VeriRT: An End-to-End Verification Framework for Real-Time Distributed Systems

YOONSEUNG KIM, Seoul National University, South Korea and Yale University, USA
SUNG-HWAN LEE*, Seoul National University, South Korea
YONGHYUN KIM, Seoul National University, South Korea
CHUNG-KIL HUR, Seoul National University, South Korea

Safety-critical systems are often designed as real-time distributed systems. Despite the need for strong guarantees of safety and reliability in these systems, applying formal verification methods to real-time distributed systems at the implementation level has faced significant technical challenges.

In this paper, we present VeriRT, an end-to-end formal verification framework that closes the formal gap between high-level abstract timed specifications and low-level implementations for real-time distributed systems. Within the framework, we establish a theoretical foundation for constructing formal timed operational semantics by integrating conventional operational semantics and low-level timing assumptions, along with principles for reasoning about their timed behaviors against abstract specifications. We leverage CompCert's correctness proofs to guarantee the correctness of the assembly implementation of real-time distributed systems. We provide two case studies on realistic real-time systems. All the results are formalized in Coq.

CCS Concepts: • **Theory of computation** → **Operational semantics**; **Program verification**; **Distributed computing models**; • **Computer systems organization** → **Real-time system specification**.

Additional Key Words and Phrases: formal verification, real-time systems, distributed systems, refinement

ACM Reference Format:
Yoonseung Kim, Sung-Hwan Lee, Yonghyun Kim, and Chung-Kil Hur. 2025. VeriRT: An End-to-End Verification Framework for Real-Time Distributed Systems. *Proc. ACM Program. Lang.* 9, POPL, Article 61 (January 2025), 28 pages. https://doi.org/10.1145/3704897

1 Introduction

Safety-critical systems, where ensuring safety and reliability is the top-priority task, are often designed as real-time distributed systems. Examples include autonomous vehicle systems, avionics systems, and nuclear systems [Chen et al. 2017; Gawand et al. 2017; Sampigethaya and Poovendran 2012], whose reliability critically depends on the prompt detection of environmental changes and timely responses. Moreover, they fall under the category of cyber-physical systems, where many of which adopt a distributed design to accommodate physical limitations [Khaitan and McCalley 2015; Shi et al. 2011].

However, applying formal verification methods directly to real-time distributed systems to achieve a high level of reliability poses considerable technical challenges. Even the verification of

*Now at Rebellions Inc.

Authors' Contact Information: Yoonseung Kim, Seoul National University, Seoul, South Korea and Yale University, New Haven, USA, yoonseung.kim@yale.edu; Sung-Hwan Lee, Seoul National University, Seoul, South Korea, sunghwan.lee@sf.snu.ac.kr; Yonghyun Kim, Seoul National University, Seoul, South Korea, yonghyun.kim@sf.snu.ac.kr; Chung-Kil Hur, Seoul National University, Seoul, South Korea, gil.hur@sf.snu.ac.kr.

ACM 2475-1421/2025/1-ART61
https://doi.org/10.1145/3704897

non-real-time distributed systems [Drăgoi et al. 2016; Hawblitzel et al. 2015; Honoré et al. 2021; Sergey et al. 2017; v. Gleissenthall et al. 2019; Wilcox et al. 2015] using conventional methods is acknowledged to be challenging, as it involves proving system-wide invariants under arbitrary interleavings of distributed nodes. Furthermore, verifying real-time systems necessitates quantitative reasoning about the specific timings of observable events, a requirement not supported by conventional methods that represent timing information for events in an abstract sequential order.

In our attempt to develop a new foundation for the real-time distributed system verification, we have faced the following challenges:

Modeling Timed Behaviors. In traditional approaches that only consider sequences of events without precise timing, the language semantics of a program (whether at the source, machine, or specification level) fully determines the sequence of observable events. However, when timing is introduced, defining program semantics that accurately predict the timing of each event becomes challenging. This difficulty arises because while compilers are expected to maintain the order of observable events from source to target code, they are not required to preserve exact timing. In fact, compiler optimizations typically aim to reduce execution time, which inevitably alters the timing of events. This discrepancy complicates the development of a semantic model that can reliably predict the timing of events in optimized code.

Quantitative Reasoning about Timing. The correctness of real-time systems hinges on both their computational behavior and timing characteristics, creating an intrinsic interdependence in correctness reasoning. To illustrate this, consider a system that launches tasks at regular intervals using a system call like `sleep`. A computational bug leading to an incorrect calculation of the sleep duration could result in job launches deviating from the specified timing (*i.e.,* a timing bug). Conversely, a delayed job launch (*i.e.,* a timing bug) might cause a message to be transmitted to another node too late, potentially disrupting the entire system's functionality (manifesting as a computational bug). Thus, we require a well-crafted verification technique that enables quantitative reasoning about both timing and computational behaviors in tandem.

Complexities Arising from Distributed Computation. Verifying distributed systems presents additional challenges due to an inherently high degree of nondeterminism. Three main factors contribute to this complexity: (*i*) real-time interleaving of operations across physical nodes comprising the system, (*ii*) each node's local clock exhibiting randomly varying skew, and (*iii*) message exchanges between nodes subject to random delays, drops, or duplications. To ensure the overall system's correctness, we first need a distributed system model that conservatively captures all possible behaviors arising from these sources of nondeterminism. Furthermore, we need reasoning principles that allow us to effectively manage this complex nondeterminism across multiple nodes.

In this paper, we present VeriRT, a novel formal verification framework for real-time distributed systems, addressing these challenges with the following features.

First, we develop a novel theory for modeling the timed behaviors of programs. In our approach, we treat the timing of events differently from the ordering of events. While the ordering is determined by the semantics of a program at different levels, the timing is only determined by the actual machine code running on hardware. Therefore, unlike the ordering of events, we allow users to specify timing conditions for observable events generated by the actual machine code, which is set out as a separate verification condition that should be externally validated via empirical methods or worst-case execution time (WCET) analyses against the compiled machine code. Then, we reflect these timing conditions in higher-level operational semantics in the form of timing assumptions. For this, we develop the notion of *timed operational semantics* that combines conventional untimed

operational semantics with user-specified timing assumptions, generating observable behaviors in the form of a sequence of *timed events*. It is important to note that imposing timing assumptions in high-level operational semantics is necessary because computational behaviors can depend on these timing assumptions, as discussed earlier.

Second, we develop proof techniques for reasoning about the timed behaviors of programs. For this, we introduce *timed simulation relations* that enable timing-sensitive reasoning to prove behavioral refinement between the timed behaviors of two programs. Moreover, we provide a lifting theorem that allows for timing-insensitive reasoning: given a conventional simulation proof between (untimed) behaviors of two programs, we can lift this simulation into a timed simulation between their timed behaviors with the same timing assumptions. This timing-insensitive reasoning is typically applied to compiler verification, in our work specifically for CompCert [Leroy 2009a], a realistic C compiler formally verified in Coq.

Third, to address the challenge regarding distributedness, our framework supports specifying and reasoning about the behavior of local clocks and the network. Specifically, our distributed system model assumes a virtual global clock as a reference, allowing users to specify the behavior of local clocks in relation to this global clock. In our case studies, we consider two different specifications of local clock behavior. For the network, users can adjust a set of parameters to express various assumptions regarding its behavior. The framework then provides *node-local* and *global* timed simulations as proof techniques to resolve nondeterminism. Node-local simulation facilitates abstraction of node-local timed behavior, including the local clock, while global simulation can be applied to deal with interleavings and network behavior. It is important to note that the assumptions about local clock and network behaviors form part of our trust base and are subject to external validation.

This paper includes two case studies that demonstrate the application of VeriRT. In the first case study, we implement and verify a well-known clock synchronization mechanism called Cristian's algorithm. We verify that the clock skews are bounded within the specified range under appropriate assumptions about the local clock's hardware behavior. The second case study presents a simplified implementation of real-time system middleware called PALSware, which provides a logically synchronous environment to applications built on top of a physically asynchronous network. We verify the correctness of this PALSware implementation under appropriate assumptions about the local clock and network behavior.

Additionally, we prove that the entire transformation chain of CompCert 3.9 preserves our timed operational semantics from the C level down to the assembly level. We prove that the generic simulation relation of CompCert also establishes refinement under our timed operational semantics, thanks to the aforementioned lifting theorem. This extension of CompCert's correctness proofs to our timed semantics is packaged as CompCertRT within our framework. Consequently, CompCertRT enables end-to-end verification from an abstract specification down to a final executable.

All results are formalized in Coq.

The remainder of the paper is organized as follows: §2 provides an overview of the theoretical foundation of VeriRT and outlines the overall structure of the framework. §3 introduces our formalism for representing real-time distributed systems, timed behaviors, and refinement. §4 offers a detailed explanation of the formal distributed system model, including our models for the network and operating systems. §5 presents the proof techniques offered by VeriRT and the rationale behind their designs. §6 describes the construction of CompCertRT, an integration of CompCert within our framework. §7 and §8 present two case studies illustrating the practical application of the framework. §9 evaluates our development. Finally, §10 concludes the paper by comparing our work with existing studies.

2 Overview

This section introduces the core concept underlying our theory for modeling and reasoning about timed behaviors of programs. We illustrate this concept with an example program that operates under real-time constraints. Next, we present the structure of VeriRT, a formal verification framework for real-time distributed systems, built upon this theoretical foundation.

2.1 Modeling Timed Behaviors of Programs

Ordered Events vs. Timed Events. Program semantics define observable behaviors, each comprising events such as I/O operations with associated timings. Traditional approaches typically represent these timings as a sequential order of events, without specifying exact times. However, for real-time system verification, we cannot abstract away precise timings, as such verification demands quantitative reasoning about when events occur.

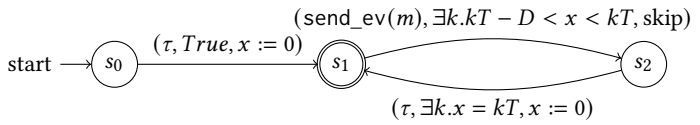
Let's first review traditional approaches to semantics with ordered events. In these approaches, language semantics can be designed as an abstract state machine. This machine processes a given program step by step, generating observable events in a sequential order. Here, a program serves as a standalone representation for the operational semantics. Compilers can legitimately transform these programs into various representations, provided they maintain the original order of events.

However, the traditional approaches face challenges when addressing timed operational semantics — semantics that generate events with precise timing values. Consider, for example, designing a standalone timed operational semantics for C (such as assigning an execution time bound for each command). In this scenario, ensuring compiler correctness would require preserving not only the order of events but also their precise timings. This approach presents technical difficulties in compiler development and doesn't align with how realistic compilers function. Compilers typically aim to reduce execution time or, in some cases, extend it for specific purposes (*e.g.,* security or file size optimization). It's widely recognized that source programs are only loosely connected to actual execution time (*e.g.,* time complexity). Consequently, the verification of precise time constraints must occur at the machine level, involving compiled executables and the target machine.

Our Approach to Modeling Timing. To construct a timed operational semantics, our method requires the user to provide timing assumptions for a program. These assumptions consist of time constraints regarding the interactions between the program and its environment (*i.e.,* invocations and returns of system calls). We then form a timed operational semantics by integrating these timing assumptions with the existing (untimed) operational semantics.

Fig. 1 illustrates an example program with its abstract specification. The system's primary task is to send a message periodically at a specified time. Each iteration involves two steps: first, computing a message (a process that may consume time), and then initiating message transmission within the time interval $(kT - D, kT)$. Here, T represents a particular period, $D(< T)$ is a given constant, and k is any integer. Such constraints are typical in time-division multiplexing where communication channels are allocated specific time slots.

Fig. 1a presents the abstract specification of this process as a timed automaton [Alur and Dill 1994]. Each edge is labeled with a triple (e, C, U) where e is an observable event, C is a condition for the transition to occur, and U is an operation that updates the clock variables after the transition. The process begins at s_0 and may transition to s_1 at any moment, resetting a clock x. While at s_1, the system may emit an event $\texttt{send\_ev}(m)$ when the clock x is within a specified time interval. For simplicity, the specification disregards the message content. After sending a message, the system can return to s_1, resetting x to 0 when its value becomes a multiple of T, ensuring the next iteration follows the correct periodic schedule. The automaton accepts any observable behavior generated by an infinite transition sequence that indefinitely visits s_1 (a Büchi condition).



(x: a clock variable)

(a) Automata specification

```
1  void main_loop() {
2    while (true) {
3      Msg m = compute_msg();
4
5      Int t_cur = get_time(); /** (a1,r1,s1,e1) |= [SC: 0<=e1-s1<Egt /\ s1<=r1<=e1] [WCET: s2-e1<E1] */
6      Int t_send = ((t_cur / T) + 1) * T - E;
7      if (t_send < t_cur)
8        t_send += T;
9
10     Int t_sleep = t_send - t_cur;
11     sleep(t_sleep);          /** (a2,r2,s2,e2) |= [SC: a2<=e2-s2<a2+Esleep] [WCET: s3-e2<E2]       */
12
13     send(m);                 /** (a3,r3,s3,e3) |= [SC: 0<=e3-s3<Esend] [WCET: True]                */
14   }
15 }
```

(b) C implementation

Fig. 1. Specification and implementation of a periodic process

Fig. 1b presents a C function `main_loop` that implements the process. Let's first examine the code, disregarding comments. The function enters a loop and begins by computing a message via `compute_msg` (whose detailed behavior is not relevant for this example). It then obtains the current local clock value `t_cur` from the system call `get_time`. Next, it calculates k = `((t_cur / T) + 1)` and computes `t_send` $= kT - E$ (to be explained later) as the initial candidate time to start sending the message. If kT occurs too soon, rendering `t_send` earlier than `t_cur`, the function may need to wait another period T. It then calls the system call `sleep` to pause execution until `t_send`, and finally sends the message through `send(m)`. We consider the event `send_ev`(m) to occur between the invocation and return of `send(m)`.

