Towards Building Verifiable CPS using Lingua Franca*

Citation for published version:

Digital Object Identifier (DOI):
10.1145/3609134

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
ACM Transactions on Embedded Computing Systems

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.
Towards Building Verifiable CPS using Lingua Franca∗

SHAOKAI LIN, University of California, Berkeley, USA
YATIN A. MANERKAR, University of Michigan, USA
MARTEN LOHOSTROH, University of California, Berkeley, USA
ELIZABETH POLGREEN, University of Edinburgh, Scotland
SHENG-JUNG YU, University of California, Berkeley, USA
CHADLIA JERAD, University of Manouba, Tunisia
EDWARD A. LEE, University of California, Berkeley, USA
SANJIT A. SESHIA, University of California, Berkeley, USA

Formal verification of cyber-physical systems (CPS) is challenging because it has to consider real-time and concurrency aspects that are often absent in ordinary software. Moreover, the software in CPS is often complex and low-level, making it hard to assure that a formal model of the system used for verification is a faithful representation of the actual implementation, which can undermine the value of a verification result. To address this problem, we propose a methodology for building verifiable CPS based on the principle that a formal model of the software can be derived automatically from its implementation. Our approach requires that the system implementation is specified in Lingua Franca (LF), a polyglot coordination language tailored for real-time, concurrent CPS, which we made amenable to the specification of safety properties via annotations in the code. The program structure and the deterministic semantics of LF enable automatic construction of formal axiomatic models directly from LF programs. The generated models are automatically checked using Bounded Model Checking (BMC) by the verification engine Uclid5 using the Z3 SMT solver. The proposed technique enables checking a well-defined fragment of Safety Metric Temporal Logic (Safety MTL) formulas. To ensure the completeness of BMC, we present a method to derive an upper bound on the completeness threshold of an axiomatic model based on the semantics of LF. We implement our approach in the LF Verifier and evaluate it using a benchmark suite with 22 programs sampled from real-life applications and benchmarks for Erlang, Lustre, actor-oriented languages, and RTOSes. The LF Verifier correctly checks 21 out of 22 programs automatically.

CCS Concepts: • Computer systems organization → Embedded software; • Software and its engineering → Formal software verification; Model checking.

Additional Key Words and Phrases: Cyber-Physical Systems, Concurrency, Safety MTL, Axiomatic Modeling, Automated Verification, Model-based Design.

ACM Reference Format:

∗This article appears as part of the ESWEEK-TECS special issue and was presented in the International Conference on Embedded Software (EMSOFT), 2023.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

EMSOFT ’23, September 17–22, 2023, Hamburg, Germany
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06...$15.00
https://doi.org/XXXXXXXX.XXXXXXX
1 INTRODUCTION

Cyber-physical systems (CPS) \cite{47, 49, 66} play a major role in today’s society. From sensor networks to autonomous vehicles, from industrial automation to avionics, these systems have permeated every aspect of our daily lives. CPS applications are often safety-critical; needless to say, it is crucial that such systems are designed and implemented with assurance of correctness.

CPS design and verification can be challenging because one needs to capture the interactions between the digital world and the physical world, and these two worlds have properties that cannot be unified easily \cite{47}. For example, the C language, a mainstream programming language for embedded applications, adopts a sequential programming model, whereas the physical world is intrinsically concurrent. In addition, timing is mostly considered as a side effect of the software implementation and most languages do not include time in their syntax or semantics; however, time is central to the evolution of physical processes and timing should be treated as a correctness criterion rather a performance metric. A CPS-oriented technology stack should address both concurrency and timing, which should also be addressed in CPS formal models.

CPS verification needs more automation. Although advances in model checking and automated theorem proving, such as bounded model checking (BMC) \cite{12, 13} and satisfiability modulo theory (SMT) solvers \cite{7}, have gained popularity due to their capabilities to help automate verification, verification models are often generated by hand. This can be greatly time-consuming for the highly complex systems typical in CPS \cite{47}. It is important to lower the burden of applying formal methods by fully automating verification, which requires not only automated verification algorithms but also generating verification models automatically from systems under verification, with an interface that is easy-to-use and tightly integrated with design and development. In the ideal case, the language used for design should also naturally support the verification process.

One natural formalism for CPS modeling is that of discrete-event systems (DES) \cite{15}. However, formally modeling the behavior of DES using a conventional operational model (i.e., a transition system) is difficult because one needs to specify properties globally over execution traces, a task that cannot be easily achieved at the transition level. In addition, modeling concurrent execution often involves modeling partial order traces, which typically involves the use of nondeterminism. Therefore, when it comes to modeling real-time, concurrent CPS as DES, one needs to look for a modeling strategy more suitable than the operational approach. In this paper, we focus on building axiomatic models.

Our solution. Our solution for building verifiable CPS starts with the adoption of LINGUA FRANCA (LF) \cite{51}, a language suitable for modeling and implementing CPS applications. LINGUA FRANCA is a polyglot coordination language designed to have native support for time and concurrency and adopt a discrete-event semantics. LF endows popular programming languages like C (which we focus on in this paper), C++, and Rust with mechanisms to define and compose reactive and concurrent software components called reactors. While traditional CPS implementations in low-level languages (e.g., C) expose no easy way of extracting the interactions between subsystems, LF programs provide highly analyzable program structures and a well-defined deterministic semantics, exposing the interactions between subsystems clearly and explicitly. These features make it well-suited for formal verification. LF also blurs the boundary between models and implementations by allowing the user to specify both the software and the environment. LF has been used to implement real-life applications, including Autoware \cite{8}, AUTOSAR \cite{56}, distributed databases \cite{48}, and labs in embedded systems courses \cite{38}.

In this paper, we aim to ensure the functional correctness of LF programs and thus define a verification problem of checking Safety Metric Temporal Logic (Safety MTL) formulas \cite{59} for LF programs. To perform verification, we present a methodology to encode the execution of an LF
program as an axiomatic model, instead of an operational model that aims to build an abstraction of the system using explicit notions of states and transitions [1, 31]. Axiomatic models allow us to specify LF behavior based on a well-defined LF semantics. An axiomatic model can thus be sound w.r.t. all valid runtime implementations that correctly implement the same LF semantics. This is a significant advantage over operational models, which need to be carefully designed to be sound abstractions of a subset of the possible implementations. To make BMC complete, we propose an algorithm that calculates an upper bound on the BMC completeness threshold based on an LF program under verification and a Safety MTL property.

We implement our approach in the LF VERIFIER, which can verify an LF program fully automatically. For each LF implementation, the LF VERIFIER generates a verification model for Uclid5 [64, 70], a modeling and verification framework. The Uclid5 model is compiled into an SMT query and then checked automatically by the Z3 solver [18]. The Uclid5 model generation is enabled by the program structure exposed by LF and its deterministic semantics.

Related work. Outside of Lingua Franca, our work is influenced by the existing literature on verifying dataflow languages [22, 27, 32, 33, 73], Petri nets [23, 26], actors [6, 45, 68, 74], and other SMT-based techniques [1, 57]. The Uclid5 verifier we use is an evolution of the older UCLID system [14], one of the first SMT solvers and SMT-based verifiers. In terms of the underlying encoding and the verification strategies, we note that various works [32, 54, 57, 65] implement similar axiomatic models in other domains.

Within the scope of Lingua Franca, we are aware of two works that are directly related to verifying reactor systems at the time of this writing. Sirjani et al. [71] map LF programs onto models written in the Timed Rebeca language [67] and perform model checking using the Rebeca model checker [37], an explicit-state model checker. They model the reactor systems based on the event-based semantics and show that their approach can handle state properties. Deantoni et al. [19] integrate the MoCCML [20] into LF’s Eclipse IDE and leverage the CADP toolbox [29] to check CCSL [3] properties, which can be considered a variant of dataflow properties. We will expand on the difference between state and dataflow properties in Sec. 4.

Our work differs from these prior approaches significantly:

1. LF VERIFIER is the first fully automated verifier for Lingua Franca and generates verification models directly from LF programs. Existing approaches require the user to first manually translate an LF program into an analyzable representation.
2. Our approach supports a well-defined fragment of Safety MTL, a temporal logic well-suited for real-time systems.
3. Our tooling uses an intermediate Uclid5 encoding that is much easier to manipulate and matches the LF semantics precisely.

