 111, 2 (1994), 193–244.

[25] Xiaowan Huang, Anu Singh, and Scott A. Smolka. 2011. Using integer clocks to verify clock-synchronization protocols. *ISSE* 7, 2 (2011), 119–130.

[26] Lina Khatib, Nicola Muscettola, and Klaus Havelund. 2001. Mapping Temporal Planning Constraints into Timed Automata. In *TIME*. IEEE Computer Society, 21–27.

[27] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: formal verification of an OS kernel. In *SOSP*. ACM, 207–220.

[28] Ahmet Koltuksuz, Burcu Kulahcioglu, and Murat Ozkan. 2010. Utilization of Timed Automata as a Verification Tool for Security Protocols. In *SSIRI (Companion)*. IEEE Computer Society, 86–93.

[29] Leslie Lamport. 2015. Who builds a house without drawing blueprints? *Commun. ACM* 58, 4 (2015), 38–41.

[30] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. 1997. UPPAAL in a Nutshell. *Int. J. Softw. Tools Technol. Transf.* 1, 1-2 (1997), 134–152.

[31] Georgiana Macariu and Vladimir Cretu. 2010. Timed Automata Model for Component-Based Real-Time Systems. In *ECBS*. IEEE Computer Society, 121–130.

[32] Enrique Martínez, María-Emilia Cambronero, Gregorio Díaz, and Gerardo Schneider. 2011. Timed Automata Semantics for Visual e-Contracts. In *FLACOS (EPTCS)*, Vol. 68. 7–21.

[33] Louchka Popova-Zeugmann. 2013. *Time and Petri Nets*. Springer.

[34] RTCA Special Committee. 2012. DO-178C, software considerations in airborne systems and equipment certification.

[35] Robert G. Sargent. 1998. Verification and Validation of Simulation Models. In *WSC*. WSC, 121–130.

[36] Lui Sha. 2001. Using Simplicity to Control Complexity. *IEEE Softw.* 18, 4 (2001), 20–28.

[37] A. Udaya Shankar and Simon S. Lam. 1987. Time-Dependent Distributed Systems: Proving Safety, Liveness and Real-Time Properties. *Distributed Comput.* 2, 2 (1987), 61–79.

[38] Amit Vasudevan, Sagar Chaki, Limin Jia, Jonathan M. McCune, James Newsome, and Anupam Datta. 2013. Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework. In *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 430–444.

[39] Amit Vasudevan, Sagar Chaki, Petros Maniatis, Limin Jia, and Anupam Datta. 2016. überSpark: Enforcing Verifiable Object Abstractions for Automated Compositional Security Analysis of a Hypervisor. In *USENIX Security Symposium*. USENIX Association, 87–104.