Now, we explain how we define the timing assumptions. We use annotations of the form `(a,r,s,e) |= [SC: C1] [WCET: C2]` to clearly present time constraints associated with a system call. The variables `(a,r,s,e)` denote the argument, return value, start time, and end time of the associated system call for each invocation, respectively. Users can provide two types of constraints: the SC constraint imposed on the system call itself, and the WCET constraint on the code between system calls. In the WCET constraint, users may reference variables declared in constraints of other system calls. Importantly, these constraints are expressed relative to the local clock. Here, we assume a value ε as the maximum clock skew between the local and global clocks[1].

In this example, we have three sets of constraints associated with `get_time`, `sleep`, and `send`. For `get_time`, the SC constraint specifies that the system call's execution time is bounded by `Egt`, and the return value must fall between the start and end times. The WCET constraint requires that the subsequent `sleep` system call invocation occurs within `E1` time after `get_time` returns. This means that the intermediate steps, including `t_send` and `t_sleep` computations, must complete within `E1`. For `sleep`, the SC constraint mandates that the blocking time is at least the given argument, with the call returning within `Esleep`. Its WCET constraint limits the time between `sleep` and `send` (involving no C commands in the source code) to at most `E2`. Lastly, the SC constraint for `send`

[1]VeriRT may support different forms of assumptions on the local clock, as we explain in the case studies.

$$\text{(Latency)}\quad \frac{t < to \quad 0 < lat \quad lat' = lat - 1}{(t, tr, lat, to, s) \xrightarrow{(t,\tau)} (t+1, tr, lat', to, s)}$$

$$\text{(Progress)}\quad \frac{s \xrightarrow{e}{}^{+} s' \quad tr' = \text{app\_last}(tr, t, e) \quad t < to \quad to' = \mathcal{T}(tr, t, e, to)}{(t, tr, 0, to, s) \xrightarrow{(t,e)} (t+1, tr', lat', to', s')}$$

$$\text{(TimeViolation)}\quad \frac{to \le t}{(t, tr, lat, to, s) \xrightarrow{(t,\text{NB})} *}$$

Fig. 2. Timed transition rules

specifies a maximum execution time of `Esend`. We do not impose any WCET condition between `send` and the subsequent `get_time`.

Construction of Timed Operational Semantics. We now present the generic construction of timed operational semantics shown in Fig. 2, explaining it in two steps.

First, our approach transforms each transition $s \xrightarrow{e} s'$ from the underlying untimed semantics into a timed transition $(t, s) \xrightarrow{(t,e)} (t', s')$, generating a timed event at time t and accounting for time passage until the next transition at $t' \geq t$. In this framework, we discretize time into extremely small units (e.g., 10^{-15} seconds), parameterized in our formalization, allowing us to model a transition step for each time unit. We introduce a latency state lat to represent the physical time required before the next transition occurs. Two key rules govern this process: the Latency rule, which decreases lat by one per step, generating a silent event τ; and the Progress rule, which captures the state transition when lat reaches zero, represented as $s \xrightarrow{e}{}^{+} s'$. After a Progress step, a new latency lat' is nondeterministically assigned. Importantly, $s \xrightarrow{e}{}^{+} s'$ allows for multiple silent steps to occur along with a single, possibly non-silent event e, a feature particularly useful for maintaining conciseness in our theory (see §4.2 for more details).

Second, to eliminate executions that violate the timing assumptions, we augment the state with two elements: an event trace tr and a timeout point to. Initially, to is set to ∞. As events occur, the $\mathcal{T}$ function, which encodes the timing assumptions, may update the timeout point. For instance, if tr and the current event e indicate a return from `get_time()` at Line 5 of our example program, to' is updated to $t +$ `E1`. The system can take arbitrary timed steps following the Latency and Progress rules until t reaches to. If t does reach to, the program generates a special *no-behavior* event, marking the execution as "invalid" (this is the TimeViolation rule). As an example, if the execution time exceeds `E1` after returning from `get_time` without reaching `sleep`, the semantics generates a no-behavior event. When defining the system's observable behaviors, we discard these invalid executions. This approach assumes that the timing assumptions will be validated against the final executable (see §3 for details). For simplicity in this explanation, we have not distinguished between local and global clocks. This distinction will be addressed in detail in §4.

2.2 Reasoning about Timed Behaviors

For a single program, the end-to-end verification from an abstract specification to the machine-level timed behavior comprises three layers, as illustrated in Fig. 3:

- Refinement proof: This layer establishes the relationship between the C implementation (including OS interactions) and the abstract specification.
- Compiler correctness: This layer ensures that the compiled code preserves the timed operational semantics of the source program.
- Timing assumption validation: This layer confirms that the user-defined timing assumptions are satisfied by the actual machine-level execution.

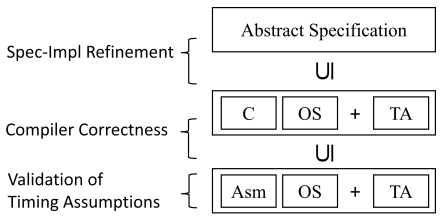


Fig. 3. Verification layers of a single program

Of the three layers, the framework directly provides the middle one concerning compiler correctness. A key theorem of VeriRT guarantees that when CompCert compiles a C program to an assembly program, the C program with its timing assumptions is refined by the resulting assembly program with the identical timing assumptions. The timing assumptions are designed to be independent of the underlying untimed operational semantics. This independence allows us to reuse CompCert's existing per-pass simulation proofs by applying our *lifting theorem*. We will present the details of this lifting theorem in §5 and our work on applying it to CompCert 3.9 in §6.

The user's responsibility for reasoning about timed behaviors is divided into two distinct tasks, represented by the top and bottom layers respectively. We will illustrate each of these tasks using the example provided in Fig. 1 in the following paragraphs.

Refinement between Specification and Implementation. The verification process linking the abstract specification to the C implementation necessitates combined reasoning about both the code's computational behaviors and the timing assumptions. Consider a scenario where the developer inadvertently introduces an error in Fig. 1b, causing the computation of `t_sleep` to result in either an excessively short or long sleep duration. In a traditional approach using ordered events to represent timings, such a mistake would go undetected during verification, as it wouldn't alter the event sequence. In contrast, our approach allows for the explicit expression of timings in the specification, enabling proper verification of real-time systems.

We now present the high-level reasoning for verifying our example. Our goal is to demonstrate that the implementation meets the timed automata specification (Fig. 1a), which defines the safe time range as $\exists k.(kT - D, kT)$. Specifically, we need to prove that every occurrence of `s3` and `e3` falls within this range for some k. To establish this, we begin by assuming the following condition on $\mathtt{E}$: $\mathtt{Egt}+\mathtt{E1}+\mathtt{Esleep}+\mathtt{E2}+\mathtt{Esend}+\varepsilon < \mathtt{E} < D-\varepsilon$ This condition forms the basis of our refinement proof. It's important to note that during the verification process, we must prove this condition holds for the concrete values of all constants involved. Our reasoning process integrates three key elements: the code itself, the timing assumptions, and the clock skew of the local clock. [2]

To begin, we will establish a lower bound for the variable `s3` within a single iteration of the main loop. By reasoning about the computational behaviors, we can obtain that `t_cur` < `t_send` and $\mathtt{t\_send} = kT - \mathtt{E}$ for some k. Then, we get a lower bound from the following steps:

$$\begin{aligned}
\mathtt{s3} \geq \mathtt{e2} \geq \mathtt{a2} + \mathtt{s2} &= (kT - \mathtt{E}) - \mathtt{t_cur} + \mathtt{s2} && \text{(from execution order, SC of sleep, and code)}\\
&\geq (kT - \mathtt{E}) - \mathtt{e1} + \mathtt{s2} \geq kT - \mathtt{E} && \text{(from SC of get_time and execution order)}
\end{aligned}$$

Then, considering the maximum skew ε, the earliest possible invocation of `send` in terms of the global clock must be at least $kT - \mathtt{E} - \varepsilon > kT - D$.

Second, we obtain an upper bound of `e3` for the same iteration as follows:

$$\begin{aligned}
\mathtt{e3} < \mathtt{s3} + \mathtt{Esend} < \mathtt{e2} + \mathtt{E2} + \mathtt{Esend} && \text{(from SC of send and WCET of sleep)}\\
< \mathtt{E2} + \mathtt{Esend} + \mathtt{s2} + \mathtt{a2} + \mathtt{Esleep} && \text{(from SC of sleep)}\\
\mathtt{s2} + \mathtt{a2} < \mathtt{E1} + \mathtt{e1} + (\mathtt{t_send} - \mathtt{t_cur}) && \text{(from WCET of get_time and code)}\\
< \mathtt{E1} + \mathtt{Egt} + \mathtt{s1} + ((kT - E) - \mathtt{r1}) && \text{(from SC of get_time)}\\
\leq \mathtt{E1} + \mathtt{Egt} + \mathtt{s1} + ((kT - E) - \mathtt{s1}) && \text{(from SC of get_time)}\\
= \mathtt{E1} + \mathtt{Egt} + (kT - E)
\end{aligned}$$

Again, considering ε, the latest possible return time of `send` is at most $\mathtt{e3} + \varepsilon < \mathtt{Egt} + \mathtt{E1} + \mathtt{Esleep} + \mathtt{E2} + \mathtt{Esend} + (kT - E) + \varepsilon < kT$. Hence the program with the timing assumptions refines the automata specification.

[2] For distributed systems, we would also need to account for message delivery times.

Validation of Timing Assumptions. Timing assumptions should be validated against the compiled executable on the target hardware with respect to its local clock. This validation ensures that the timed behaviors of the assembly code, as defined by our timed operational semantics, encompass all possible physical behaviors. It's important to note that this validation process is conducted outside our framework, as it relies on hardware-specific factors not modeled within our system. Users have flexibility in choosing validation methods based on their required confidence level. Options include rigorous testing or the use of Worst-Case Execution Time (WCET) analysis tools. Some compositional WCET analyses [Leveque et al. 2011; Marref 2010; Maxim et al. 2017] may be particularly helpful, allowing users to validate constraints on system calls and user-written code separately, then combine these results.

2.3 Structure of the Framework

Building upon the theoretical foundation we have discussed, VeriRT enables the establishment of end-to-end refinement at the distributed system level, as illustrated in Fig. 4. The framework's most distinctive feature is its ability to divide the end-to-end verification process into multiple layers. This layered approach significantly simplifies the overall proof by allowing users to focus on reasoning about individual, local components within each layer. This modular verification strategy enhances both the manageability and scalability of the verification process for complex systems.

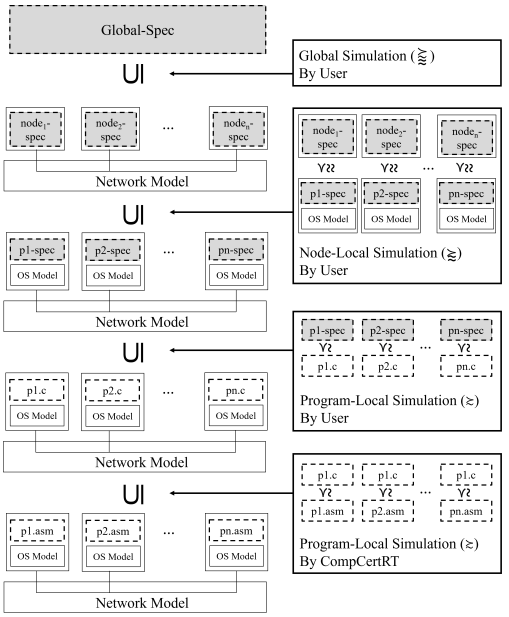


Fig. 4. Structure of the framework

The verification process begins with the user constructing a formal model of the distributed system. This is achieved by writing programs in C language, which are represented as white dashed boxes in Fig. 4. Our framework then integrates these user-provided programs with its pre-built network and operating system models. This integration results in a comprehensive system that serves as the target for verification.

After constructing the formal model, the user proceeds with verification by building a series of refinement proofs. To facilitate this process, the user can define local specifications for individual system components, serving as intermediate steps in the verification. Our framework supports this approach by offering several simulation techniques that enable localized reasoning. In Fig. 4, we use gray shading to represent all specifications, illustrating the gradual abstraction process within our framework. The verification culminates in an end-to-end refinement, resulting from a vertical composition of the individual refinement proofs.

Network and OS Models. The network and OS models are designed to represent all possible behaviors of their real-world counterparts. For the network model, the user may control its behaviors by setting the model parameters μ and κ, each denotes the maximum delivery time assumption and the maximum number of possible message duplication, respectively. The OS model contains the modeled behaviors of selected system calls, about local clock accesses, timers, and network sockets. The user may control the local clock's behavior by imposing a rule (*e.g.,* imposing the maximum clock skew bound). We will explain the two models in detail in §4.

Refinement Layers. The end-to-end verification shown in Fig. 4 is a composition of four refinement layers of proof. The framework provides three simulation relations as proof techniques: program-local, node-local, and global simulations. Global simulation is the most general one, and the other two are for reasoning about local components, which imply contextual refinement for the whole system. We will explain the proof techniques in §5. We summarize each layer along the bottom-up direction as follows.