Contributions. This paper makes the following key contributions:

- We advocate for the use of Lingua Franca for a “design-for-verification” approach to CPS design. We define a verification problem for checking Safety MTL properties of LF programs.
- We present an approach to performing BMC on axiomatic models of LF programs, which models executions of LF programs at the semantics level and hence ensures soundness for all valid LF runtime implementations. To ensure the completeness of BMC, we present an algorithmic method to upper bound the completeness threshold based on an LF program and a Safety MTL property.
- We present the LF VERIFIER, which can automatically generate axiomatic models directly from an LF program. The models are subsequently checked by the Uclid5 verification engine.
with the Z3 solver. The LF Verifier also provides an annotation system fully integrated into the LF development environment.

- We developed a benchmark suite that consists of 22 LF programs sampled from real-life applications and benchmarks for Erlang, Lustre, actor-oriented languages, and RTOSes. And we evaluate the LF Verifier against the benchmark suite. The results show that the LF Verifier correctly checks 21 out of 22 programs automatically.

The remainder of the paper is organized as follows. Sec. 2 presents a running example of a simplified ADAS system encoded in LF. Sec. 3 presents background on Safety Metric Temporal Logic (Safety MTL) and Bounded Model Checking (BMC). In Sec. 4, we present our axiomatic approach for modeling the execution of an LF program and checking Safety MTL formulas using BMC. To ensure the completeness of BMC, Sec. 5 presents a method to compute an upper bound for a completeness threshold. In Sec. 6, we present the implementation details of the LF Verifier and work through verifying a property in the running example. In Sec. 7, we evaluate the LF Verifier against a benchmark suite. We conclude the paper and discuss the future work in Sec. 8.

2 RUNNING EXAMPLE

The reactor model [50, 53] is a deterministic model of concurrent computation for reactive systems. It adopts a discrete-event semantics and extends the actor model [35] with a deterministic order of handling concurrent timestamped messages. Reactors also share similarities with synchronous languages [9], logical execution time (LET) [34], dataflow [21, 58], and process networks [39].

To informally introduce the reactor model, we present a simplified model of an ADAS (Advanced Driver Assistance System) system [75], commonly found in modern vehicles. In this system, a camera and a LiDAR periodically collect data from the environment and send them to an ADAS processor, which performs image recognition on the given data. If an object is getting too close to the vehicle, a warning message is displayed on the dashboard. If the object is within some safety-critical distance, then the ADAS processor automatically applies the brakes. The user can also manually apply the brakes by pressing on the pedal.

We now explain how the diagram in Fig. 1 represents this simplified ADAS system. A reactor is a stateful concurrent component. In the diagram, reactors are represented as rectangles with rounded corners, and they are Camera, LiDAR, Pedal, Brake, Dashboard, ADASProcessor, and ADASModel. A reactor can contain reactions, actions, timers, ports, and connections.

A reaction (e.g., line 6) is a routine that executes when any of its triggers is present, and it is rendered as a chevron in the diagram. The body of a reaction (e.g., lines 7-9) is written in a target language and represents the “business logic” of the application. In this example, the target language is C (line 1), and C code snippets are wrapped between the {= and =} brackets. In addition, the ADASProcessor reactor (lines 35-52) has two reactions, the one on the left labeled as 1 and the one on right as 2. These numbers correspond to the priorities of the reactions within ADASProcessor: when they are triggered simultaneously, reaction 1 will fire before reaction 2. It is important to note that reaction priorities establish a total order only within the same reactor.

A reaction can be triggered by a port (e.g., line 36), a timer (e.g., line 5), or an action (e.g., line 40). A port, rendered as a black triangle in the diagram, is used to communicate with other reactors. A timer, rendered as a clock in the diagram, is used to generate periodic signals. An action is used to schedule future events within a reactor. It is rendered as a white triangle with an “L” if it is a logical action, or a “P” if it is a physical action. A logical action is scheduled by a reaction, such as reaction 2 in ADASProcessor. A physical action is scheduled asynchronously by the environment, such as a press on the brake pedal that triggers an interrupt service routine (ISR). Connections can be drawn between two ports using connection statements such as lines 73-77. Logical connections
are specified using -> with an optional logical delay specified using after (e.g., line 76). Without a delay specified, an event travels from an output port to an input port logically instantaneously.

All events in the system are handled in the order of tags, which use a superdense representation of time \([55]\), encoded as tuples of the form \(g = (t, m)\), where \(t \in \mathbb{T}\) is the time value and \(m \in \mathbb{N}\) is a microstep index. Reactions are logically instantaneous; logical time does not elapse during the execution of a reaction (physical time, denoted as \(T\), on the other hand, does elapse). If a reaction produces an output that triggers another reaction then the two reactions execute logically simultaneously (i.e., at the same tag). Deadlines can be specified and are violated when \(T - t\) is greater than the specified intervals (e.g., lines 33 & 51).

**LF features used in this work.** The formal analyses from the proposed methodology operate on the logical behavior of the program, and the set of language features this work addresses includes all features introduced earlier except physical actions, deadlines, as well as advanced features such as modal models [69]. Lingua Franca is a highly expressive language, and our proposed methodology makes use of most but not all of the language features available. We leave analyzing programs with these language features for future work.
**Safety properties in reactors.** Safety properties are properties that state, in loose terms, “something bad should not happen.” We focus on safety properties because they are generally considered the most important class of properties for safety-critical systems, and they tend to be amenable to automated techniques [33]. In addition, since bounded liveness properties can be converted into safety properties [10], we do not make a distinction between bounded liveness and safety properties.

In reactors, we identify three types of properties: (I) state properties, (II) dataflow properties, and (III) mixed properties. State properties focus on the values of variables. In the context of reactors, they include state variables, ports, and actions. Dataflow properties, on the other hand, specify the temporal behavior of reaction invocations. Mixed properties combine state and dataflow properties.

To illustrate the difference between these types of properties, consider the ADASModel reactor shown in Fig. 1. The property “The ‘frame’ variable in Camera is initially set to an empty value” is a state property (Type I) that only refers to the state variable frame. The property “In ADASProcessor, Reaction 2 is invoked within 100 ms after Reaction 1 is invoked.” is a dataflow property (Type II). Notice that this property only refers to reaction invocations and does not refer to variables. Lastly, a mixed property (Type III) is shown at lines 60-65 in the LF program in Fig. 1. In natural language, the property states “Globally from time 0 to 10 ms, when the LiDAR reaction invokes and the ADASProcessor requests stop at the same tag, the brakes will eventually be applied within 55 ms.” We write the property in MTL

\[
\begin{align*}
&G[0, 10 \text{ ms}]((\text{ADASModel}_l\_reaction_0 \land (F[0]\text{ADASModel}_p\_requestStop == 1))) \\
&\Rightarrow (F[0, 55 \text{ ms}]\text{ADASModel}_b\_brakesApplied == 1))
\end{align*}
\]

where ADASModel_l_reaction_0 refers to the invocation of the reaction inside the LiDAR reactor, ADASModel_p_requestStop is the requestStop variable inside the ADASProcessor reactor, and ADASModel_b_brakesApplied is the brakesApplied variable inside the Brakes reactor. We wrap the property inside an @property annotation, which we will expand on and verify in Sec. 6.

At a high level, the key contribution of this paper is to give a methodology for verification where the only thing a designer needs to do is adding a specification of the safety requirements (as an @property annotation) to the implementation program. This contrasts with most verification methodologies, which require the designer to first construct a model of the implementation. Here, we work directly with the implementation, which is given in LINGUA FRANCA. The model of the implementation is generated automatically from the LINGUA FRANCA program, as is the low-level C code that realizes the application.

3 PRELIMINARIES

**Metric Temporal Logic.** Metric Temporal Logic (MTL) [41], a real-time extension of Linear Temporal Logic (LTL), has been a successful logic for specifying properties of reactive, embedded systems. The syntax and semantics of MTL are given below. We cite the definitions of MTL from Ouaknine and Worrell [59, 60]. The syntax and semantics of MTL are given as follows.

