Towards Building Verifiable CPS using Lingua Franca

SHAOKAI LIN, University of California, Berkeley, USA
YATIN A. MANERKAR, University of Michigan, USA
MARTEN LOHSTROH, 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 and 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;**

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

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.

Authors' addresses: S. Lin, M. Lohstroh, S.-J. Yu, E. A. Lee, and S. A. Seshia, University of California, Berkeley, Cory Hall, Berkeley, CA, 94720-1770, USA; emails: {shaokai, marten, shengjungyu, eal}@berkeley.edu, sseshia@eecs.berkeley.edu; Y. A. Manerkar, University of Michigan, 2260 Hayward Street, Ann Arbor, MI, 48109-2121, USA; email: manerkar@umich.edu; E. Polgreen, University of Edinburgh, 10 Crichton St, Edinburgh, Lothian, EH8 9AB, UK; email: elizabeth.polgreen@ed.ac.uk; C. Jerad, University of Manouba, Campus universitaire de la Manouba, Manouba, Manouba Governorate, 2010, Tunisia; email: chadlia.jerad@ensi-uma.tn.

1539-9087/2023/09-ART155 $15.00
https://doi.org/10.1145/3609134

Additional Key Words and Phrases: Cyber-physical systems, concurrency, safety MTL, axiomatic modeling, automated verification, model-based design

ACM Reference format:
Shaokai Lin, Yatin A. Manerkar, Marten Lohstroh, Elizabeth Polgreen, Sheng-Jung Yu, Chadlia Jerad, Edward A. Lee, and Sanjit A. Seshia. 2023. Towards Building Verifiable CPS using Lingua Franca. *ACM Trans. Embedd. Comput. Syst.* 22, 5s, Article 155 (September 2023), 24 pages.
https://doi.org/10.1145/3609134

1 INTRODUCTION

Cyber-physical systems (CPS) [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 [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) [12, 13] and satisfiability modulo theory (SMT) solvers [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 [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) [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) [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 [8], AUTOSAR [56], distributed databases [48], and labs in embedded systems courses [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 [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 Section 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. Section 2 presents a running example of a simplified ADAS system encoded in LF. Section 3 presents background on Safety Metric Temporal Logic (Safety MTL) and Bounded Model Checking (BMC). In Section 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, Section 5 presents a method to compute an upper bound for a completeness threshold. In Section 6, we present the implementation details of the LF Verifier and work through verifying a property in the running example. In Section 7, we evaluate the LF Verifier against a benchmark suite. We conclude the paper and discuss the future work in Section 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 Figure 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.

```
1  target C
2  reactor Camera {
3    output out  :{=c_frame_t=}
4    state  frame:{=c_frame_t=}
5    timer  t(0,17ms) //60fps
6    reaction(t) -> out {=
7      // Capture a frame.
8      self->frame = ...;
9      lf_set(out,self->frame);
10   =}
11 }
12 reactor LiDAR {
13   output out  :{=l_frame_t=}
14   state  frame:{=l_frame_t=}
15   timer  t(0,34ms) //30fps
16   reaction(t) -> out {=
17     // Capture a frame.
18     self->frame = ...;
19     lf_set(out,self->frame);
20   =}
21 }
22 reactor Pedal {
23   output out:int
24   physical action a
25   reaction(a) -> out {=
26     lf_set(out, 1); =}
27 }
```

```
26 reactor Brakes {
27   input inADAS:int
28   input inPedal:int
29   state brakesApplied:int(0)
30   reaction(inADAS,inPedal){=
31     // Actuate brakes.
32     self->brakesApplied = 1;
33   =} deadline(10ms) {=...=}
34 }
35 reactor ADASProcessor {
36   input  in1:{=c_frame_t=}
37   input  in2:{=l_frame_t=}
38   output out1:int
39   output out2:int
40   logical action a(50ms)
41   state requestStop:int
42   reaction(in1,in2) -> a {=
43     // ... Detect danger
44     // and request stop.
45     lf_schedule(a, 0);
46     self->requestStop = 1;
47   =}
48   reaction(a) -> out1,out2{=
49     if (self->requestStop==1)
50       lf_set(out1, 1);
51   =} deadline(20ms) {=...=}
52 }
```

```
53 reactor Dashboard {
54   input in:int
55   state received:int
56   reaction(in) {=
57     self->received = 1; =}
58 }
59
60 @property(name="responsive",
61 tactic="bmc", spec="G[0,10 ms]
62 ((ADASModel_l_reaction_0 &&
63 (F[0](ADASModel_p_requestStop
64 ==1)))==>(F[0,55 ms](ADASModel
65 _b_brakesApplied==1)))")
66 main reactor ADASModel {
67   c = new Camera()
68   l = new LiDAR()
69   v = new ADASProcessor()
70   b = new Brakes()
71   d = new Dashboard()
72   p = new Pedal()
73   l.out  -> v.in1
74   c.out  -> v.in2
75   v.out2 -> d.in
76   v.out1 -> b.inADAS after 5ms
77   p.out  -> b.inPedal
78 }
```