- **Program Simulation by CompCertRT:** For any C programs given by the user, CompCertRT guarantees the program-local simulation between the C programs and their corresponding assembly programs compiled by itself. Program-local simulation is used to match the computational behavior of two programs. CompCertRT contains a generic proof that every translation of CompCert 3.9 satisfies program-local simulation, whose details will be presented in §6.
- **Program Simulation by User:** In this layer, the user provides a computational specification for each C program and proves program-local simulation to abstract away concrete details of the formal C operational semantics. The removal of the complex details helps the user to focus on reasoning about times in the above layers.
- **Node-Local Simulation by User:** This layer performs abstraction at the node layer, *i.e.,* showing that the user programs with the OS model together behave as a more abstract timed model (*e.g.,* Fig. 1a). Thus we use node-local simulation, which requires reasoning about timings.
- **Global Simulation by User:** Finally, the user may combine all node models and the network model to obtain the refinement until a global-level specification. This layer requires global reasoning, and we can use global simulation to match two distributed system models.

3 Timed Behaviors of Real-Time Distributed Systems

In our formalization, we represent a distributed system as a transition system generating timed events (*i.e.,* events paired with timestamps) from each node. To address special situations triggering errors or violating timing constraints, we introduce the notions of undefined behavior (UB) and no-behavior (NB) events. Observable behaviors of a system are defined as infinite traces formed by transitions. Then, the refinement relation is straightforwardly defined as a subset relation of timed behaviors between two systems. We also explain how the framework supports imposing different assumptions using the event classification mechanism. For precise type definitions in this section, we direct readers to [Kim et al. 2024].

3.1 Distributed System Model

We represent a distributed system as a transition system of a triple $(\Sigma, \rightarrow, I)$: a set of states Σ, a transition relation $\rightarrow$, and initial states $I \subseteq \Sigma$. A transition $\sigma \xrightarrow{\vec{tr}} \sigma'$ generates a distributed trace $\vec{tr}$, in which the i'th node-local trace $\vec{tr}[i] = [(t_1, e_1), ..., (t_k, e_k)]$ contains events paired with timestamps that precisely denote timings. Additionally, we only consider the case $I \neq \emptyset$.

We represent an event as a pair $\langle ec, r \rangle \in$ Event of an event request and a return value. [3] For example, $\langle \texttt{read\_int(4)}, 10 \rangle$ may denote an event where a program requests a 4-byte integer input and receives 10 as the return value. When events are generated from a sub-component of the model, some of them are processed by other components, while others are considered *visible*, and included in the distributed trace $\vec{tr}$ of the transition.

[3] In §2, we treated event requests and returns as separate events, since from the programs' perspective they occur separately in terms of timings.

Undefined Behavior vs. No-Behavior. Some of the visible events are designated to indicate special situations that trigger undefined behavior (UB) and no-behavior (NB) in our model by the *event classification function* $C : \text{Event} \to \{\texttt{UB}, \texttt{NB}, \texttt{Obs}\}$ parametrized in the model, where `Obs` corresponds to ordinary events that form observable behaviors.

First, undefined behavior is a widely used concept that captures unexpected results that may arise from abnormal operations, *e.g.,* accessing an unallocated memory region. Therefore, it is often formally interpreted as a set of all behaviors, so that any observable behavior refines undefined behavior. It is the programmer's responsibility to avoid generating undefined behaviors in their programs. In our model, we trigger undefined behavior when the system is stuck (*i.e.,* cannot take the next step) or explicitly generates UB events.

In contrast, as a dual concept of undefined behavior, no-behavior captures improbable executions, *i.e.,* executions that violate axiomatized assumptions. NB can be interpreted as an empty set of behaviors, and thus it refines arbitrary observable behaviors. Programmers can rely on the absence of no-behavior events in reasoning about their programs, *e.g.,* considering highly-reliable time constraints as true, since no-behavior events would not happen as long as the constraints hold.

3.2 Timed Observable Behaviors and Refinement

For a system state $\sigma \in \Sigma$, we define the set of timed observable behaviors $\text{Beh}(\sigma)$ by collecting observable traces formed by infinite transition sequences starting from σ. Instead of revealing its coinductive definition, we state a property equivalent to the definition:

$$\overrightarrow{beh} \in \text{Beh}(\sigma) \quad \Leftrightarrow \quad \text{err}(\sigma) \ \vee \ (\exists \sigma', \overrightarrow{beh'}, \vec{tr}.\ \sigma \xrightarrow{\vec{tr}} \sigma' \wedge \text{AllObs}_C(\vec{tr}) \wedge \text{concats}(\vec{tr}, \overrightarrow{beh'}) = \overrightarrow{beh} \wedge \overrightarrow{beh'} \in \text{Beh}(\sigma')) \tag{1}$$

in which err is a predicate for erroneous states (*i.e.,* a state generating `UB` or a "stuck" state lacking any possible next step, which has the same effect as `UB`) whose behaviors are unpredictable, AllObs_C is a predicate that holds if every event is classified as `Obs` by C, and concats is a pointwise concatenation of traces and behaviors. Intuitively, σ generates a behavior $\overrightarrow{beh}$ if (i) σ is unsafe or (ii) it can make a valid transition whose trace can form $\overrightarrow{beh}$ with one behavior of the subsequent state. From this, we define the system's behaviors $\text{Beh}((\Sigma, \to, I))$ as $\bigcup_{\sigma \in I} \text{Beh}(\sigma)$.

Then, we can define the refinement relation of two systems as the subset relation regarding the timed observable behaviors:

Definition 3.1 (Timed Behavioral Refinement). For two distributed systems $sys_{\text{conc}}, sys_{\text{abs}}$ and under an event classifcation function C, sys_{conc} refines sys_{abs} if $\text{Beh}(sys_{\text{conc}}) \subseteq \text{Beh}(sys_{\text{abs}})$.

Embedding Relation of Event Classification Functions. The event classification C can have multiple instantiations, each imposing different assumptions on the system. For example, our network model generates an event `late_delivery`(m) when it fails to deliver a message m within the parameter value μ. Users may instantiate C to classify the late delivery events as either `Obs` or `NB`. In the former, late deliveries are permitted as valid behaviors (still identifiable by the events), where the latter disallows late deliveries.

To utilize such usages regarding event classification, we define an *embedding* relation within them. Intuitively, the one that classifies more events as `Obs` permits more behaviors than the one that classifies those events as `NB`.

Definition 3.2 (Embedding of Event Classifications). For two event classification functions C_1 and C_2, $C_1 \hookleftarrow C_2$ if $\forall e, C_1(e) = C_2(e) \vee (C_1(e) = \texttt{Obs} \wedge C_2(e) = \texttt{NB})$.

Consequently, with a single refinement proof, users can obtain multiple verification results under different assumptions simply by substituting C. Intuitively, if a system refines a specification under a "permissive" assumption (*e.g.,* allowing late deliveries) then the system still refines the specification under a stricter assumption (*e.g.,* disallowing late deliveries). We show in §8 how to utilize this feature in practice. We present a theorem below that clarifies the implication of the embedding in refinement proofs.

Theorem 3.3 (Refinement Preserved by Embedding). *For any* $C_1 \hookrightarrow C_2$ *and* $sys_{\text{conc}}, sys_{\text{abs}}$, $\text{Beh}(sys_{\text{conc}}) \subseteq \text{Beh}(sys_{\text{abs}})$ under C_1 *implies* $\text{Beh}(sys_{\text{conc}}) \subseteq \text{Beh}(sys_{\text{abs}})$ under C_2.

4 Concrete Formal Distributed System Model

In this section, we explain how we construct a "concrete" distributed system model that consists of the OS model, the user program for each node, and the network model. Here, we focus on how we handle the asynchrony in timings between nodes. The full details are available in [Kim et al. 2024].

4.1 OS Model

The OS model aims to capture the nondeterminism arising from the behavior of local clocks and asynchronous message communication. It takes as input a set of user-defined parameters that reflect the system's assumptions about local clock skew. The model then defines the semantics for a suite of system calls related to clock access and network communication, which a real-time operating system is expected to support.

Local Clock Parameters. In the local clock parameters, users define the type of abstract clock states and specify functions and predicates describing local clock behavior in four cases:

- $\text{LCInit}(t, lc)$: A predicate specifying an initial clock state lc when the global time is t.
- $\text{LCVal}(lc)$: A function retrieving the local time value from the current state.
- $\text{LCAdv}(t, lc, lc')$: A predicate for a clock state transition from t to $t+1$
- $\text{LCSet}(t, lc, lt)$: A function updating the clock state at t when the user program requests to set the local clock value to lt.

We present two instantiations of the parameters, which are used in the case studies in §7 and §8:

(1) Hardware clock rate assumption: This assumes a constraint on the physical clock hardware, limiting the ratio of global time to local time difference between two points (t, lt) and (t', lt') by $\rho \in [0, 1)$, such that: $1 - \rho < (lt' - lt)/(t' - t) < 1 + \rho$.
We define corresponding local clock parameters as follows:
   - Clock state: $lc = (\hat{t}, \hat{lt}, p)$ contains the global and local times at which the local clock is last set, and the "perceived" amount of time $p \in \mathbb{R}$ since then.
   - $\text{LCInit}(t, lc) \stackrel{\text{def}}{=} \exists lt.\ lc = (t, lt, 0)$
   - $\text{LCVal}(lc) \stackrel{\text{def}}{=} \text{let } lc = (\_, \hat{lt}, p) \text{ in } \hat{lt} + \lfloor p \rfloor$
   - $\text{LCAdv}(t, lc, lc') \stackrel{\text{def}}{=} \exists \hat{t}, \hat{lt}, p, p_{\text{inc}}.\ 1 - \rho < p_{\text{inc}} < 1 + \rho \wedge lc = (\hat{t}, \hat{lt}, p) \wedge lc' = (\hat{t}, \hat{lt}, p + p_{\text{inc}})$
   - $\text{LCSet}(t, lc, \hat{lt}) \stackrel{\text{def}}{=} (t, \hat{lt}, 0)$

(2) Clock synchronization assumption: This assumption relies on the presence of a synchronization algorithm running in the background. It limits the absolute difference between global and local times by $\varepsilon > 0$ at every point: $|t - lt| < \varepsilon$.
The local clock parameters are defined as:
   - Clock state: $lc = lt$, representing the current local time.
   - $\text{LCInit}(t, lt) \stackrel{\text{def}}{=} |t - lt| < \varepsilon$
   - $\text{LCVal}(lt) \stackrel{\text{def}}{=} lt$

$$\mathrm{OSInit}(t, (lc, skts, sts)) \stackrel{\text{def}}{=} \mathrm{LCInit}(t, lc) \wedge skts = [] \wedge sts = \texttt{Idle}$$

$$\mathrm{OSLCAdv}(t, (lc, skts, sts), (lc', skts', sts')) \stackrel{\text{def}}{=} \mathrm{LCAdv}(t, lc, lc') \wedge skts = skts' \wedge sts = sts'$$

$$\mathrm{OSAccept}((lc, skts, sts), ms_{\text{in}}) \stackrel{\text{def}}{=} (lc, \mathrm{accept\_skts}(skts, ms_{\text{in}}), sts)$$

$$\mathrm{OSCall}((lc, skts, sts), f(v_1, .., v_n)) \stackrel{\text{def}}{=} \textbf{match } sts \textbf{ with } \texttt{Idle} \Rightarrow \texttt{Some}(lc, skts, \texttt{Proc}(f(v_1, .., v_n))) \mid \_ \Rightarrow \texttt{None } \textbf{end}$$

$$\mathrm{OSRet}((lc, skts, sts)) \stackrel{\text{def}}{=} \textbf{match } sts \textbf{ with } \texttt{Ret}(f(v_1, .., v_n), r) \Rightarrow \texttt{Some}((lc, skts, \texttt{Idle}), f(v_1, .., v_n), r) \mid \_ \Rightarrow \texttt{None } \textbf{end}$$

$$\frac{sc = \texttt{sendto}(sid, ip, port, m)) \quad l = \mathrm{size}(m) \quad skts[sid] \neq \bot \quad ms_{\text{out}} = [(ip, port, m)]}{t \vdash (lc, skts, \texttt{Proc}(sc)) \xrightarrow{ms_{\text{out}}} (lc, skts, \texttt{Ret}(sc, l))}$$

$$\frac{sc = \texttt{recvfrom}(sid, sz) \quad \mathrm{fetch}(skts[sid]) = \texttt{Some}(skt', m) \quad skts' = skts[sid \mapsto skt'] \quad m' = \mathrm{prefix}(m, sz)}{t \vdash (lc, skts, \texttt{Proc}(sc)) \xrightarrow{[]} (lc, skts', \texttt{Ret}(sc, m'))}$$

$$\frac{sc = \texttt{get\_time}() \quad lt = \mathrm{LCVal}(lc)}{t \vdash (lc, skts, \texttt{Proc}(sc)) \xrightarrow{[]} (lc, skts, \texttt{Ret}(sc, lt))}$$

$$\frac{sc = \texttt{set\_time}(t_{l,\text{new}}) \quad lc' = \mathrm{LCSet}(t, lc, t_{l,\text{new}})}{t \vdash (lc, skts, \texttt{Proc}(sc)) \xrightarrow{[]} (lc', skts, \texttt{Ret}(sc, 0))}$$

$$\frac{sc = \texttt{sleep}(lt_w) \quad lt_s = \mathrm{LCVal}(lc)}{t \vdash (lc, skts, \texttt{Proc}(sc)) \xrightarrow{[]} (lc, skts, \texttt{Wait}(lt_s, lt_w))} \qquad \frac{lt = \mathrm{LCVal}(lc) \quad lt < lt_s + lt_w}{t \vdash (lc, skts, \texttt{Wait}(lt_s, lt_w)) \xrightarrow{[]} (lc, skts, \texttt{Wait}(lt_s, lt_w))}$$

$$\frac{sc = \texttt{sleep}(lt_w) \quad lt = \mathrm{LCVal}(lc) \quad lt_s + lt_w \leq lt}{t \vdash (lc, skts, \texttt{Wait}(lt_s, lt_w)) \xrightarrow{[]} (lc, skts, \texttt{Ret}(sc, 0))}$$

Fig. 5. Selected OS-model transition rules

- $\mathrm{LCAdv}(t, lt, lt') \stackrel{\text{def}}{=} lt \leq lt' \wedge (|t - lt| < \varepsilon \implies |(t + 1) - lt'| < \varepsilon)$
- $\mathrm{LCSet}(t, lt, lt_{\text{new}}) \stackrel{\text{def}}{=} lt_{\text{new}}$

Note that the second intantiation maintains the skew between the local and global clocks within the given bound, provided the system does not set the clock. When the system does set the clock, we consider two scenarios: (i) if the new time still falls within the clock skew bound, the model continues to behave normally, and (ii) if the new time exceeds the clock skew bound, the local clock will be updated to an arbitrary value in a nondeterministic manner, which prevents any proof from proceeding successfully. In essence, when setting the clock, we must prove that the new time does not violate the clock skew bound. This approach ensures that our proofs remain valid only when clock adjustments stay within the specified boundaries.