**Definition 3.1 (Syntax of MTL).** Given an alphabet \( \Sigma \) of events, the formulas of MTL are built up from \( \Sigma \) by Boolean connectives and time-constrained versions of the next operator \( X \), the until operator \( U \), and the dual until operator \( eU \) as follows:

\[
\psi ::= \text{true} | \text{false} | a | \psi_1 \land \psi_2 | \psi_1 \lor \psi_2 | X_I \psi | \psi_1 U_I \psi_2 | \psi_1 eU_I \psi_2
\]

where \( a \in \Sigma \), and \( I \subseteq \mathbb{R}_{\geq 0} \) is an open, closed, or half-open interval with end points in \( \mathbb{R}_{\geq 0} \cup \{\infty\} \).

**Definition 3.2 (Semantics of MTL).** Given a (finite or infinite) time word \( \pi = (\sigma, \tau) \) over alphabet \( \Sigma \), a word position \( i < |\pi| \), and an MTL formula \( \psi \), the satisfaction relation \( (\pi, i) \models \psi \) (read \( \pi \)}
satisfies $\psi$ at position $i$ is defined as follows.

\[
\begin{align*}
(p, i) \models a & \iff \sigma_1 = a \\
(p, i) \models \psi_1 \land \psi_2 & \iff (p, i) \models \psi_1 \land (p, i) \models \psi_2 \\
(p, i) \models \psi_1 \lor \psi_2 & \iff (p, i) \models \psi_1 \lor (p, i) \models \psi_2 \\
(p, i) \models X_1\psi & \iff i < |\pi| \land \tau_{i+1} - \tau_i \in I \land (p, i + 1) \models \psi \\
(p, i) \models \psi_1 U_1\psi_2 & \iff \exists j, i \leq j \leq |\pi| \land \tau_j - \tau_i \in I, [(p, j) \models \psi_2 \\
& \land \forall k, i \leq k < j . (p, k) \models \psi_1] \\
(p, i) \models \psi_1 \bar{U}_1\psi_2 & \iff \forall j, i \leq j \leq |\pi| \land \tau_j - \tau_i \in I, [(p, j) \models \psi_2 \\
& \lor \exists k, i \leq k < j . (p, k) \models \psi_1] 
\end{align*}
\]

From here, additional temporal operators can be defined: constrained eventually $F_I$ and constrained globally $G_I$, where $F_I\psi = \text{true} \bar{U}_1\psi$, and $G_I\psi = \text{false} \bar{U}_1\psi$. In this work, we focus on a fragment of MTL called Safety MTL [59, 61, 62], which requires the interval $I$ of the until operator $U_I$ to have finite length. Both the satisfiability problem and the model-checking problem for Safety MTL over infinite timed words are decidable. Within Safety MTL, in this work we additionally constrain the interval $I$ of the next operator $X_I$ and the dual until operator $\bar{U}_I$ to have finite length. We leave the relaxation of this constraint for future work.

**Bounded Model Checking (BMC).** BMC [12] is a well-known technique for bug finding and bounded verification. A model checker implementing BMC usually proceeds as follows: It first checks a property $\psi$ is valid along a path $\pi$ for some $k$ steps (denoted as $\pi \models_k \psi$). If a satisfying assignment is found at this point, the counterexample is returned to the user. If not, the model checker increments $k$ and checks $\pi \models_{k+1} \psi$. The procedure continues until checking $k$ steps becomes intractable, or a completeness threshold is reached. A completeness threshold [42], denoted as $CT$, is a minimum number of $k$ such that $\pi \models_{CT} \psi \implies \pi \models \psi$, i.e. if $\pi$ satisfies $\psi$ up to $CT$ steps, then $\pi$ satisfies the property $\psi$. In Sec. 5, we present an algorithm for upper-bounding $CT$.

### 4 Axiomatic Semantics for LF for Verification

Fig. 2 shows the verification workflow of LF programs and the architecture of the LF Verifier. In this section, we zoom in on the “Uclid5 model” in the figure and present the method of performing BMC using axiomatic models derived from the semantics of Lingua Franca. We begin by describing the typical properties of interest in a system of reactors.

**Axiomatic semantics.** In this section, we give an axiomatic semantics for reactors and use the semantics to construct a model in Uclid5 to solve the verification problem defined below.

**Definition 4.1 (Verification Problem).** Given a reactor, denoted as $r$, and a Safety MTL formula $\psi$, determine whether all execution paths $\pi$ of $r$ satisfy the formula $\psi$, i.e., $\pi \models \psi$.

In the proposed axiomatic semantics, a reaction invocation defines a transition, which matches the event-based semantics in Sirjani et al. [71]. To show the correspondence between the LF semantics in prose and our axiomatic semantics, we will constantly refer to parts of the LF semantics in prose from prior works [50, 52, 53] as we present the axiomatic semantics. The proposed semantics include four segments: (1) logical time and path; (2) reactor semantics; (3) connections and triggers; (4) reaction code. This section only includes the core axioms used to construct the axiomatic model.
Fig. 2. Verification workflow of LF programs and the architecture of the LF Verifier. Blue represents software modules implemented in this work. The other matching colors indicate correspondence between the input program and the generated model.

\[\text{indices} = (1, 2, ..., k)\]  
\[\text{val} \in \text{vals} = V^{\lvert S \rvert}\]  
\[\pi_i = (\text{rxn}, \text{tag}, \text{val}, \text{trig}, \text{sch}, \text{pld})\]  
\[\bar{\pi} = (\pi_1, \pi_2, ..., \pi_k) \in \Pi\]  
\[\text{indices} = \text{indices} \cup (k+1, ..., k+\lvert N \rvert)\]  
\[\pi = \bar{\pi} \cup (\pi_{k+1}, ..., \pi_{k+\lvert N \rvert}) \in \Pi\]  

(0) Notational conventions. We use uppercase letters to denote sets defined in LF (e.g. \(N\) is the set of reactions in an LF program) and italicized words to denote sets in an axiomatic model (e.g. \(\text{rxns}\) is the set of reactions in a corresponding axiomatic model). A lowercase letter is always used to denote an element of a set (e.g. \(n \in N\)). For functions, we use the calligraphic font (e.g. \(G\)) and italicized words (e.g. \(\text{val}_i\)). In addition, we use the blackboard-bold font to denote common sets (e.g. \(\mathbb{N}\) for integers and \(\mathbb{B}\) for \{true, false\}).

(1) Logical time and path. The definitions covered in this segment are shown in Fig. 3. A logical time tag is defined as \(\text{tag} := (t, m)\), which is a tuple of two natural numbers: \(t \in \mathbb{N}\) to represent a timestamp (i.e., the number of nanoseconds elapsed in logical time since the start of the program), and \(m \in \mathbb{N}\) to represent a microstep index, which is used to enforce an ordering on events with the same timestamp. To extract a timestamp or a microstep from a tag, we define projection functions \(T : \text{tags} \to \mathbb{N}\) for mapping a time tag to its time value and \(M : \text{tags} \to \mathbb{N}\) for mapping a time tag to its microstep index. To compare two tags, we define an ordering relation

\[
< = \{(\text{tag}_1, \text{tag}_2) \in \text{tags}^2 | T(\text{tag}_1) < T(\text{tag}_2) \}
\]

\[
\lor (T(\text{tag}_1) = T(\text{tag}_2) \land M(\text{tag}_1) < M(\text{tag}_2))\}
and a relation $\preceq = \{(\text{tag}_1, \text{tag}_2) \in \text{tags}^2 \mid \text{tag}_1 \prec \text{tag}_2 \lor \text{tag}_1 = \text{tag}_2\}$. We use $R$ to denote the set of reacors and $N$ the set of reactions defined in an LF program. In the axiomatic model, a reaction invocation is defined as $\text{rxn} \in \text{rxns} = N \cup \{\epsilon\}$, where $\epsilon$ indicates an empty reaction. Let $S$ denote the union of state variables, ports, and actions defined in the LF program. A vector of values carried by elements in $S$ is defined as $\text{val} \in \text{vals} = V^{[S]}$, where $V$ is an abstract set of values. Let $T$ denote the union of ports and actions defined in the LF program. A boolean vector denoting the presence of triggers is defined as $\text{trig} \in \text{trigs} = \mathbb{B}^{[T]}$. We further let $A \subseteq T$ denote the set of actions specifically. A boolean vector denoting whether actions are scheduled is defined as $\text{sch} \in \text{schs} = \mathbb{B}^{[A]}$. This is different than the same actions in $\text{trigs}$, which are set to true when the scheduled actions actually become present at a future time tag. When actions are scheduled, their payloads are stored in $\text{pld} \in \text{plds} = \mathbb{V}^{[A]}$.