Building OS Model with System Calls. We now define the OS model based on the local clock parameters as follows. An abstract OS state $os = (lc, skts, sts)$ consists of a clock state, a set of open sockets, and a status indicating the ongoing process. Sockets can be bound to a port number, and store inbound messages in their buffers. A status is either:

- `Idle`: No ongoing process
- `Proc`(sc): Processing a system call sc with arguments (*e.g.,* $sc = \texttt{sleep}(10)$)
- `Wait`(lt_s, lt_w): OS started waiting for lt_w amount of time since lt_s (in terms of local time)
- `Ret`(sc, v): Returing from sc with value v

Fig. 5 presents the functions and predicates that define the OS model's event handling. These events include initialization, local clock advancement, network message reception, and the invocation, processing, and return of system calls. The OSInit predicate defines valid initial states of the OS, while OSLCAdv advances the local clock in accordance with LCAdv. When messages are received from the network, the OSAccept function distributes them to the appropriate open sockets. System call invocations are handled by OSCall, which changes the status from `Idle` to `Proc`. Once processing is complete, OSRet returns control and the resulting value to the program. For system call processing, the rules follow the form $t \vdash os \xrightarrow{ms_{\text{out}}} os'$ describing how the processing changes the state of the OS with any messages transmitted to the network (ms_{out}) at one time unit. Throughout the paper, we use the shorthand notation $t \vdash s \xrightarrow{e} s'$ for $(t, s) \xrightarrow{e} (t + 1, s')$. The bottom of Fig. 5 presents specific rules for selected system calls.

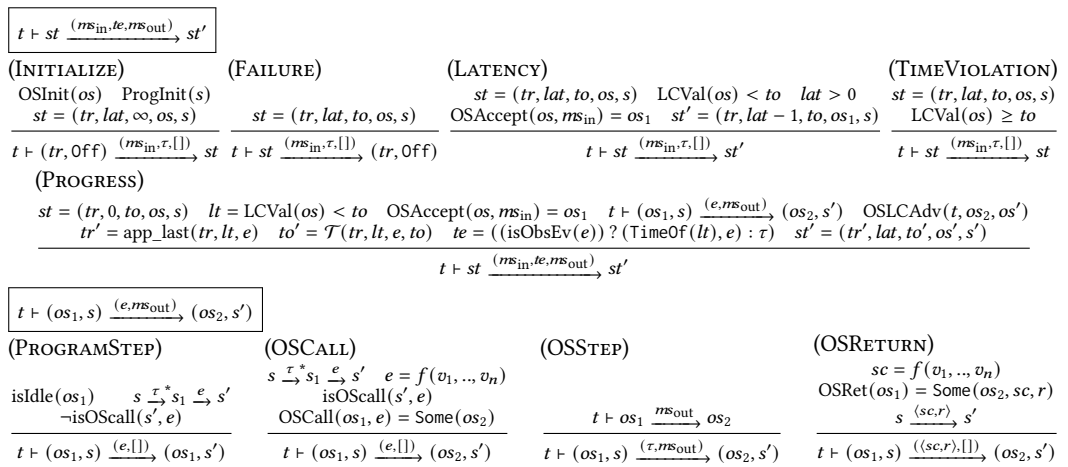


Fig. 6. Node transition rules

4.2 Concrete Node Model

Now, we construct a concrete node model by integrating the OS model with the untimed semantics of a program. Building upon the timed operational semantics construction method described in §2, we address additional complexities, including node failures, local clocks, message communication through the network, and control transitions between the program and OS. Fig. 6 presents the transition rules for this model, which we will explain in detail.

To model node failures, we define a node state as either inactive $(tr, \texttt{Off})$ or active (tr, lat, to, os, s), with the active state following our construction method. An inactive node may reboot at any time, as described by the Initialize rule. We assume a predicate ProgInit specifies the program's initial states. Conversely, an active node may unexpectedly fail, as described by Failure.

The next three rules largely follow our construction method, with adaptations for local clock and message communication. In these rules, we compare the timeout to against LCVal(os), implicitly coercing os to its clock state. For message communication, non-violation transitions (Latency and Progress) first process incoming messages ms_{in} via OSAccept. Latency simply decreases lat thereafter. Progress involves a nested transition for OS and program state, followed by local clock advancement via OSLCAdv. To determine the timed observable event te, we exclude non-observable events like τ and system call events, which are considered internal program-OS communication. The framework takes as input a timestamp generator `TimeOf`, supporting timestamp abstraction. This feature is used in §8 for matching timestamps between asynchronous and synchronous executions. System call events are still recorded in tr, as $\mathcal{T}$ uses them to update to.

The bottom four rules in Fig. 6 present the transitions of program and OS states, and how control passes between them. ProgramStep and OSStep represent normal transitions of the program and OS components, respectively. Program steps can occur when the OS is idle, while OS steps may involve releasing outgoing messages, as we have seen in §4.1. We define a predicate isOScall to determine if a program state has just called a specific system call. OSCall and OSReturn represent the control transfers between the program and OS, potentially updating timeout values.

4.3 Network Model

The top-level structure of the concrete distributed system model is composed of a list of node models and our network model. A node model is a tuple $(ip, ST, \rightarrow, st_{\text{init}})$ of the node's distinct IP address ip, set of states ST, transition rules $\rightarrow$, and initial state st_{init}. Our concrete node model is an instance of node model, with $st_{\text{init}} = \{([], \texttt{Off})\}$. The network model is a transition system with

two separate transition stages for distributing ($\longrightarrow_{\text{distr}}$) and gathering ($\longrightarrow_{\text{gather}}$) messages generated from the nodes. The network model can represent various types of network behaviors by setting two parameters: the maximum delivery time μ and the maximum duplicated deliveries κ for a message. If the network takes a transition step violating the parameters, it generates a special event that can later be classified as no-behavior by the event classification C. We present the construction of a distributed system model $(\Sigma, \longrightarrow, I)$ in §3 as follows.

A global state $(t, nw, \vec{st})$ consists of a global time, a network state, and a list of node states. First, a network state is a pair $nw = (\vec{mc}, \vec{ms})$ of a multicast group table and a list of in-transit messages. Each entry $(ip_{\text{mc}}, ip, t_{\text{send}}, b) \in \vec{mc}$ contains the IP address of a multicast group (*e.g.,* one of Class D addresses [Cotton and Vegoda 2010]), the local IP address of a node belonging to the group, the time the "join" request is released to the network, and the boolean flag indicating whether the join process is completed (which also takes at most μ since it is requested). Each entry $(ip_{\text{dst}}, m, t_{\text{send}}, d) \in \vec{ms}$ contains a message, the time it was released to the network, and the number of deliveries (>1 for duplicates) until now. Third, each entry $\vec{st}[i]$ represents the current state of the i'th node. The initial global state is given by $(0, ([], []), [st_{\text{init},0}, \ldots, st_{\text{init},N-1}])$, for N nodes.

The transition rule of a global state is given as follows:

$$\frac{t \vdash nw \xrightarrow{(\vec{ms}_{\text{in}}, te_{\text{nw}})}_{\text{distr}} nw_1 \quad \forall i < N.\ t \vdash \vec{st}[i] \xrightarrow{(\vec{ms}_{\text{in}}[i], \vec{te}[i], \vec{ms}_{\text{out}}[i])} \vec{st}'[i] \quad t \vdash nw_1 \xrightarrow{\vec{ms}_{\text{out}}}_{\text{gather}} nw'}{(t, nw, \vec{st}) \xrightarrow{te_{\text{nw}} :: \vec{te}} (t+1, nw', \vec{st}')}$$

Note that a transition occurs at every tick of the time unit in three stages:

- **Distributing Messages:** The distributing network transition $\longrightarrow_{\text{distr}}$ non-deterministically selects the messages $\vec{ms}_{\text{in}}$ to be distributed in this step and updates the multicast group table, resulting in the intermediate network state nw_1. Here, $\vec{ms}_{\text{in}}$ is an indexed list according to the IP addresses of the nodes. te_{nw} may contain three kinds of network events related to μ and κ: (i) `LateDeliv`(ip_{dst}, m) if μ has just expired now but m is not delivered yet, (ii) `DupOverLimit`(ip_{dst}, m) if m has already been delivered at least κ times but is included again in $\vec{ms}_{\text{in}}$, and (iii) `LateArrv`(ip_{dst}, m) if μ has already expired but m is included in $\vec{ms}_{\text{in}}$.
- **Taking a Step:** With the incoming messages, each node takes a step in parallel. Each node state $\vec{st}[i]$ at time t takes the incoming messages $\vec{ms}_{\text{in}}[i]$, generates timed events $\vec{te}[i]$ and a (possibly empty) outgoing message $\vec{ms}_{\text{out}}[i]$, and transitions to the next state $\vec{st}'[i]$ at time $t+1$.
- **Gathering Messages:** After every node takes a step, the network gathers the outgoing messages $\vec{ms}_{\text{out}}$ to form the next network state nw'. Specifically, this step adds entries of $\vec{mc}$ or $\vec{ms}$ for each well-formed outgoing message according to the message type: either a normal data message or a multicast join request. For example, if a message contains normal data sent to a multicast group, the message is copied for each group member according to $\vec{mc}$ and then included in $\vec{ms}$.

5 Proof Techniques

VeriRT offers three simulation relations as proof techniques to assist the user in proving the refinement. Each technique is designed for different stages in the end-to-end refinement proof outlined in Fig. 4. From the bottom of the figure, we will discuss how each proof technique addresses the challenges at each stage. For the presentation purpose, we omit stuttering indexes [Leroy 2009b] from our simulation relations in this section, which is indeed necessary for their soundness.

Note that, the three simulation relations we present here, each matching *states* of two semantics at each level, can be easily lifted to relations matching two *semantics* themselves. Hence, when referring to simulation, we interchangeably use the term for both between states and semantics.

5.1 Program-Local Simulation

The program-local simulation relation is used for local abstractions on the program components in the concrete distributed system model, which occurs in the first two stages of the end-to-end refinement proof. In the first stage, the framework internally applies this technique to construct the formal compiler correctness proof with respect to the global-level timed behaviors (see §6). In the second stage, the user is expected to verify their C programs against their abstract specifications, which will ensure the absence of implementation bugs and also facilitate subsequent refinement proofs by preemptively abstracting away the subtleties of formal C semantics.

For this purpose, we design our program-local simulation as a conventional, untimed simulation relation. This is made possible by the fact that our composition of timed operational semantics from the computational behaviors and timing assumptions is essentially orthogonal. Once the user establishes a program-local simulation proof, the framework will internally *lift* it to a node-local simulation (§5.2) by combining it with the given timing assumptions.

The definition of the simulation relation between two "source" and "target" program semantics $(S_s, \rightarrow_s, I_s, F_s)$ and $(S_t, \rightarrow_t, I_t, F_t)$ is like the following:

$$\begin{aligned} s_s \gtrsim s_t \overset{\text{coind}}{=} & (i)\ (s_s \in F_s \wedge s_t \in F_t) \ \vee \ (ii)\ \mathrm{err}(s_s) \ \vee \\ & (iii)\ \neg\mathrm{err}(s_t) \wedge (\forall e, s_t'.\ s_t \xrightarrow{e} s_t' \implies \exists s_s'.\ s_s \xrightarrow{e}{}^{+} s_s' \wedge s_s' \gtrsim s_t') \end{aligned} \tag{2}$$

The relation coinductively matches two program states s_s and s_t as follows. First, two states are matched if they are (i) final states, (ii) if s_s may trigger undefined behavior (denoted as $\mathtt{err}(s_s)$), or (iii) s_t is a safe (*i.e.,* non UB-generating) state where every step from it can be matched with one or more steps of s_s.

5.2 Node-Local Simulation

After the abstraction of programs, the user may use our node-local simulation relation for further local abstractions on the OS-Node models. Specifically, the user is supposed to show that the compositions of the timing assumptionss and the computational specifications of programs together refine more abstract node-level specifications (*e.g.,* Fig. 1).

Now, we need to build a *timed simulation*, *i.e.,* one can simulate another in terms of not only the order of events, but also the timings, too. In particular, the incoming and outgoing messages should occur at exactly identical times, since the incoming messages are controlled by the environment (*i.e.,* the network model) and differences in timings of outgoing messages may alter the behaviors of other nodes.

Therefore, the user is required to resolve node-local timing issues throughout the proof. Especially, the OS-Node model may produce an infinite number of non-deterministic timed behaviors due to the random latencies of internal program steps and skews of the local clock. The main task of the proof is thus to make sure that the non-determinism in timings is well-constrained by no-behavior events generated according to the given time constraints.