A path $\pi = (\pi_1, \pi_2, ..., \pi_k)$ is a bounded sequence of states, where the length of the sequence is equal to a bound $k$. This is modeled in SMT using the theory of arrays with the constraint that any element outside of the range $[1, k]$ carries an empty value. We also define an indices set as a sequence of natural numbers from 1 to the bound $k$, i.e., $\text{indices} = (1, 2, ..., k)$. Intuitively, a path can be interpreted as an unrolled execution trace over which the axioms will be enforced. To ensure that certain axioms, to be introduced below, also hold at the boundary of the path (i.e., at step $k$), we define an extended version of the path with a padding, where $\text{indices} = \text{indices} \cup (k+1, ..., k+|N|)$, $\pi = \pi \cup (\pi_{k+1}, ..., \pi_{k+|N|}) \in \Pi$, and the length of the padding is set to $|N|$. Note that Safety MTL formulas are evaluated on $\pi$ and not on $\pi$, the padded version, which is used here to prevent the axioms from being trivially unsatisfiable at the path boundary.

A state $\in \text{states}$ is a 6-tuple $(\text{rxn}, \text{tag}, \text{val}, \text{trig}, \text{sch}, \text{pld})$, where $\text{rxn} \in \text{rxns}$ is the reaction invoked, $\text{tag} \in \text{tags}$ is the logical time at which the invocation occurs, $\text{val} \in \text{vals}$ is the variable values after the invocation, $\text{trig} \in \text{trigs}$ the presence of triggers at the moment of the invocation, $\text{sch} \in \text{schs}$ indicates whether actions are scheduled during the current reaction invocation, and $\text{pld} \in \text{plds}$ the payloads actions carry when they are scheduled.

We use subscripts to denote an individual elements in a vector. The $i^{th}$ entry of $\text{val}$ is denoted as $\text{val}_i$, and the $i^{th}$ entry of $\text{trig}$ is denoted by $\text{trig}_i$. We use the same symbols and define projection functions $\text{val}_i : \text{states} \to V$ and $\text{trig}_i : \text{states} \to \mathbb{B}$, to extract the $i^{th}$ value or trigger at a particular state in a path. We define a projection function $\mathcal{N} : \text{states} \to N$, which extracts a $\text{rxn}$ from a state, and a projection function $\mathcal{G} : \text{states} \to \text{tags}$, which extracts a $\text{tag}$ from a state. We are now ready to start axiomatizing the core reactor semantics.

(2) Reactor semantics. One of the most important sets of axioms is the semantics of the reactor model. We first define an axiom enforcing that the ordering of reaction invocations must be in the order of time tags:

$$\forall i, j \in \text{indices. } \mathcal{G}(\pi_i) \prec \mathcal{G}(\pi_j) \implies i < j.$$  \hspace{1cm} \text{(TimeOrder)}

where $\prec$ is the ordering relation defined earlier. At a time tag, a reactor maintains a deterministic order of execution by executing its reactions in the order of their priorities.

$$\forall i, j \in \text{indices. } \mathcal{G}(\pi_i) = \mathcal{G}(\pi_j) \land \mathcal{P}(\mathcal{N}(\pi_i)) < \mathcal{P}(\mathcal{N}(\pi_j)) \implies i < j.$$  \hspace{1cm} \text{(ReactionOrder)}

where $\mathcal{P} : N \to \mathbb{N}$ is a function that maps a reaction to a priority value, which is a natural number. Besides enforcing a tag order, we also require that a given reaction can only be invoked once at any logical time instant:

$$\forall i, j \in \text{indices. } (\mathcal{N}(\pi_i) = \mathcal{N}(\pi_j) \land i \neq j) \implies \mathcal{G}(\pi_i) \neq \mathcal{G}(\pi_j).$$  \hspace{1cm} \text{(Uniqueness)}
Next, we place constraints on the time tags, where both the timestamp and the microstep need to be non-negative.

\[ \forall i \in \text{indices}. \ T(G(\pi_i)) \geq 0 \land M(G(\pi_i)) \geq 0. \]  

(ValidTag)

For an empty invocation, we require that it must always occur at the end of the trace. In other words, empty invocations cannot occur in between real reaction invocations in a trace. This is desired behavior as the purpose of empty events is to fill in unused entries in the trace array when modeling an execution trace that is shorter than the verification bound.

\[ \forall j \in \text{indices}. N(\pi_j) \neq \epsilon \implies (\forall i \in \text{indices}, i < j. N(\pi_i) \neq \epsilon) \]  

(EmptyLast)

(3) Connections and triggers. Since we represent the system axiomatically, we do not explicitly specify the transition relation between each state variable, but instead specify a set of invariants over the traces being examined.

Let \( p_u \) and \( p_d \) be a pair of upstream and downstream ports, with their values stored in \( val_u, val_d \in val \) and their presence in \( trig_u, trig_d \in trig \). We use a variable \( physical \in B \) to denote whether the connection is a physical connection, and we define a variable \( delay \in \mathbb{N} \) to indicate the logical delay carried by the connection. We also define a schedule operator \( \oplus : tags \times \mathbb{N} \rightarrow tags \) as \( g \oplus t = g' \) s.t. \( T(g') = T(g) + t \) and \( M(g') = 0 \) if \( t > 0 \) and \( M(g') = M(g) + 1 \) if \( t = 0 \). The behavior of a connection is specified as follows.

\[ \forall i \in \text{indices}. (trig_u(\pi_i) \implies (\exists j \in \text{indices}. j > i \land trig_d(\pi_j)) \land val_d(\pi_j) = val_u(\pi_i) \land (\neg physical \implies G(\pi_j) = G(\pi_i) \oplus delay)) \]  

(SendToDownstream)

The SendToDownstream axiom specifies that when the upstream trigger is present at some step in the path, there either exists a step in the path with the downstream trigger present. We further specify the Correspondence axiom below to ensure a one-to-one correspondence between the presence of triggers in logical time.

\[ \forall i \in \text{indices}. (trig_d(\pi_i) \implies (\exists j \in \text{indices}. j < i \land trig_u(\pi_j)) \land val_u(\pi_j) = val_d(\pi_i) \land (\neg physical \implies G(\pi_j) = G(\pi_i) \oplus delay)) \]  

(Correspondence)

Since the default behavior of variables in an axiomatic model is nondeterministic assignment, the SameIfAbsent axiom states that when a trigger is absent, the value corresponding to the trigger stays the same as the previous step.

\[ \forall i \in \text{indices}. trig_d(\pi_i) = false \implies (val_d(\pi_i) = val_d(\pi_{i-1})) \]  

(SameIfAbsent)

If a reaction \( v \in N \) is sensitive to a startup trigger, then it will be invoked at tag (0,0). To encode this behavior, we state a StartupTrigger axiom.

\[ \exists i \in \text{indices}. N(\pi_i) = v \land G(\pi_i) = (0, 0) \land \neg (\exists j \in \text{indices}. N(\pi_j) = v \land j \neq i) \]  

(StartupTrigger)

Besides ports, LF also supports actions and timers as triggers. Let us first start with actions. To axiomatize the behavior of actions, we let \( a \) be an action with its value stored in \( val_a \in vals \), its presence stored in \( trig_a \in trigs \), and a logical delay \( delay_a \in \mathbb{N} \). We further let \( reactions_a \) denote a set of reactions that can schedule the action \( a \). We can state an axiom, ScheduleAction, to encode the semantics of actions.

\[ \exists i \in \text{indices}. \bigvee_{n \in reactions_a} trig_a(\pi_i) = true \implies (\exists j \in \text{indices}. j < i \land N(\pi_j) = n \land G(\pi_i) = G(\pi_j) \oplus delay_a \land sch_a(\pi_j) = true \land val_a(\pi_j) = pld_a(\pi_j)) \]  

(ScheduleAction)

Timers are another important feature for generating periodic events. To axiomatize their behavior, let \( t \) be a timer with an initial offset \( offset \in \mathbb{N} \), a period \( period \in \mathbb{N} \), and its presence stored in
trig_i \in trigs$. Recall also that $k$ is the largest index in $\text{indices}$. Then, a timer must start firing at the pre-defined offset, as stated in $\text{TimerOffset}$.

\[(\mathcal{T}(\mathcal{G}(\pi_k)) \geq \text{offset}) \implies \exists i \in \text{indices}.\ trig_i(\pi_i) = \text{true} \land \mathcal{G}(\pi_i) = (\text{offset}, 0)\]

\[\land (\mathcal{T}(\mathcal{G}(\pi_k)) < \text{offset}) \implies \forall j \in \text{indices}.\ N(\pi_j) \neq \epsilon\]

(TimerOffset)

Intuitively, if the last state of $\pi$ (i.e., $\pi_k$) has reached the timestamp specified by offset, then there must be a state in which the timer trigger is present and the timestamp matches offset. Otherwise, if the last state does not reach the timestamp specified by offset, all the reaction invocations along the path must be non-empty. And the subsequent firing should occur repeatedly at an interval specified by the period. We state two axioms, $\text{TimerPeriod}$ and $\text{TimerSpacing}$ to encode this behavior.

\[\forall i \in \text{indices}.\ trig_i(\pi_i) = \text{true} \implies\]

\[(\exists j \in \text{indices}.\ j > i \land trig_i(\pi_j) = \text{true} \land \mathcal{G}(\pi_j) = \mathcal{G}(\pi_i) \oplus \text{period})\]

(TimerPeriod)

\[\forall i \in \text{indices}.\ trig_i(\pi_i) = \text{true} \implies\]

\[(\exists x \in \mathbb{N}.\ x \geq 0 \land \mathcal{T}(\mathcal{G}(\pi_i)) = \text{offset} + x \times \text{period}) \land M(\mathcal{G}(\pi_i)) = 0\]

(TimerSpacing)

The LF semantics specifies that a reaction is triggered when any of its triggers is present. To encode this behavior, we state a $\text{ReactionTriggers}$ axiom. Let $n \in \mathbb{N}$ be a reaction and let $\text{triggers}_n$ be the set of triggers the reaction $n$ is sensitive to.

\[\forall i \in \text{indices}.\ (\exists t \in \text{triggers}_n.\ t = \text{true}) \implies N(\pi_i) = n\]

(ReactionTriggers)

4) Reaction code. To axiomatize reaction body code written in C, we adopt the preprocessing method presented by Armando et al. [4], in which a block of C code is transformed into a representation called If Normal Form. Starting from a C abstract syntax tree (AST) in If Normal Form, we then translate each if block into an implication where the if condition becomes the antecedent and the then branch becomes the consequent. Since axiomatizing C code is not our contribution, we will not expand this here but will later revisit it in Sec. 7.

5 BOUNDING THE COMPLETENESS THRESHOLD FOR BMC

Recall that the completeness threshold, denoted as $C_0T$, is an attribute of a system $M$ with respect to a property $p$ and a translation scheme such that checking a trace up to $C_0T$ steps without errors proves $M \models p$ [13]. In the context of our work, the system $M$ is a reactor system. The property $p$ is a Safety MTL formula. The translation scheme is the translation of a reactor system to an axiomatic model (Sec. 4) and the translation of a Safety MTL formula to a First-Order Logic formula over a timed word $\pi$ based on the semantics of MTL (Def. 3.2).

Here, we do not seek to compute the precise completeness threshold since such computation is as hard as the BMC problem itself [43]. Instead, we efficiently compute an upper bound on the completeness threshold, and we denote this bound as $\hat{C_0T}$. Since we adopt the event-based semantics introduced by Sirjani et al. [71] (i.e., modeling an LF execution as a sequence of reaction invocations), computing a bound for $\hat{C_0T}$ requires finding the maximum number of reaction invocations within a logical time interval specified by the Safety MTL property. To do this, we construct a state space diagram for an LF program.

State space diagram. Intuitively, each node in the state space diagram consists of a time tag, a set of reactions invoked at this tag, and a set of pending events set to appear at some future time tags. The state space diagram of the running example is shown in Fig. 4.

Formally, a state space diagram is a directed graph $(V, E)$ where $V \subseteq \text{tags} \times P(\text{rxns}) \times P(\text{events})$ and $E \subseteq V \times V$, where $P$ denotes a power set. The symbols $\text{tags}$ and $\text{rxns}$ are defined previously in
To determine the set of initial events, the algorithm identifies the initial firings of timers and reactionsInvoked to the startup trigger used in the program and the initial firings of timers and add them to pendingEvents.

Then, the procedure starts simulating the execution by stepping into a while loop, which can only be exited when the following stop conditions are reached:

1. there are no more pending events (i.e., $|\text{pendingEvents}| = 0$);
2. the simulation completes the time horizon (i.e., $T(currentTag) > \text{horizon}$);
3. a cycle is found in the state space diagram.

During each cycle iteration, the simulation pops all the pending events that are set to appear at the current tag off the pendingEvents queue. Then, reactions triggered by these events are added to the reactionsInvoked list. Note that since this is a lightweight simulation, the C bodies of these reactions will not be simulated, but instead, they are assumed to produce outputs at all the ports and schedule all the actions available to them. This is an over-approximation of the actual behaviour, and leads to the worst-case execution we seek. The set of new outputs and actions produced are inserted back to pendingEvents, waiting to be processed at some future loop iterations. When all the events at the current tag have been processed, the execution advances to the nearest future tag. A new node in the state space diagram is then created for the previous tag, storing the reactionsInvoked list and the pendingEvents queue. There is also an edge from the previous node to this new node.
Algorithm 1 Explore the state space and build a state space diagram

1: procedure Explore(horizon, findLoop)
2:   previousTag, currentTag ← ⊥, pendingEvents ← ∅
3:   pendingEvents ← getInitialEvents()
4:   stop ← pendingEvents.isEmpty()
5:   while not stop do
6:     currentEvents ← processEarliestEvents(pendingEvents)
7:     reactionsInvoked ← triggerReactions(currentEvents)
8:     for each reaction in reactionsInvoked do
9:       pendingEvents ← pendingEvents ∪ processReactionEffects(reaction)
10:   end for
11:   if previousTag = null then
12:     currentNode ← initializeCurrentNode()
13:   else if currentTag.timestamp > previousTag.timestamp then
14:     if findLoop and duplicateNodeExists() then
15:       identifyAndMarkLoop(); return
16:     end if
17:     newNode ← createNewNode(currentTag, reactionsInvoked, pendingEvents)
18:     addToDiagram(newNode)
19:     currentNode ← newNode
20:   else
21:     currentNode ← updateCurrentNode(reactionsInvoked, pendingEvents)
22:   end if
23:   previousTag ← currentTag
24:   currentTag ← pendingEvents.peek().tag
25:   if pendingEvents.isEmpty() or currentTag.timestamp > horizon.timestamp then
26:     stop ← true
27:   end if
28: end while
29: finalizeDiagram()
30: end procedure

We prove that the state space diagram generated using this algorithm is indeed the worst case.

Proposition 5.2. Algorithm 1 produces a state space diagram representing the theoretical worst-case execution of an LF program.

Proof. We prove this lemma by induction on the logical tag. In the base case, we focus on the first time tag and show that the algorithm computes the theoretical worst-case execution path up to the conclusion of the first time tag. When the algorithm begins, it adds initial events (line 3) including the startup trigger (set to occur at tag (0, 0)) and the initial firings of timers (could happen at any time tags depending on the offsets) to pendingEvents. At this point, these initial events are not dependent on any prior state, but their addition to pendingEvents ensures that their effects will be considered when computing the next state, thus establishing state-dependency from the very beginning. If there are no initial events, the algorithm terminates and we arrive at a correct state space diagram that is empty. If there are initial events, then we enter the while loop (line 5) and pop off the earliest events at line 6. Reactions triggered by these events are added to reactionsInvoked and all of their effects are added back to pendingEvents. This means that the effects of all the
reactions invoked at the current state are taken into account when constructing the next state. These effects can influence both the state at the next time tag and the states at subsequent time tags, which ensures the propagation of state-dependency across different time tags. If the newly added effects happen at some future tag, then we advance the current tag (line 24) and are done with the first tag, leaving us with a correct state space diagram constructed up to the conclusion of the first tag with as many reactions invoked as possible at the first tag. On the other hand, if the added events are instantaneous, e.g., setting output ports connected to a downstream input port with zero delay, then line 24 maintains the current tag, runs the loop iteration until we are done with the first tag. The resulting state space diagram is representative of the theoretical worst-case execution because during the loop iterations, all the possible reaction invocations have been accounted for. We thus conclude the base case.