To assist the proof from the framework's side, we design the node-local simulation as follows, where the key differences with conventional simulations are highlighted in red:

$$\begin{aligned} t \vdash st_s \gtrapprox st_t \overset{\text{coind}}{=} & \mathrm{err}(st_s) \ \vee \\ & \neg\mathrm{err}(st_t) \wedge (\forall ms_{\text{in}}, te, ms_{\text{out}}, st_t'.\ (t \vdash st_t \xrightarrow{(ms_{\text{in}}, te, ms_{\text{out}})} st_t') \implies \\ & (i)\ \exists t_N, e_N.\ (t_N, e_N) \in te \wedge C(e_N) = \mathtt{NB} \vee \\ & (ii)\ \exists st_s'.\ (t \vdash st_s \xrightarrow{(ms_{\text{in}}, te, ms_{\text{out}})} st_s') \wedge ((t+1) \vdash st_s' \gtrapprox st_t')) \end{aligned} \tag{3}$$

The differences are introduced for the following reasons. First, the relation takes the global time t since the transition rules of a node-level operational semantics may depend on t. Second, for a target step generating NB events, the simulation does not require matched source steps, since the target step will not produce any observable behaviors. Third, for each target step it requires exactly one matched source step to enforce the events and message exchanges to occur at the identical time. Recall that a single OS-Node step takes one time tick, during which an arbitrary number of program steps may take place to account for the non-determinism of timings.

We present our simulation-lifting theorem that converts a program-local simulation into a node-local simulation between two OS-Node states here (see [Kim et al. 2024] for the proof sketch):

Theorem 5.1 (Simulation-Lifting Theorem). *For two programs* p_s, p_t,
$\forall s_s \in p_s.S, s_t \in p_t.S. \quad s_s \gtrsim s_t \implies \forall t, tr, lat, to, os.\ t \vdash (tr, lat, to, os, s_s) \gtrapprox (tr, lat, to, os, s_t)$.

5.3 Global Simulation

Finally, the framework provides the global simulation relation for abstractions of the entire system. With this simulation, the user can complete the end-to-end refinement from the distributed system model with assembly programs to a monolithic, centralized abstract specification of the system. The user would aim to abstract away subtleties caused by distributed computation, *e.g.,* randomness in network message delivery times and asynchronous paces of execution between the nodes.

Eliminating the asynchronies may require the reallocation of events over multiple steps. For example, consider the top-level global specification as an ideally synchronized model in which every node periodically runs in a lock-step manner. In this case, events for a certain period will be generated at a designated time, simultaneously. However, the corresponding concrete distributed system model will generate events with certain intervals between them, asynchronously between the nodes, reflecting the real-world scenario. Therefore, a proof technique that supports event reallocation over multiple steps would be helpful in this stage.

For this purpose, we designed our global simulation as a *delayed simulation* below, which allows one to postpone the decision of matched source steps during the proof by accumulating the trace generated by multiple target steps:

$$
\begin{aligned}
\sigma_s \gtrapprox_{tr_{acc}} \sigma_t \overset{\text{coind}}{=} & \ \mathrm{err}(\sigma_s) \ \vee \\
& \neg\mathrm{err}(\sigma_t) \wedge (\forall \vec{te}_t, \sigma_t'.\ (\sigma_t \xrightarrow{\vec{te}_t} \sigma_t') \Rightarrow \\
& (i)\ \exists t_N, e_N.\ (t_N, e_N) \in \vec{te}_t \wedge C(e_N) = \mathsf{NB} \ \vee \ (ii)\ \sigma_s \gtrapprox_{(tr_{acc} +\!\!+ \vec{te}_t)} \sigma_t' \ \vee \\
& (iii)\ \exists \vec{te}_s, \sigma_s'.\ \sigma_s \xrightarrow{\vec{te}_s}{}^{+} \sigma_s' \wedge \vec{te}_s \equiv (tr_{acc} +\!\!+ \vec{te}_t) \wedge (\sigma_s' \gtrapprox_{[]} \sigma_t')
\end{aligned}
\tag{4}
$$

Now, the relation is indexed with a trace tr_{acc}. The main difference is that when the target state σ_t takes a step, the user may choose to accumulate the trace (ii) to delay the matching process. After that, when the target system reaches a proper state (iii), the user gets to pick proper source steps that generate an equivalent trace to resolve the accumulated trace and can proceed to the next relation with the empty trace index. The equivalence relation $\equiv$ decides whether the two traces have timed events with identical order and timestamps but without requiring them to be generated in identical time ticks.

6 Lifting CompCert's Proof

The bottom-most stage in the end-to-end refinement is a composition of two sub-stages of proof included in CompCertRT. First, we generically convert the CompCertM [Song et al. 2019]'s "mixed-simulation" to our timed program-local simulation, from which we derive the timed refinement

using Theorem 5.1. Up to this point, the assembly modules are logically linked by CompCertM's module semantics. Then, by porting CompCertM's proof of the adequacy of logical linking of assemblies against syntactic linking, we complete the end-to-end timed refinement proof with respect to the syntactic linking of compiled assemblies.

6.1 Lifting of Mixed Simulation

First, We derive our program simulation from the mixed simulation of CompCertM. Let $\mathcal{R}$ be the set of mixed simulations—a few instantiations of a single generic open simulation—used by CompCertM that covers every compiler pass of CompCert. See [Kim et al. 2024] for the properties of mixed simulation.

Lemma 6.1 (Mixed Simulation to Program Simulation). *For any pair of modules (M_S, M_T) related by a mixed simulation relation $R \in \mathcal{R}$, and any two list of modules $\vec{M}_L$ and $\vec{M}_R$ whose element is self-related by R (i.e., $\forall M \in \vec{M}_L \cup \vec{M}_R.(M, M) \in R$), the following program-local simulation holds: $(\vec{M}_L +\!\!+ [M_S] +\!\!+ \vec{M}_R) \gtrsim (\vec{M}_L +\!\!+ [M_T] +\!\!+ \vec{M}_R)$.*

By applying Lemma 6.1 and Theorem 5.1 repeatedly, we prove Lemma 6.2 that guarantees the correctness of the CompCertM's translation C w.r.t. timed refinement.

Lemma 6.2 (Correctness of CompCertM's translation). *For any list of Clight modules $[S_1, ..., S_n]$ and Asm modules $[T_1, ..., T_n]$, if $C(S_i) = T_i$ for each $i \in [1, n]$, then for any identical ip, the concrete node model with the program $[T_1, ..., T_n]$ contextually refines the concrete node model with the program $[S_1, ..., S_n]$.*

6.2 Syntactic Linking of Assembly Programs

While the result of Lemma 6.2 lowers each Clight module into the compiled Asm modules, the modules are logically linked by the interaction semantics. We first prove Lemma 6.3 that implies that the syntactic linking of the Asm modules preserves the semantics, and combine it with Lemma 6.2 to prove the final compiler correctness theorem in terms of the concrete system model.

Lemma 6.3 (Syntactic Linking of Asms). *For any list of Asm modules $[T_1, ..., T_n]$, if the syntactic linking succeeds and produces an Asm module T such that $T_1 \circ ... \circ T_n = T$, then $[T_1, \ldots, T_n] \gtrsim [T]$.*

Theorem 6.4 (Lifting CompCert's Proof). *For a C program $C^i = [S^i_1, ..., S^i_n]$ consisting of multiple C files and the assembly program $A^i = [T^i]$ obtained by syntactically linking the separately compiled assemblies for each i, the concrete distributed system model with $[A^1, \ldots, A^n]$ refines the concrete distributed system model with $[C^1, \ldots, C^n]$.*

7 Case Study 1: Clock Synchronization

Cristian's algorithm is a protocol for synchronizing local clocks in distributed systems using a client-server architecture [Cristian 1989]. The algorithm's simple structure involves one round-trip message exchange between a client and server. The process begins with the client requesting the server's local clock value. Upon receiving the response, the client adjusts its local clock based on the assumption that the travel times for the request and response messages are likely to be similar. In this case study, we develop and verify a client-server system that periodically synchronizes local clocks using Cristian's algorithm.

7.1 Analysis on Clock Synchronization

We conduct a formal analysis of the bounds of clock skews based on these assumptions:

- **Hardware clock skew:** The local clocks follow the hardware clock rate assumption (§4.1).
- **System call response delay:** When the program is waiting for a certain condition via a system call (*e.g.,* a response from the timer or the network), the execution will resume within a constant time δ in terms of the local clock, after the condition is satisfied.

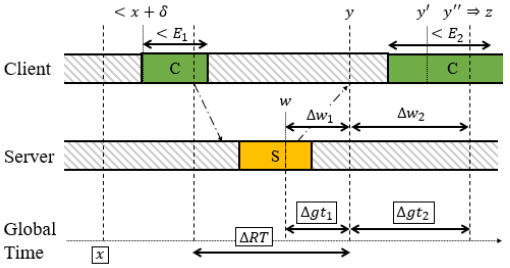


Fig. 7. Analysis of Cristian's algorithm

- **WCET (Bounded execution time for sending a request):** The local time between a timer return and waiting for the server's response is bounded by E_1 (see Fig. 7).
- **WCET (Bounded execution time for processing the response):** After the client receives the response, the local time until the client's local clock is updated is bounded by E_2.

Now, we walk through one successful period of synchronization illustrated in Fig. 7. Consider a period that is supposed to start at x. When the period starts, the timer wakes up the client before its local clock points $x + \delta$ and the round trip of messages begins within E_1. Within the round trip that takes ΔRT, the server receives the request and sends the response message that contains a local time w sampled in between. Let Δt_1 denote the global time taken from the sampling of w to the end of the round trip, during which the local clocks of the client and server change to y and $w + \Delta w_1$. Then, within δ, the client will resume, so reads its local time y' and then sets the local clock from a value y'' to $z = w + (y' - x)/2$ within E_2 after the resume. Let Δt_2 denote the global time taken from the end of the round trip up to this point, during which the local clocks of the server change to $v = w + \Delta w_1 + \Delta w_2$.

We derive the following bound, which depends on ΔRT (see [Kim et al. 2024] for details):

$$-(\frac{1+3\rho}{2})\Delta RT - \frac{1+\rho}{1-\rho}(\delta + E_2) \leq z - v \leq \frac{1+\rho}{2}\Delta RT + \frac{1}{2}(\delta + E_1)$$

Building on this result, we can bound the skew by imposing additional assumptions on the network and the server-side execution time. Let us assume that the message delivery time is bounded by μ, and the server-side operation takes at most $\delta + E_3$. Then, we obtain $\Delta RT \leq 2\mu + E_3/(1-\rho)$, resulting in the following skew bound ε:

$$|z-v| < \varepsilon = \max\left((\frac{1+3\rho}{2})(2\mu + E_3/(1-\rho)) + \frac{1+\rho}{1-\rho}(\delta + E_2), \frac{1+\rho}{2}(2\mu + E_3/(1-\rho)) + \frac{1}{2}(\delta + E_1)\right)$$

Our System Design. In practical implementations of the above algorithm, server-side queues of pending requests can complicate verification. These queues may form due to various factors: overly frequent or duplicate client requests, delayed message deliveries, competition among multiple clients, and other issues. Since an unpredictable number of pending requests can prevent accurate estimation of an upper bound for ΔRT, we need to establish conditions to control these factors to eliminate pending requests.

To address the issue, we configure our network model as $\kappa = 1$, setting $C(\texttt{LateDeliv}(\_, \_)) = \texttt{Obs}$ while classifying other network events as `NB`. This configuration eliminates message duplication and late arrivals, while permitting message drops. Such a setup can be realized through a background procedure that runs a duplicate resolution algorithm and discards excessively delayed messages by examining their timestamps.

For simplicity, we narrow our focus to a single-client scenario. We program the client to maintain a minimum interval E_{wait} between successive requests. Specifically, this interval must satisfy the condition: $\mu + (\delta + E_3)/(1-\rho) < E_{\text{wait}}/(1+\rho)$. Here, $(\delta + E_3)/(1-\rho)$ represents the maximum global time for the server to process a request, while $E_{\text{wait}}/(1+\rho)$ is the minimum global time for the client to wait E_{wait} with respect to its local clock.

7.2 Verification

To verify the skew bound presented in §7.1 for our implementation, we design a specification and proof structure based on two abstract node models. We define two such models, one for the server and one for the client, each with its own local clock. For each abstract node, we establish a node-local simulation with its corresponding concrete node, which also ensures exact matching of local clocks. We then prove that the distributed system with these abstract nodes satisfies a global invariant I_{glob}, from which we derive the skew bound. By combining these results with the soundness of node-local simulation provided by VeriRT, we prove our goal.

Handling Distributed Concurrency. For a system state $(t, nw, [st_{\text{sv}}, st_{\text{cl}}])$ where st_{sv} and st_{cl} represent the abstract server and client node states respectively, we define local invariants for each node to facilitate local reasoning. Our proof proceeds in three steps: we derive the local invariants from the global invariant, verify their preservation during transition steps, and then combine these results to establish the global invariant for the subsequent state. We focus here on the server-side local invariant; the client-side follows a similar approach, which is detailed in [Kim et al. 2024].

The server-side local invariant consists of two parts: $I_{\text{sv}}(t, nw, st_{\text{sv}}) = I^{\text{st}}_{\text{sv}}(t, nw, st_{\text{sv}}) \wedge I^{\text{rsp}}_{\text{sv}}(t, nw, st_{\text{sv}})$. At a high level, $I^{\text{st}}_{\text{sv}}$ implies:

(1) If st_{sv} is idle, nw contains at most one in-transit request from the client.
(2) If st_{sv} is processing a request $req = \langle id_{\text{req}} \rangle$, then nw contains no in-transit requests. For such req, let t_{s}^{req} and t_{r}^{req} be the client's sending time and server's receiving time, respectively. Then $t_{\text{r}}^{req} - t_{\text{s}}^{req} < \mu$ and $t - t_{\text{r}}^{req} < (\delta + E_3)/(1-\rho)$.

$I^{\text{rsp}}_{\text{sv}}$ implies that for any in-transit response $rsp = \langle id_{\text{req}}, t_{\text{sv}} \rangle$ in nw, there exists a corresponding request $req = \langle id_{\text{req}} \rangle$ in nw satisfying the following: let t_{s}^{req} and t_{r}^{req} be req's sending and receiving times, and t_{s}^{rsp} be rsp's sending time, then $t_{\text{r}}^{req} - t_{\text{s}}^{req} < \mu$, $t_{\text{s}}^{rsp} - t_{\text{r}}^{req} < (\delta + E_3)/(1-\rho)$, and $t - t_{\text{s}}^{rsp} < \mu$. Additionally, t_{sv} must be a valid local time sampled by the server between t_{r}^{req} and t_{s}^{rsp}.

For message generation, we assume a property of client-generated messages that must be proven on the client side: $G_{\text{cl}}(t, nw, ms_{\text{out,cl}}) = \forall req' \in ms_{\text{out,cl}}, req \in nw.\ req' \neq req \wedge t_{\text{s}}^{req} + E_{\text{wait}}/(1+\rho) < t$. This ensures the client adheres to the minimum interval E_{wait}. Conversely, the server must guarantee a property for the client side: $G_{\text{sv}}(t, nw, ms_{\text{out,sv}}) = \forall rsp = \langle id_{\text{req}}, t_{\text{sv}} \rangle \in ms_{\text{out,sv}}.\ \langle id_{\text{req}} \rangle \in nw$.

Lemma 7.1, which states the preservation of I_{sv}, is presented below along with a high-level proof:

Lemma 7.1 (Server-side local invariant). *For a global state $(t, nw, [st_{\text{sv}}, st_{\text{cl}}])$, a distribution of messages $t \vdash nw \xrightarrow{([ms_{\text{in,sv}}, ms_{\text{in,cl}}], te_{\text{nw}})}_{\text{distr}} nw_1$ with no* NB *events in te_{nw} and a node-local transition $t \vdash st_{\text{sv}} \xrightarrow{(ms_{\text{in,sv}}, te_{sv}, ms_{\text{out,sv}})} st'_{\text{sv}}$, if $I_{\text{sv}}(t, nw, st_{\text{sv}})$ and $G_{\text{cl}}(t, nw, ms_{\text{out,cl}})$ hold, then $G_{\text{sv}}(t, nw, ms_{\text{out,sv}})$ holds, and for the network gathering step $t \vdash nw_1 \xrightarrow{[ms_{\text{out,sv}}, ms_{\text{out,cl}}]}_{\text{gather}} nw'$, $I_{\text{sv}}(t+1, nw', st'_{\text{sv}})$ holds.*

Proof. We perform a case analysis on st_{sv} as follows:

- If st_{sv} is idle: nw' will have at most one request. If there is a new request in $ms_{\text{out,cl}}$, G_{cl} ensures $t_{\text{s}}^{req} + \mu < t$ for all existing req, implying req is not in-transit.
  - If no request arrives: st'_{sv} remains idle, implying $I^{\text{st}}_{\text{sv}}$. $I^{\text{rsp}}_{\text{sv}}$ holds as there is no new response in nw' and remaining in-transit responses satisfy the conditions.

  - If a new request req arrives: The server processes req, establishing $I^{\mathrm{st}}_{\mathrm{sv}}$ from the network assumptions. $I^{\mathrm{rsp}}_{\mathrm{sv}}$ holds for the same reason as above.
- If st_{sv} is processing a request req: G_{cl} and $I^{\mathrm{st}}_{\mathrm{sv}}$ together imply $ms_{\mathrm{out,cl}} = []$.
  - If st'_{sv} still processes req: $I^{\mathrm{st}}_{\mathrm{sv}}$ holds, otherwise $t - t_{\mathrm{r}}^{req} \geq (\delta + E_3)/(1-\rho)$ contradicts the timing assumptions.
    * If $ms_{\mathrm{out,sv}}$ is empty: $I^{\mathrm{rsp}}_{\mathrm{sv}}$ holds as in previous cases.
    * If $ms_{\mathrm{out,sv}}$ contains rsp: rsp is a new response for req. $I^{\mathrm{rsp}}_{\mathrm{sv}}$ holds for the new response. G_{sv} is ensured straightforwardly with req.
  - If st'_{sv} completes processing req: $I^{\mathrm{st}}_{\mathrm{sv}}$ holds as st'_{sv} is idle with no in-transit request. $I^{\mathrm{rsp}}_{\mathrm{sv}}$ holds under the same case analysis on $ms_{\mathrm{out,sv}}$.

□

Finally, we define the global invariant as $I_{\mathrm{glob}}(t, nw, [st_{\mathrm{sv}}, st_{\mathrm{cl}}]) = I_{\mathrm{sv}}(t, nw, st_{\mathrm{sv}}) \wedge I_{\mathrm{cl}}(t, nw, st_{\mathrm{cl}})$. From the lemmas on local invariants, we derive Theorem 7.2, which establishes the preservation of the global invariant.

Theorem 7.2 (Global invariant). *For a global state* $(t, nw, [st_{\mathrm{sv}}, st_{\mathrm{cl}}])$ *that satisfies the invariant* $I_{\mathrm{glob}}(t, nw, [st_{\mathrm{sv}}, st_{\mathrm{cl}}])$*, any transition step* $(t, nw, [st_{\mathrm{sv}}, st_{\mathrm{cl}}]) \xrightarrow{\vec{r}} (t', nw', [st'_{\mathrm{sv}}, st'_{\mathrm{cl}}])$ *without generating* NB *satisfies* $I_{\mathrm{glob}}(t', nw', [st'_{\mathrm{sv}}, st'_{\mathrm{cl}}])$.

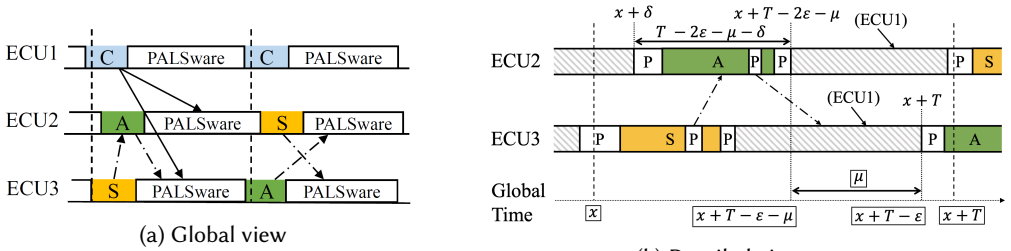
8 Case Study 2: PALSware

PALSware [Al-Nayeem et al. 2013] is a middleware implementation of physically asynchronous logically synchronous (PALS) architectural design [Al-Nayeem et al. 2009; Sha et al. 2009] for real-time distributed systems. It aims to provide a logically synchronous view for the application layer on top of an asynchronous underlying environment based on a set of assumptions. An application of PALSware consists of multiple modules, called *tasks*, that run on each physical node. Each physical node runs a PALSware instance, which periodically launches its task's job.

8.1 Correctness of Synchronization

PALSware achieves correct synchronization based on the following requirements on the environment [Sha et al. 2009]. As we target safety-critical systems, we achieve strong and precise guarantees on top of strong assumptions obtained from additional hardware/software support, *e.g.,* duplicating networks, while most other distributed system verification work aims to guarantee the preservation of certain safety invariants under realistic network failures [Hawblitzel et al. 2015; Honoré et al. 2022; Wilcox et al. 2015].

- **Clock synchronization assumption:** The local clocks follow the clock synchronization assumption (§4.1). Operating a clock synchronization protocol [PTP 2020; Cristian 1989; Mills 1992] on a real-time OS and reliable network will realize this condition.
- **Reliable network:** The network delivers every packet within the maximum network delay μ, possibly with reorderings but without any duplication or loss. This assumption has been studied [Abbasloo and Chao 2020; Chou et al. 1990] and realized with a high probability [Inc. 2005] for safety-critical systems.
- **Timer response delay:** `OS_wait_timer` will wake up within the maximum wakeup delay δ in the local clock after the timer expires. Conceptually, it could be subsumed by the WCET assumption below, but we separate it for a precise OS model.



(a) Global view

(b) Detailed view

Fig. 8. Active-standby system running on PALSware

- **WCET (Worst-Case Execution Time):** For each period, the program execution time (*i.e.,* the time taken from waking up to the next sleep) does not exceed $T - 2\varepsilon - \mu - \delta$ in the local clock for the length of period T.[4]

Fig. 8 illustrates how PALSware correctly operates the active-standby system under these assumptions. Fig. 8a shows an execution of the actual system in which the controllers are toggled, and Fig. 8b zooms in on the first period. Initially, PALSware on ECU2 is sleeping after setting the timer to x. Then, it wakes up before $x + \delta$ due to the third assumption and soon launches an active job. The job sends the heartbeat message to ECU3 before completion, at which the global clock (boxed in the figure) is at most $x + T - \varepsilon - \mu$. Therefore, the message is delivered no later than $x + T - \varepsilon$ in the global clock, which is the earliest time for any node to start the next period, guaranteeing that the next job on ECU3 will receive the heartbeat.

Our Implementation of PALSware. In this work, we simplified three functional features of the original work of PALSware [Al-Nayeem et al. 2013] for verification. First, we support a single period length, while the original work supports the multi-rate and multi-phase extensions. Second, we allow at most one message for each sender-receiver pair of tasks in a period.[5] Third, we do not implement the fault management logic with user-defined exception handlers for violations of the assumptions. Our middleware works as PSync [Drăgoi et al. 2016] when violations occur, though our formal result does not include it.

Our Application on PALSware. In this case study, we consider an active-standby system simplifying a flight control system [Al-Nayeem et al. 2013, 2009; Sha et al. 2009] for an application of PALSware to verify. The system is composed of three distributed tasks: two replicated controllers for fault tolerance and one console for the user interface to switch the active side. As a practical application of the active-standby system, we implement and verify a resource scheduling system whose scheduler is replicated as two active-standby controllers. The controllers manage a mutex lock for a single resource, which is occasionally requested by multiple device tasks. The controllers maintain a circular queue of requests, which is inserted in the Heartbeat for backup. The controllers and devices are designed to achieve safety and liveness of the system under unexpected failures of any devices and at most one controller.

8.2 Verification

Our end-to-end timed behavioral refinement proof is a vertical composition of four layers, where each *incrementally* abstracts away concrete details of the bottom-level system model. Table 1

[4]The original work of PALSware uses the global clock for the WCET assumption. We use the local clock to simplify the integration of the assumption in our formal model. Indeed, both result in identical constraints, as shown in this section.
[5]We can simulate multiple messages by exploiting synchrony; by accumulating messages locally and sending one aggregate message at the end. Considering the headers of UDP and PALSware, we allow a maximum of 65498 bytes for a message.

summarizes the four layers. Each row in the table shows the name of the layer, lower- and upper-level models, simulation technique used (which we discuss in §5), assumptions needed for the simulation proof, key properties verified for the proof of the layer, and the advantages of the abstraction that helps the verification of higher layers.

Name	Abstraction	Simulation	Required Assumptions	Proven Properties	Advantages
Ref. 4	Asynch to Synch	Global		Job movable within a period	Elimination of nondeterminism in time
Ref. 3	Network + AbsPALS to Asynch	Global	Reliable network assumption, Correct system config	No interference by messages, All deliveries done in time	Abstract and deterministic network
Ref. 2	OS + PALS-Spec to AbsPALS	Node	WCET assumption, Timer assumption, Clock skew assumption	Periodic execution	No node-local clocks, Simplified interaction between App & network
Ref. 1	C to ITree	Prog	User's proof obligation	Absence of impl. bugs	Abstract data & execution

Table 1. Summary of the incremental abstraction

Building on this result, we formally verify the active-standby system, by establishing program simulation between the C implementation and specification of each task, and then linking this proof with the middleware verification result.

Abstract Global Models. Here, we explain the two global system models: the asynchronous and synchronous systems. The asynchronous model abstracts away the details of the OS and network models. The synchronous model serves as the top-level specification of the system, eliminating asynchrony caused by concurrent executions. The refinement between these two models (Ref. 4) addresses the asynchrony resulting from concurrent execution (which will be discussed later).

The two system models are parametrized by a PALSware application specification, which includes the period length T, the number of nodes N, and tasks specifications for each node represented as untimed operational semantics. The transition steps of the i'th task are described in the form $inb \vdash s_i \xrightarrow{e} s_i'$, where inb is the inbox storing incoming messages from the previous period, and e is either an observable event or a message transmission to another task.

the asynchronous system model is constructed from a given application as follows. A system state $\sigma_\mathrm{a} = (t_\mathrm{a}, [(s_{\mathrm{a},0}, inb_{\mathrm{c},0}, inb_{\mathrm{n},0}), \cdots, (s_{\mathrm{a},N-1}, inb_{\mathrm{c},N-1}, inb_{\mathrm{n},N-1})])$ consists of the global time and N asynchronous node states, where each contains a task state , an inbox for the current period, and another inbox for the next period. This model simulates the exact timings of the concrete model; while $kT - \varepsilon < t_\mathrm{a} < (k+1)T - \varepsilon - \mu$ for some k, every step of the system model executes each task state. Specifically, a task state may take a step $(s_{\mathrm{a},i}, inb_{\mathrm{c},i}, inb_{\mathrm{n},i}) \xrightarrow{e} (s'_{\mathrm{a},i}, inb_{\mathrm{c},i}, inb'_{\mathrm{n},i})$ if $inb_{\mathrm{c},i} \vdash s_{\mathrm{a},i} \xrightarrow{e^*} s'_{\mathrm{a},i}$, and $inb'_{\mathrm{n},i}$ is the updated inbox by accepting new incoming messages from other tasks. After t_a reaches $(k+1)T - \varepsilon - \mu$, the node states stop changing until $t_\mathrm{a} = (k+1)T - \varepsilon$, at which each node transitions to $(s_{\mathrm{a},i}, inb_{\mathrm{c},i}, inb_{\mathrm{n},i}) \xrightarrow{\tau} (s_{\mathrm{a},i}, inb_{\mathrm{n},i}, [])$ for the next period $(k+1)T$.