For the inductive step, we make the inductive hypothesis that the algorithm has correctly computed a state space diagram up to some time tag later than the first tag, we seek to show that the algorithm can correctly compute the next tag. At this point, there must be unprocessed events in pendingEvents and the current tag must not have reached horizon, otherwise the algorithm would have been terminated at line 26. By our inductive hypothesis, pendingEvents contains the effects of all the reactions invoked at the previous states. Hence, the current state is dependent on all the previous states, and any reaction invoked at the current state will also take into account the effects of the reactions invoked at the previous states. We begin handling the next tag by popping the earliest events, checking the set of reactions invoked by these popped events, and again add all the effects of the invoked reactions back to pendingEvents. By adding all the effects of the reactions invoked at the current state to pendingEvents, we ensure that these effects are propagated to the next state and states at subsequent time tags. This maintains state-dependency across different time tags. If the newly added events are in the future, then we update currentTag (line 24) and are done with the next tag. If not, line 24 does not update currentTag, and we run the loop iteration until the next tag is finished, at which point, we are left with a state space diagram correctly constructed up to the conclusion of this next tag since all the possible reaction invocations have been accounted for. Invoking all possible reactions at each time tag results in the maximal number of reactions invoked over the entire execution, which constitutes the worst-case scenario.

\[ \text{Cycle detection.} \] As an optimization, the simulation seeks to find a cycle in a state space diagram. LF programs often exhibit periodic behaviors (e.g., the execution of our running example shown in Fig. 4). For such programs, the simulated execution does not need to simulate until the completion of a horizon but only needs to simulate until the completion of a period (i.e., a cycle). The rest of the execution up to the horizon can be inferred from the periodic behavior. To check whether an execution completes a period, when the execution advances to a new tag, the simulation checks whether the newly added node is equivalent to any of the previous nodes. Two nodes, \( n_1 \) and \( n_2 \), are considered equivalent if

\[
\begin{align*}
1. & \quad \text{rxns}(n_1) = \text{rxns}(n_2); \\
2. & \quad \{ \text{trigs}(e_1) \subseteq \text{trigs} \mid e_1 \in \text{events}(n_1) \} = \{ \text{trigs}(e_2) \subseteq \text{trigs} \mid e_2 \in \text{events}(n_2) \}; \\
3. & \quad \{ \text{tag}_e(e_1) - \text{tag}_n(n_1) \mid e_1 \in \text{events}(n_1) \} = \\
& \quad \{ \text{tag}_e(e_2) - \text{tag}_n(n_2) \mid e_2 \in \text{events}(n_2) \}.
\end{align*}
\]

The cycle detection approach is similar to [30], which constructs state space diagrams for synchronous dataflow graphs [46].

\[
\begin{align*}
\text{From diagram to } \hat{CT}. \quad \text{Once the state space diagram is generated, we can then compute } \hat{CT} \text{ from it. For this, we state two theorems.}
\end{align*}
\]
Reactions invoked:
ADASModel.l.reaction_0
ADASModel.c.reaction_0
ADASModel.p.reaction_0
Pending events:
(ADASModel.c.t, (17 msec, 0))
(ADASModel.l.t, (34 msec, 0))
(ADASModel.p.a, (50 msec, 0))

Fig. 4. An abstract view of the state space diagram for the running example in Fig. 1. In this abstract view, except for S0, which is shown in full, each node consists of (from left to right) an identifier, the number of reactions invoked at the current tag, the number of events pending, and (at the bottom) the current tag.

**Theorem 5.3**: \( \hat{C^T} \text{ w.r.t. a cycle-free diagram.} \) The completeness threshold w.r.t. an LF program that generates a cycle-free state space diagram \( (V, E) \) is upper bounded by \( \hat{C^T} \) such that

\[
\hat{C^T} = \sum_{n \in V} |rxns(n)|.
\]

**Proof.** A cycle-free diagram is by construction a finite linear sequence of nodes. For a node \( n \in V \), the number of reaction invocations is given by \( |rxns(n)| \). The maximum possible number of reaction invocations, i.e., \( \hat{C^T} \), can be computed by summing up the number of reaction invocations of each node. We then obtain the desired expression. \( \blacksquare \)

Intuitively, this \( \hat{C^T} \) counts the number of reaction invocations up to the completion of the horizon. If a cycle is found, we use its periodicity to infer \( \hat{C^T} \).

**Theorem 5.4**: \( \hat{C^T} \text{ w.r.t. a diagram with a cycle.} \) The completeness threshold w.r.t. an LF program that generates a state space diagram \( (V, E) \) with a cycle is upper bounded by \( \hat{C^T} \) such that

\[
\hat{C^T} = \sum_{n'' \in V'' \cup \{n_c\}} |rxns(n'')| + \sum_{n' \in V'} |rxns(n')| \times \text{iterations},
\]

where \( V' \) denotes the set of nodes that form the cycle, \( V'' = V \setminus V' \) denotes the rest of the nodes, \( n_c \in V' \) denotes the node where the cycle begins, iterations = \( \left\lceil \frac{\text{horizon} - T(\text{tag}(n_c))}{\rho} \right\rceil \), and \( \rho \) denotes the cycle period.

**Proof.** A state space diagram with a cycle has a initialization phase, consisting of nodes \( V'' \cup \{n_c\} \), and a periodic phase, consisting of nodes \( V' \). Note that the first time of reaching \( n_c \in V' \) counts toward the initialization phase since the periodic phase cannot start without first reaching \( n_c \). The overall \( \hat{C^T} \) is the sum of reaction invocations during both the initialization phase and the periodic
phase. The first expression, $\sum_{n'' \in V \cup \{n_c\}} |rxns(n'')|$, follows directly from the proof of Theorem 5.3. The second expression over-approximates the number of reaction invocations during the periodic phase by multiplying the number of invocations during one iteration, given by $\sum_{n' \in V} |rxns(n')|$ (also directly following the proof of Theorem 5.3), and, iterations, the minimum number of cycle iterations required to guarantee completing the time horizon. The value of iterations can be obtained by taking the remaining horizon after entering the periodic phase, i.e., horizon – $T(\tag{n_c})$, dividing it by the period of the cycle, $\rho$, and applying the ceiling operation. We then obtain the desired expression.

6 IMPLEMENTATION

We now describe the implementation details of the LF Verifier, and walk through the steps to verify the safety property in the running example. The LF Verifier is embedded inside the Lingua Franca compiler, which is open-source on Github. From the user’s perspective, using the LF Verifier simply requires extending the original program with @property annotations, and the rest is invoking the compiler.

Recall from Fig. 2 that the user first provides an LF program along with annotations specifying temporal properties of the program in Safety MTL. Once the input files are provided, they are then passed into the LF Verifier, which parses LF code, reaction code (written in C, in our case), and the Safety MTL annotations. Based on the parsed LF code, LF Verifier analyzes the topology of the system and generates an axiomatic model that can be processed by Uclid5. In the rest of the section, we will focus on the modules used by the LF Verifier to verify properties and ignore the modules used for compiling the LF program.

Annotations. The user specifies an MTL property using an @property annotation. At the time of writing, the @property annotation requires three parameters: (i) the name of the property, (2) the tactic used to check the property (only bmc is supported for now), and (3) the MTL formula. An valid MTL formula follows the syntax shown in Fig. 5. Safety MTL is enforced by ensuring “int” to be finite non-negative integers. Reaction invocations (resp., values of LF components) are specified using fully qualified names of the reactions (resp., the components).