The synchronous system model, on the other hand, executes each task synchronously for each period. In a system state $\sigma_\mathrm{s} = (t_\mathrm{s}, [(s_{\mathrm{s},0}, inb_{\mathrm{s},0}), \cdots, (s_{\mathrm{s},N-1}, inb_{\mathrm{s},N-1})]$, each node state contains a task state and the inbox for the next period. While $t_\mathrm{s} \neq kT$ for any k, the state does not change. When $t_\mathrm{s} = kT$ for some k, each task takes a big-step transition, $inb_{\mathrm{s},i} \vdash s_{\mathrm{s},i} \Downarrow_{tr_i} s'_{\mathrm{s},i} \stackrel{\mathrm{def}}{=} inb_{\mathrm{s},i} \vdash s_{\mathrm{s},i} \xrightarrow{tr_i{}^*} s'_{\mathrm{s},i}$. Then, all outgoing messages from all tasks are gathered to create the new inboxes for each node.

Handling Distributed Concurrency. Here, we present our approach to handling concurrency and eliminating asynchrony during the refinement proof between the two models described above. This is achieved by desiginig global and local invariants that relate two system states from each side and proving their preservation.

We begin by relating two states from each system that are ready to begin a new period. Consider an asynchronous system state ${\sigma_\mathrm{a}}^\dagger$ at $t_\mathrm{a} = kT - \varepsilon + 1$, where each node is initialized to $(s^\dagger_{\mathrm{a},i}, inb^\dagger_{\mathrm{c},i}, [])$

to start the new period kT. This state can be related to a synchronous system state σ_s at $t_s = kT$ whose i'th node state is $(s_{s,i}, inb_{s,i}) = (s^{\dagger}_{a,i}, inb^{\dagger}_{c,i})$.

Generalizing this relation, we design a global invariant I_{glob} that matches the state σ_s and an asynchronous state σ_a after taking several steps from ${\sigma_a}^{\dagger}$ until $t_a \in (kT - \varepsilon, (k+1)T - \varepsilon]$, as illustrated in Fig. 9. Let $\vec{tr}$ be the trace generated from those steps and $(s_{a,i}, inb_{c,i}, inb_{n,i})$ be the i'th node states of σ_a. Then, for each node i, $inb_{c,i} \vdash s^{\dagger}_{a,i} \xrightarrow{\vec{tr}[i]}{}^{*} s_{a,i}$ holds where $inb_{c,i} = inb^{\dagger}_{c,i} = inb_{s,i}$ and $s^{\dagger}_{a,i} = s_{s,i}$. Also, $inb_{n,i}$ contains the inbound messages generated in $\vec{tr}$, which we may denote as $inb_{n,i} = \text{gather\_msgs}(\vec{tr}, i)$. Thus, the following invariant I_{glob} holds:

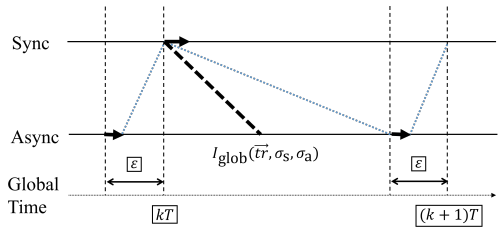


Fig. 9. Global invariant

$$I_{loc}(\vec{tr}, i, (s_{s,i}, inb_{s,i}), (s_{a,i}, inb_{c,i}, inb_{n,i})) \stackrel{\text{def}}{=} inb_{c,i} \vdash s_{s,i} \xrightarrow{\vec{tr}[i]}{}^{*} s_{a,i} \wedge inb_{s,i} = inb_{c,i} \wedge inb_{n,i} = \text{gather\_msgs}(\vec{tr}, i)$$

$$I_{glob}(\vec{tr}, (t_s, \vec{st}_s), (t_a, \vec{st}_a)) \stackrel{\text{def}}{=} (\exists k.\ t_s = kT \wedge kT - \varepsilon < t_a \leq (k+1)T - \varepsilon) \wedge \bigwedge_{i<N} I_{loc}(\vec{tr}, i, \vec{st}_s[i], \vec{st}_a[i])$$

The next step is to show that I_{glob} implies global simulation, *i.e.*, $\forall \vec{tr}, \sigma_s, \sigma_a. I_{glob}(\vec{tr}, \sigma_s, \sigma_a) \implies \sigma_s \gtrapprox_{\vec{tr}} \sigma_a$, using a coinductive reasoning. A high-level proof is as follows:

Proof. For $\vec{tr}, \sigma_s, \sigma_a$ that satisfy $I_{glob}(\vec{tr}, \sigma_s, \sigma_a)$, we need to show $\sigma_s \gtrapprox_{\vec{tr}} \sigma_a$. From I_{glob}, we can denote $\sigma_s = (kT, \vec{st}_s)$ and $\sigma_a = (t_a, \vec{st}_a)$ where $kT - \varepsilon < t_a \leq (k+1)T - \varepsilon$. We consider two cases:

(1) When $t_a < (k+1)T - \varepsilon$: For any step $\sigma_a \xrightarrow{\vec{te}_a} \sigma'_a$, $I_{glob}(\vec{tr} +\!\!+ \vec{te}_a, \sigma_s, \sigma'_a)$ holds. Therefore, we can apply the coinductive hypothesis to prove part (ii) of Equation (4).

(2) When $t_a = (k+1)T - \varepsilon$: The next step, say $\sigma_a \xrightarrow{\vec{t}} {\sigma_a}^{\dagger\prime}$, initializes the asynchronous state for the next period. We now consider part (iii) of Equation (4). From I_{glob}, we can construct the first step $\sigma_s \xrightarrow{\vec{tr}} {\sigma_s}^{\dagger}$ where all tasks run their jobs synchronously. Following this, there exists an execution ${\sigma_s}^{\dagger} \xrightarrow{\tau}{}^{*} \sigma'_s$ until $t'_s = (k+1)T$, during which no node states change. Finally, we need to show $\sigma'_s \gtrapprox_{[]} {\sigma_a}^{\dagger\prime}$. We can apply the coinductive hypothesis with $I_{glob}([], \sigma'_s, {\sigma_a}^{\dagger\prime})$ which trivially holds.

□

Finally, we verify that the initial states satisfy $I_{glob}([], \sigma_{s,init}, \sigma_{a,init})$, which implies the global simulation between the two systems.

Applying Different Event Classifications. By instantiating the event classification C with different functions, we can impose different assumptions on the system. The top-level global specification is an ideally synchronized abstract system that may randomly drop an incoming message m of a task, with generating a specific kind of network events $\texttt{LateDeliv}(ip_{dst}, m)$. In the refinement process, we treat $\texttt{LateDeliv}(ip_{dst}, m)$ as an observable event, so we can instantiate $C = C_1$ such that $C_1(\texttt{LateDeliv}(ip_{dst}, m)) = \texttt{Obs}$ for all m, implying that the PALSware implementation works as the model described in PSync [Drăgoi et al. 2016] under the network without the network delay bound assumption. Then, we can construct $C_2 \hookrightarrow C_1$ (Definition 3.2) such that $C_2(\texttt{LateDeliv}(ip_{dst}, m)) = \texttt{NB}$ and $C_2(e) = C_1(e)$ for other events, and Theorem 3.3 implies that the PALSware implementation works as the deterministic synchronous model without message drops under the reliable network assumption of the original PALSware paper.

9 Evaluation

We summarize the line count information of our development in Table 2. It includes the framework and two case studies: the clock synchronization and and PALSware. For Coq files, we used the `coqwc` tool, which separately counts the lines for specification and for proof. For C files, we used the `cloc` tool. Both tools exclude empty lines and comments.

Components	Def.	Proof
VeriRT		
Library	2099	2106
Models	2966	2464
ProofTech	1866	3610
CompCertRT	3644	6804
ClockSync		
Impl (C)	313	
ImplVerif	1050	3526
Spec	1763	3004
Refinement	2274	6597
PALSware		
Impl (C)	327	
ImplVerif	2239	5913
Spec	1627	1303
Refinement	2503	8861
Active-Standby		
Impl (C)	352	
ImplVerif	1775	4718

Table 2. Development

VeriRT is composed of four main parts: general Coq libraries, formal models and their properties, proof techniques with soundness proofs, and CompCertRT. Library contains definitions and utility lemmas for standard Coq definitions (*e.g.,* option, list, and numbers), and declares standard axioms used in our development. Models corresponds to the development in §3 and §4, and ProofTech is related to §5. CompCertRT corresponds to §6, with a significant portion copied and adapted from CompCertM for proving §6.2. It also includes utility lemmas and tactics for verifying concrete C programs.

The results of the case studies are structured as follows. The ImplVerif rows represent the program-local abstractions of C programs, which is not extensively engineered in this work. We believe we could significantly reduce the proof efforts by employing state-of-the-art simulation techniques such as CCR [Song et al. 2023]. This method offers the full power of separation logic based on resource algebras, combined with inductive-coinductive simulation techniques. The Spec rows include abstract specifications and their verified properties. For ClockSync, concurrent reasoning was performed at this level (see §7.2). The Refinement rows include the application of our proof techniques for abstraction, performing combined reasoning about computational and timing behaviors. Subtle behaviors in our concrete OS and network models are eliminated at this level. In PALSware, we handled concurrency during the refinement proof (see §8.2).

10 Related Work

Our work is built on the intersection of real-time system verification and distributed system verification, and we compare ours with significant related work from each field.

Verification on Real-Time Systems. There have been various verification tools and methodologies focusing on abstract designs of real-time systems. Uppaal [Bengtsson et al. 1996] is a model-checking tool for system designs expressed as timed automata. The time Petri net [Merlin and Farber 1976] is proposed as a model of distributed systems with execution time constraints, which is extended in various ways in verifying real-time systems [Bucci et al. 2003; Ding et al. 2013]. The Real-Time Maude Tool [Ölveczky and Meseguer 2007] supports formal specification and verification of real-time system models expressed as rewriting rules, on which various system designs are verified [Meseguer and Ölveczky 2012; Ölveczky and Caccamo 2006; Ölveczky and Thorvaldsen 2009], including the PALS architecture. While the abovementioned approaches have successfully verified real-time system *designs*, they do not extend toward verifying code-level implementations, which we aim for.

Verification Framework for Distributed Systems. Verdi [Wilcox et al. 2015; Woos et al. 2016] is a framework designed for the development of verified distributed systems across multiple network

semantics. In terms of modeling, our network model can express Verdi's models by manipulating the network parameters. However, we take different verification approaches to dealing with multiple network semantics. Verdi offers automatic transformations of systems by adding "handler" code for abnormal network behaviors. Our framework does not provide such transformations, although such things can potentially be verified in VeriRT, using our event classification facility.

Ironfleet [Hawblitzel et al. 2015] is another framework built on Dafny [Leino 2010], which offers a centralized top-level model similar to ours. They support TLA-style refinement, which supports liveness proofs. In addition, by leveraging Dafny, they support semi-automated proofs using SMT solvers. While they offer a practical formal verification tool for distributed systems, their underlying formal foundation is not as rigorous as ours, especially when it comes to the infrastructure models (*e.g.,* network and local clocks) and handling timed behaviors.

A notable advantage of our framework over the aforementioned ones is our support for verifying C-level systems. The formal result ensures correctness down to the assembly level, whereas the others support systems with representations at a higher level.

There are some studies focusing on the composition of system components, such as Disel [Sergey et al. 2017]. They support modular abstraction (program-to-protocol) and linking multiple protocols, which improves the reusability of formal verification results of distributed system components. In terms of compositionality, we inherit the advantages of CompCertM, which offers high flexibility in program composition. In the PALSware system, we leverage this feature to independently verify the middleware and application components and then integrate the results afterwards. Also, we support the vertical composition of proofs as we presented in this paper. However, horizontal compositions of multiple protocols are not addressed in this work.

Some frameworks focus on specific types of distributed systems. Chapar [Lesani et al. 2016] is a framework for verifying causally consistent key-value stores and their clients. The studies on atomic distributed objects [Honoré et al. 2021, 2022] aim to verify various consensus protocols with a unified abstract model. These studies propose abstract models for specific domains of systems, while we focus on general systems involving real-time aspects.

Synchronization in Distributed System Implementation and Verification. PSync [Drăgoi et al. 2016] suggests a programming model for distributed systems with a fault model that describes the system as synchronous in which only message drops occur, presenting observational equivalence to the asynchronous system. While the original concept of PALSware aims to support safety-critical systems under the assumption of a highly reliable infrastructure, we have implemented our version of PALSware to function as PSync with network failures, and prove that this implementation indeed adheres to the original specification of PALSware when the network is reliable.

Transforming an asynchronous system to a synchronous system using the reduction technique [Kragl et al. 2020; Lipton 1975; v. Gleissenthall et al. 2019], *i.e.,* reordering operations without altering observable behaviors to construct a synchronous equivalent, significantly reduces verification effort, making model checking a tractable solution. Our framework shares the same spirit: to abstract unnecessary race conditions away to improve the efficiency of verification. The main difference is that those studies rely on logical synchronization points, while our work involves reasoning on timed behaviors and time constraints.

Formal Timing Analysis on OS and Hardware. Several works have focused on formal timing analysis at both OS and hardware levels. For schedulability and response time analysis on the OS level, works such as [Cerqueira et al. 2016; Maida et al. 2022] provide formal frameworks, though their WCET definitions differ from ours; they consider the execution time of tasks in isolation, while our concept captures the actual time consumed between system calls in a task while other tasks are present. For hardware-level timing verification, researchers have developed cycle-accurate formal

models [Hu and Chang 2001; Schwarz et al. 2017] that could potentially be integrated with our framework. Such integration would enable end-to-end formal timing verification from high-level system properties down to hardware behavior.

11 Conclusion

In this paper, we present VeriRT, a formal verification framework with a novel approach to handling timing assumptions in formal refinement proofs. This approach offers a robust concept of timing assumptions that are independent of execution code. We achieve this by using the No Behavior concept, which reduces all timing assumptions to WCET conditions between system calls for the target assembly. This approach allows us to easily extend CompCert's existing untimed semantics to timed semantics by defining a simple wrapper. Consequently, we can lift the existing correctness proof to a timed semantics setting by simply proving a meta-level lemma.

Nevertheless, our current framework has a limitation: it does not provide a formal verification method for the generated WCET conditions on the target assembly. The current validation of these conditions relies on comprehensive whole-system testing with reasonable deadline margins.

We envision integrating formal schedulability and response time analysis to strengthen WCET condition validation. Our framework provides a solid foundation for future research in time-aware formal verification, particularly in safety-critical systems where timing guarantees are essential.

Acknowledgments

This work was primarily supported by Samsung Research Funding Center of Samsung Electronics under Project Number SRFC-IT2102-03, and partially supported by the National Science Foundation under Grant No. 2019285. Chung-Kil Hur is the corresponding author. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

2020. IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. *IEEE Std 1588-2019 (Revision of IEEE Std 1588-2008)* (2020), 1–499. https://doi.org/10.1109/IEEESTD.2020.9120376

Soheil Abbasloo and H Jonathan Chao. 2020. SharpEdge: An asynchronous and core-agnostic solution to guarantee bounded-delays. *CCF Transactions on Networking* 3, 1 (2020), 35–50. https://doi.org/10.1007/s42045-020-00032-z

Abdullah Al-Nayeem, Cheolgi Kim, Woochul Kang, Po-Liang Wu, and Lui Sha. 2013. Middleware design for Physically-Asynchronous Logically-Synchronous (PALS) systems. In *2013 Proceedings of the International Conference on Embedded Software (EMSOFT)*. 1–10. https://doi.org/10.1109/EMSOFT.2013.6658583

Abdullah Al-Nayeem, Mu Sun, Xiaokang Qiu, Lui Sha, Steven P. Miller, and Darren D. Cofer. 2009. A Formal Architecture Pattern for Real-Time Distributed Systems. In *2009 30th IEEE Real-Time Systems Symposium*. 161–170. https://doi.org/10.1109/RTSS.2009.50

Rajeev Alur and David L. Dill. 1994. A theory of timed automata. *Theoretical Computer Science* 126, 2 (1994), 183–235. https://doi.org/10.1016/0304-3975(94)90010-8

Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. 1996. UPPAAL — a tool suite for automatic verification of real-time systems. In *Hybrid Systems III*, Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 232–243.

G. Bucci, A. Fedeli, L. Sassoli, and E. Vicario. 2003. Modeling flexible real time systems with preemptive time Petri nets. In *15th Euromicro Conference on Real-Time Systems, 2003. Proceedings.* 279–286. https://doi.org/10.1109/EMRTS.2003.1212753

Felipe Cerqueira, Felix Stutz, and Björn B. Brandenburg. 2016. PROSA: A Case for Readable Mechanized Schedulability Analysis. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*. 273–284. https://doi.org/10.1109/ECRTS.2016.28

Baiyu Chen, Zhengyu Yang, Siyu Huang, Xianzhi Du, Zhiwei Cui, Janki Bhimani, Xin Xie, and Ningfang Mi. 2017. Cyber-physical system enabled nearby traffic flow modelling for autonomous vehicles. In *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*. 1–6. https://doi.org/10.1109/PCCC.2017.8280498

C.-T. Chou, I. Cidon, I.S. Gopal, and S. Zaks. 1990. Synchronizing asynchronous bounded delay networks. *IEEE Transactions on Communications* 38, 2 (1990), 144–147. https://doi.org/10.1109/26.47845

Michelle Cotton and Leo Vegoda. 2010. Special Use IPv4 Addresses. RFC 5735. https://doi.org/10.17487/RFC5735

Flaviu Cristian. 1989. Probabilistic clock synchronization. *Distributed computing* 3, 3 (1989), 146–158. https://doi.org/10.1007/BF01784024

Zhijun Ding, Changjun Jiang, and Mengchu Zhou. 2013. Design, Analysis and Verification of Real-Time Systems Based on Time Petri Net Refinement. *ACM Trans. Embed. Comput. Syst.* 12, 1, Article 4 (Jan. 2013), 18 pages. https://doi.org/10.1145/2406336.2406340

Cezara Drăgoi, Thomas A. Henzinger, and Damien Zufferey. 2016. PSync: A Partially Synchronous Language for Fault-Tolerant Distributed Algorithms. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 400–415. https://doi.org/10.1145/2837614.2837650

Hemangi Laxman Gawand, A.K. Bhattacharjee, and Kallol Roy. 2017. Securing a cyber physical system in nuclear power plants using least square approximation and computational geometric approach. *Nuclear Engineering and Technology* 49, 3 (2017), 484–494. https://doi.org/10.1016/j.net.2016.10.009

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 1–17. https://doi.org/10.1145/2815400.2815428

Wolf Honoré, Jieung Kim, Ji-Yong Shin, and Zhong Shao. 2021. Much ADO about failures: a fault-aware model for compositional verification of strongly consistent distributed systems. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 97 (Oct. 2021), 31 pages. https://doi.org/10.1145/3485474

Wolf Honoré, Ji-Yong Shin, Jieung Kim, and Zhong Shao. 2022. Adore: Atomic Distributed Objects with Certified Reconfiguration. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 379–394. https://doi.org/10.1145/3519939.3523444

Alan J. Hu and Felix Sheng-Ho Chang. 2001. Fast Specification of Cycle-Accurate Processor Models . In *Proceedings 2001 International Conference on Computer Design. ICCD 2001*. IEEE Computer Society, Los Alamitos, CA, USA, 0488. https://doi.org/10.1109/ICCD.2001.955072

Aeronautical Radio Inc. 2005. ARINC Specification 664: Aircraft Data Network, Part 7 - Avionics Full Duplex Switched Ethernet (AFDX) Network.

Siddhartha Kumar Khaitan and James D. McCalley. 2015. Design Techniques and Applications of Cyberphysical Systems: A Survey. *IEEE Systems Journal* 9, 2 (2015), 350–365. https://doi.org/10.1109/JSYST.2014.2322503

Yoonseung Kim, Sung-Hwan Lee, Yonghyun Kim, and Chung-Kil Hur. 2024. Artifact for POPL 2025 - VeriRT: An End-To-End Verification Framework for Real-Time Distributed Systems. https://doi.org/10.5281/zenodo.13937956

Bernhard Kragl, Constantin Enea, Thomas A. Henzinger, Suha Orhun Mutluergil, and Shaz Qadeer. 2020. Inductive Sequentialization of Asynchronous Programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 227–242. https://doi.org/10.1145/3385412.3385980

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

Xavier Leroy. 2009a. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. https://doi.org/10.1145/1538788.1538814

Xavier Leroy. 2009b. A formally verified compiler back-end. *Journal of Automated Reasoning* 43, 4 (2009), 363–446. https://doi.org/10.1007/s10817-009-9155-4

Mohsen Lesani, Christian J. Bell, and Adam Chlipala. 2016. Chapar: Certified Causally Consistent Distributed Key-Value Stores. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 357370. https://doi.org/10.1145/2837614.2837622

Thomas Leveque, Etienne Borde, Amine Marref, and Jan Carlson. 2011. Hierarchical Composition of Parametric WCET in a Component Based Approach. In *2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. 261–268. https://doi.org/10.1109/ISORC.2011.38

Richard J. Lipton. 1975. Reduction: A Method of Proving Properties of Parallel Programs. *Commun. ACM* 18, 12 (dec 1975), 717–721. https://doi.org/10.1145/361227.361234

Marco Maida, Sergey Bozhko, and Björn B. Brandenburg. 2022. Foundational Response-Time Analysis as Explainable Evidence of Timeliness. In *34th Euromicro Conference on Real-Time Systems (ECRTS 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 231)*, Martina Maggio (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 19:1–19:25. https://doi.org/10.4230/LIPIcs.ECRTS.2022.19

Aminé Marref. 2010. Compositional timing analysis. In *2010 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*. 144–151. https://doi.org/10.1109/ICSAMOS.2010.5642071

C. Maxim, A. Gogonel, I. Asavoae, M. Asavoae, and L. Cucu-Grosjean. 2017. Reproducibility and Representativity: Mandatory Properties for the Compositionality of Measurement-Based WCET Estimation Approaches. *SIGBED Rev.* 14, 3 (nov 2017), 24–31. https://doi.org/10.1145/3166227.3166230

P. Merlin and D. Farber. 1976. Recoverability of Communication Protocols - Implications of a Theoretical Study. *IEEE Transactions on Communications* 24, 9 (1976), 1036–1043. https://doi.org/10.1109/TCOM.1976.1093424

José Meseguer and Peter Csaba Ölveczky. 2012. Formalization and correctness of the PALS architectural pattern for distributed real-time systems. *Theoretical Computer Science* 451 (2012), 1–37. https://doi.org/10.1016/j.tcs.2012.05.040

D. Mills. 1992. RFC1305: Network Time Protocol (Version 3) Specification, Implementation. https://doi.org/10.17487/RFC1305

Peter Csaba Ölveczky and Marco Caccamo. 2006. Formal simulation and analysis of the CASH scheduling algorithm in real-time maude. In *Proceedings of the 9th International Conference on Fundamental Approaches to Software Engineering* (Vienna, Austria) *(FASE'06)*. Springer-Verlag, Berlin, Heidelberg, 357–372. https://doi.org/10.1007/11693017_26

Peter Csaba Ölveczky and José Meseguer. 2007. Semantics and pragmatics of Real-Time Maude. *Higher Order Symbol. Comput.* 20, 1–2 (June 2007), 161–196. https://doi.org/10.1007/s10990-007-9001-5

Peter Csaba Ölveczky and Stian Thorvaldsen. 2009. Formal modeling, performance estimation, and model checking of wireless sensor network algorithms in Real-Time Maude. *Theor. Comput. Sci.* 410, 2–3 (Feb. 2009), 254–280. https://doi.org/10.1016/j.tcs.2008.09.022

Krishna Sampigethaya and Radha Poovendran. 2012. Cyber-physical system framework for future aircraft and air traffic control. In *2012 IEEE Aerospace Conference*. 1–9. https://doi.org/10.1109/AERO.2012.6187151

Michael Schwarz, Carlos Villarraga, Dominik Stoffel, and Wolfgang Kunz. 2017. Cycle-accurate software modeling for RTL verification of embedded systems. In *2017 IEEE 20th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*. 103–108. https://doi.org/10.1109/DDECS.2017.7934571

Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2017. Programming and proving with distributed protocols. *Proc. ACM Program. Lang.* 2, POPL, Article 28 (Dec. 2017), 30 pages. https://doi.org/10.1145/3158116

Lui Sha, Abdullah Al-Nayeem, Mu Sun, Jose Meseguer, and Peter C Olveczky. 2009. *PALS: Physically asynchronous logically synchronous systems.* Technical Report.

Jianhua Shi, Jiafu Wan, Hehua Yan, and Hui Suo. 2011. A survey of Cyber-Physical Systems. In *2011 International Conference on Wireless Communications and Signal Processing (WCSP)*. 1–6. https://doi.org/10.1109/WCSP.2011.6096958

Youngju Song, Minki Cho, Dongjoo Kim, Yonghyun Kim, Jeehoon Kang, and Chung-Kil Hur. 2019. CompCertM: CompCert with C-assembly linking and lightweight modular verification. *Proc. ACM Program. Lang.* 4, POPL, Article 23 (Dec. 2019), 31 pages. https://doi.org/10.1145/3371091

Youngju Song, Minki Cho, Dongjae Lee, Chung-Kil Hur, Michael Sammler, and Derek Dreyer. 2023. Conditional Contextual Refinement. *Proc. ACM Program. Lang.* 7, POPL, Article 39 (Jan. 2023), 31 pages. https://doi.org/10.1145/3571232

Klaus v. Gleissenthall, Rami Gökhan Kıcı, Alexander Bakst, Deian Stefan, and Ranjit Jhala. 2019. Pretend Synchrony: Synchronous Verification of Asynchronous Distributed Programs. *Proc. ACM Program. Lang.* 3, POPL, Article 59 (jan 2019), 30 pages. https://doi.org/10.1145/3290372

James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. 2015. Verdi: a framework for implementing and formally verifying distributed systems. *SIGPLAN Not.* 50, 6 (June 2015), 357–368. https://doi.org/10.1145/2813885.2737958

Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. 2016. Planning for change in a formal verification of the raft consensus protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs* (St. Petersburg, FL, USA) *(CPP 2016)*. Association for Computing Machinery, New York, NY, USA, 154–165. https://doi.org/10.1145/2854065.2854081

Received 2024-07-11; accepted 2024-11-07