---

GitHub link: [https://github.com/lf-lang/lingua-franca](https://github.com/lf-lang/lingua-franca)
**Parsing.** After opening the LF file provided by the user, the LF Verifier parses the LF code using an “LF Parser” based on the Xtext framework [11, 25], and it parses the reaction body code (in C) and the MTL annotations using ANTLR4 parsers [63] (shown in Fig. 2 as “C Parser” and “MTL Parser”), generating a parse tree for C and another parse tree for MTL.

**AST transformations.** Once the C parse tree is generated by ANTLR4, the “C AST Builder” is used to build an abstract syntax tree (AST). Our implementation uses the technique presented by Armando et al. [4], which supports a subset of the C language. For language features we do not currently support, including pointers, loops, and macros, the AST builder marks these AST nodes as *opaque* and returns a warning to the user. While the support of C is incomplete, the LF Verifier currently can analyse arithmetic operations, variable assignments, LF runtime APIs, and if statements (including nesting if statements), which are arguably expressive enough to cover many real-time systems applications. We will discuss our plan to expand the support for C in Sec. 8.

Once the C AST is built, it is sent to the “INF Converter”, where INF stands for If Normal Form, a term introduced in Armando et al. [4]. The main idea is to flatten the nested if statements until an if condition is associated with each atomic statement. For example, the two reaction bodies in ADASProcessor in the running example (Fig. 1) effectively turn into

```c
/* (C) Reaction body 1 */
if (true) lf_schedule(a, 0);
if (true) self->requestStop = 1;
/* (C) Reaction body 2 */
if (self->requestStop == 1) lf_set(out1, 1);
```

The latter reaction body does not change since it follows the If Normal Form before transformation.

**Computing $\hat{C}T$.** Let us come back to the LF code. After the LF program passes through the “LF Parser”, it goes into a “Dependency Analyzer,” which extracts the dependency relations between reactions. When the dependency analysis is done, the LF Verifier invokes a “$\hat{C}T$ Calculator” which executes Algorithm 1 in Sec. 5 to obtain an upper bound on the completeness threshold.

To check the property in the running example, the LF Verifier first computes a logical time horizon from the property, which, in this case, is 65 ms: the outer $G[0, 10 \text{ ms}]$ operator requires 10 ms while the inner $F[0, 55 \text{ ms}]$ operator requires another 55 ms. The $F[0]$ does not contribute to the horizon due to the interval of 0. Therefore, to ensure the completeness of BMC, the model needs to unroll until 60 ms is reached.

The state space diagram of the program (Fig. 4) shows that within an interval of 65 ms, the program reaches six states (S0-S5). $\hat{C}T$, i.e., the maximum sum of reaction invocations along the six states, is $\hat{C}T = 3+2+3+2+2+1 = 13$. The “$\hat{C}T$ Calculator” passes this number to the downstream model generators.

**Model generation.** During the model generation phase, three generators work together to produce a single Uclid5 model: an “LF Axiom Generator”, a “Reaction Axiom Generator”, and an “MTL2FOL Generator.” The “LF Axiom Generator” generates the definitions and axioms (except the reaction axioms) presented in Sec. 4. Here is a snippet of the generated Uclid5 code for the preliminary definitions. In this paper, we do not explain the Uclid5 syntax in detail, but instead refer the reader to [64, 70, 72].

```c
/* (Uclid5) Preliminary definitions */
const START : integer = 0; // The start index of the trace
const END : integer = 13; // The end index of the trace (i.e., CT bound)
const END_TRACE : integer = 19; // The end index of the trace with padding
typedef state_t = {
    rxn_t, tag_t, val_t, trig_t, sch_t, pld_t}; // A state is a 6-tuple.
typedef path_t = [integer]state_t; // A path is a sequence of states indexed by integers.
```
The "Reaction Axiom Generator" generates Uclid5 axioms from the C AST in the If Normal Form. Again, we use the two reaction bodies in ADASProcessor as examples:

```
/* (Uclid5) Axioms of reaction 1 of ADASProcessor */
axiom (finite_forall (i : integer) in indices :: (i > START && i <= END) ==> ( ADASModel_p_reaction_0(rxn(i))) ==> (((true ==> ((ADASModel_p_requestStop(s(i)) == 1)) && (true ==> (ADASModel_p_a_scheduled(d(i)))))))));
```

As mentioned in Sec. 4, the generated reaction axioms are derived from a C AST in the If Normal Form and work in the same way as the other axioms by specifying relationships between variables across multiple time steps. For example, line 16 of the listing above is a direct translation of line 5 (i.e., if (self->requestStop==1) lf_set(out1, 1);) in the C code listing above. The if statement becomes an implication where the if condition is the antecedent and the then branch of the if statement becomes the consequent.

The "MTL2FOL Generator" generates a First-Order Logic (FOL) property over the path in Uclid5 from the given MTL property. For the property in the running example, the generator produces the following:

```
/* (Uclid5) The FOL property translated from user-defined MTL property */
define p(i : step_t) : boolean = (((finite_forall (j0 : integer) in indices :: (j0 >= i && j0 <= END && !isNULL(j0) && (pi1(g(j0)) >= (pi1(g(i)) + 0)) && (pi1(g(j0)) <= (pi1(g(i)) + 10000000)))) ==> (((((ADASModel_l_reaction_0(rxn(j0))) && (((finite_exists (j1 : integer) in indices :: j1 >= j0 && j1 <= END && !isNULL(j1) && (((ADASModel_p_requestStop(s(j1)) == 1)) && (tag_same(g(j1), tag_delay(g(j0), 0)))))))))) ==> ((((finite_exists (j1 : integer) in indices :: j1 >= j0 && j1 <= END && !isNULL(j1) && (((ADASModel_b_brakesApplied(s(j1)) == 1)) && (pi1(g(j1)) >= (pi1(g(j0)) + 0)) && (pi1(g(j1)) <= (pi1(g(j0)) + 55000000)))))))))))));
```

The "MTL2FOL Generator" recursively traverses the subformulas of the given MTL formula and builds up an FOL formula in a bottom-up direction. The generated FOL formula implements the semantics of MTL shown in Sec. 3.

Checking and reporting results. Once the model is generated, it is then passed into Uclid5, which generates an SMT file and invokes the Z3 SMT solver [18]. If the result is UNSAT, then the temporal properties are valid w.r.t. the formal model of the system. If the result is SAT, a counterexample trace is returned by Uclid5, which can help guide the user to revise the LF program. The process iterates until the unsafe behaviors are eliminated from the program. For the property in the running example, the LF VERIFIER reports UNSAT with no counterexamples found, hence the program satisfies the MTL property.

On the other hand, there are a few locations in the program where bugs can be introduced to invalidate the property.
Incorrect functional behavior. The property could be invalidated if reactions have incorrect functional behavior, e.g., the second reaction of ADASProcessor (lines 48-51) produces an output to the wrong port.

```
48 reaction(a) -> out1, out2 {=
49   if (self->requestStop == 1)
50     lf_set(out2, 1);  // Original: lf_set(out1, 1);
51   } deadline(20ms) {=...=}
```

Incorrect composition. The property is also invalidated when the components are not correctly composed, e.g., when LiDAR is not connected to ADASProcessor.

```
// l.out -> v.in1  // Originally connected
73 c.out -> v.in2
74 v.out2 -> d.in
76 v.out1 -> b.inADAS after 5ms
77 p.out -> b.inPedal
```

Incorrect timing. Timing behavior also plays a crucial role in determining the truth value of the property. The timer in LiDAR having a large enough offset, the logical action in ADASProcessor having a larger delay, or the connection between ADASProcessor and Brakes having a larger delay could invalidate the given property.

```
5 timer t(11ms,17ms)  // Original: timer t(0,17ms)
40 logical action a(51ms)  // Original: logical action a(50ms)
76 v.out1 -> b.inADAS after 6ms  // Original: v.out1 -> b.inADAS after 5ms
```

If any of the bugs (or their combinations) occurs, the LF Verifier reports SAT and displays a counterexample path showing that the reaction in Brakes does not get triggered within 13 steps of the generated model. The user can then work through the counterexample to identify the specific bugs in the program.

7 EVALUATION

**Benchmarks.** We evaluate our approach against a set of 22 LF programs drawn from real-life applications [2, 17, 28, 33, 49, 53, 71, 75], most of which are CPS applications with real-time properties, as well as benchmarks for Erlang [22], Lustre [16, 33, 40], actors [36] and RTOSes [24]. Thirteen out of the 22 are safe examples and 9 are unsafe examples. The benchmark focuses on verifying the three types of properties introduced in Sec. 4: State properties are named Type I, dataflow properties Type II, and mixed properties Type III. The benchmark suite is also open-source on GitHub.

**Results.** Table 1 shows the results of evaluating our approach against the verification benchmark suite. The experiments are conducted on a personal laptop running macOS version 11.7 with an 2.3 GHz 8-Core Intel Core i9 and 16GB RAM. The versions of tools used are Uclid5 0.9.5 and Z3 4.8.8.

The LF Verifier correctly checks 21 out of 22 programs automatically. The one program the LF Verifier fails to check is TrafficLight because the C implementation used in one of the reaction bodies exceeds the subset of C currently supported by the LF Verifier at the time of writing. The results shows that our tool is sound but incomplete. We expect a future version of LF Verifier supporting an larger subset of C will be able handle this program. We discuss our plan to achieve this in Sec. 8.

2GitHub link: https://github.com/lf-lang/lf-verifier-benchmarks
Table 1. Benchmark results.

<table>
<thead>
<tr>
<th>Program</th>
<th>Type</th>
<th>#Rxns</th>
<th>(\hat{CT})</th>
<th>Safe?</th>
<th>LOC</th>
<th>TIME (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>LF</td>
<td>Uclid5</td>
<td>GenUclid</td>
<td>GenSMT</td>
</tr>
<tr>
<td>ADASModel [75]</td>
<td>III</td>
<td>6</td>
<td>13</td>
<td>yes</td>
<td>78</td>
<td>875</td>
</tr>
<tr>
<td>AircraftDoor [53]</td>
<td>III</td>
<td>3</td>
<td>3</td>
<td>yes</td>
<td>37</td>
<td>402</td>
</tr>
<tr>
<td>Alarm [33]</td>
<td>III</td>
<td>2</td>
<td>2</td>
<td>no</td>
<td>24</td>
<td>338</td>
</tr>
<tr>
<td>CoopSchedule [24]</td>
<td>I</td>
<td>2</td>
<td>18</td>
<td>no</td>
<td>29</td>
<td>659</td>
</tr>
<tr>
<td>Elevator [16]</td>
<td>III</td>
<td>13</td>
<td>33</td>
<td>yes</td>
<td>118</td>
<td>1899</td>
</tr>
<tr>
<td>Election [17]</td>
<td>I</td>
<td>9</td>
<td>18</td>
<td>yes</td>
<td>73</td>
<td>1045</td>
</tr>
<tr>
<td>Election2 [17]</td>
<td>I</td>
<td>6</td>
<td>9</td>
<td>no</td>
<td>61</td>
<td>700</td>
</tr>
<tr>
<td>Factorial [16]</td>
<td>III</td>
<td>3</td>
<td>33</td>
<td>yes</td>
<td>38</td>
<td>461</td>
</tr>
<tr>
<td>Fibonacci [16]</td>
<td>III</td>
<td>5</td>
<td>55</td>
<td>yes</td>
<td>48</td>
<td>649</td>
</tr>
<tr>
<td>PingPong [36]</td>
<td>I</td>
<td>4</td>
<td>16</td>
<td>no</td>
<td>39</td>
<td>510</td>
</tr>
<tr>
<td>Pipe [22, 40]</td>
<td>I</td>
<td>5</td>
<td>24</td>
<td>no</td>
<td>54</td>
<td>2059</td>
</tr>
<tr>
<td>ProcessMsg [24]</td>
<td>I</td>
<td>4</td>
<td>19</td>
<td>yes</td>
<td>33</td>
<td>487</td>
</tr>
<tr>
<td>ProcessSync [24]</td>
<td>I</td>
<td>1</td>
<td>3</td>
<td>yes</td>
<td>12</td>
<td>284</td>
</tr>
<tr>
<td>Railroad [2]</td>
<td>I</td>
<td>10</td>
<td>20</td>
<td>yes</td>
<td>115</td>
<td>2333</td>
</tr>
<tr>
<td>Ring [22]</td>
<td>II</td>
<td>4</td>
<td>8</td>
<td>yes</td>
<td>39</td>
<td>897</td>
</tr>
<tr>
<td>RoadsideUnit [28]</td>
<td>I</td>
<td>8</td>
<td>21</td>
<td>yes</td>
<td>70</td>
<td>1536</td>
</tr>
<tr>
<td>SafeSend [22]</td>
<td>II</td>
<td>4</td>
<td>4</td>
<td>yes</td>
<td>37</td>
<td>479</td>
</tr>
<tr>
<td>Subway [33]</td>
<td>II</td>
<td>10</td>
<td>13</td>
<td>no</td>
<td>70</td>
<td>1097</td>
</tr>
<tr>
<td>Thermostat [49]</td>
<td>I</td>
<td>6</td>
<td>10</td>
<td>yes</td>
<td>59</td>
<td>652</td>
</tr>
<tr>
<td>TrafficLight† [16]</td>
<td>I</td>
<td>4</td>
<td>14</td>
<td>-</td>
<td>69</td>
<td>-</td>
</tr>
<tr>
<td>TrainDoor [71]</td>
<td>II</td>
<td>3</td>
<td>3</td>
<td>no</td>
<td>31</td>
<td>398</td>
</tr>
<tr>
<td>UnsafeSend [22]</td>
<td>II</td>
<td>4</td>
<td>4</td>
<td>no</td>
<td>38</td>
<td>478</td>
</tr>
</tbody>
</table>

† No result is produced because the C code in this program exceeds the analyzable subset.

We find that the majority of the computation time is spent on model generation: the median generation time for the Uclid5 model is 2.93 seconds, and the median generation time for the SMT formulas from the Uclid5 model is 12.15 seconds. However, once the SMT models are generated, solving them is very fast: the median solve time by Z3 is 0.31 seconds. The median total time is 15.9 seconds. Fig. 6 shows the distributions of our benchmark result. The total time scales almost linearly as the LF lines of code (LF LOC) and \(\hat{CT}\) increase.

8 CONCLUSION AND FUTURE WORK

In this paper, we present a methodology for building verifiable Cyber-Physical Systems using LINGUA FRANCA. We specify CPS applications in LF, which provide language primitives for time and concurrency, and automatically generate axiomatic models in Uclid5 directly from LF programs to perform BMC. The properties of interest in this work are safety
properties specified as Safety MTL formulas. Our axiomatic approach is sound w.r.t. all valid LF runtime implementations, and to ensure completeness of BMC, we present an algorithmic method to compute an upper bound on the completeness threshold. We implement our automated verification workflow in the LF Verifier, and our evaluation shows that the LF Verifier can effectively check a suite of benchmark programs fully automatically.

For the future work of the LF Verifier, it is our plan to include larger programs (in terms of lines of LF code) in the benchmark suite and make the LF Verifier more scalable. To tame the explosion of complexity, we plan to explore two strategies: (i) detecting reactors in an LF program that do not affect the truth value of a given property using static analysis and remove them from the axiomatic model, and (ii) using assume-guarantee reasoning to support compositional verification based on the modular structure of LF. In addition, we plan to support more complex C reaction bodies by expanding our “C AST Builder” or by offloading C code to external frameworks such as CBMC [44]. We also aim to support unbounded verification by potentially leveraging temporal induction [5]. Furthermore, we hope to support other LF features including physical actions, deadlines, modal models, etc. We leave these improvements for future work.

ACKNOWLEDGMENTS

The work in this paper was supported in part by the National Science Foundation (NSF), awards #CNS-1836601 (Reconciling Safety with the Internet) and #CNS-2233769 (Consistency vs. Availability in Cyber-Physical Systems) and the iCyPhy Research Center (Industrial Cyber-Physical Systems), supported by Denso, Siemens, and Toyota. This work was also supported in part by DARPA grant FA8750-20-C-0156 and by Intel.

REFERENCES


Towards Building Verifiable CPS using Lingua Franca


Received 23 March 2023; revised 2 June 2023; accepted 30 June 2023