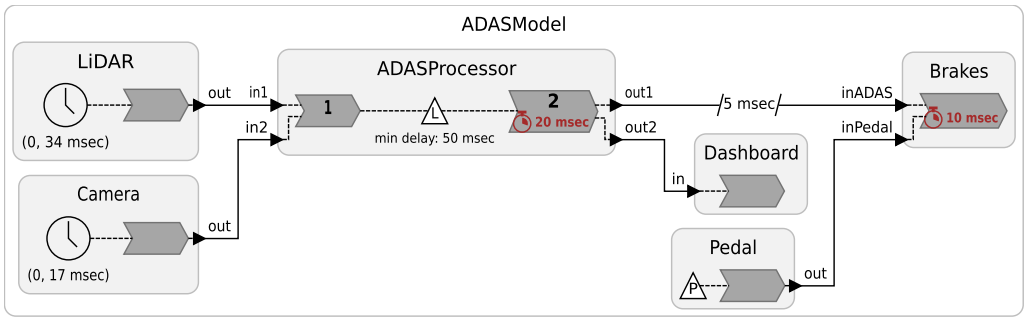

Fig. 1. LF code and diagram of a simplified model of an ADAS system.

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 35 & 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 Figure 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 Figure 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

```
G[0, 10 ms]((ADASModel_l_reaction_0 && (F[0](ADASModel_p_requestStop == 1)))
        ==> (F[0, 55 ms](ADASModel_b_brakesApplied == 1)))
```

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 Section 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 Σ of events, the formulas of MTL are built up from Σ by Boolean connectives and time-constrained versions of the next operator $\mathbf{X}$, the until

operator $\mathbf{U}$, and the dual until operator $\widetilde{\mathbf{U}}_I$ as follows:

$$\psi ::= \textbf{true} \mid \textbf{false} \mid a \mid \psi_1 \wedge \psi_2 \mid \psi_1 \vee \psi_2 \mid \mathbf{X}_I \psi \mid \psi_1 \mathbf{U}_I \psi_2 \mid \psi_1 \widetilde{\mathbf{U}}_I \psi_2$$

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 Σ, a word position $i < |\pi|$, and an MTL formula ψ, the satisfaction relation $(\pi, i) \models \psi$ (read π satisfies ψ at position i) is defined as follows.

$$\begin{aligned}
(\pi, i) &\models a && \text{iff} \quad \sigma_i = a \\
(\pi, i) &\models \psi_1 \wedge \psi_2 && \text{iff} \quad (\pi, i) \models \psi_1 \wedge (\pi, i) \models \psi_2 \\
(\pi, i) &\models \psi_1 \vee \psi_2 && \text{iff} \quad (\pi, i) \models \psi_1 \vee (\pi, i) \models \psi_2 \\
(\pi, i) &\models \mathbf{X}_I \psi && \text{iff} \quad i < |\pi| \ \wedge \ \tau_{i+1} - \tau_i \in I \ \wedge \ (\pi, i+1) \models \psi \\
(\pi, i) &\models \psi_1 \mathbf{U}_I \psi_2 && \text{iff} \quad \exists j, i \leq j \leq |\pi| \wedge \tau_j - \tau_i \in I . [(\pi, j) \models \psi_2 \\
& && \qquad \wedge \ \forall k, i \leq k < j . (\pi, k) \models \psi_1] \\
(\pi, i) &\models \psi_1 \widetilde{\mathbf{U}}_I \psi_2 && \text{iff} \quad \forall j, i \leq j \leq |\pi| \wedge \tau_j - \tau_i \in I . [(\pi, j) \models \psi_2 \\
& && \qquad \vee \ \exists k, i \leq k < j . (\pi, k) \models \psi_1]
\end{aligned}$$

From here, additional temporal operators can be defined: *constrained eventually* $\mathbf{F}_I$ and *constrained globally* $\mathbf{G}_I$, where $\mathbf{F}_I \psi = \textbf{true} \mathbf{U}_I \psi$, and $\mathbf{G}_I \psi = \textbf{false} \widetilde{\mathbf{U}}_I \psi$. 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 $\mathbf{U}_I$ 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 $\mathbf{X}_I$ and the dual until operator $\widetilde{\mathbf{U}}_I$ 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 ψ is valid along a path π 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 $\mathcal{CT}$, is a minimum number of k such that $\pi \models_{\mathcal{CT}} \psi \implies \pi \models \psi$, i.e. if π satisfies ψ up to $\mathcal{CT}$ steps, then π satisfies the property ψ. In Section 5, we present an algorithm for upper-bounding $\mathcal{CT}$.

4 AXIOMATIC SEMANTICS FOR LF FOR VERIFICATION

Figure 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 ψ, determine whether all execution paths $\bar{\pi}$ of r satisfy the formula ψ, i.e., $\bar{\pi} \models \psi$.

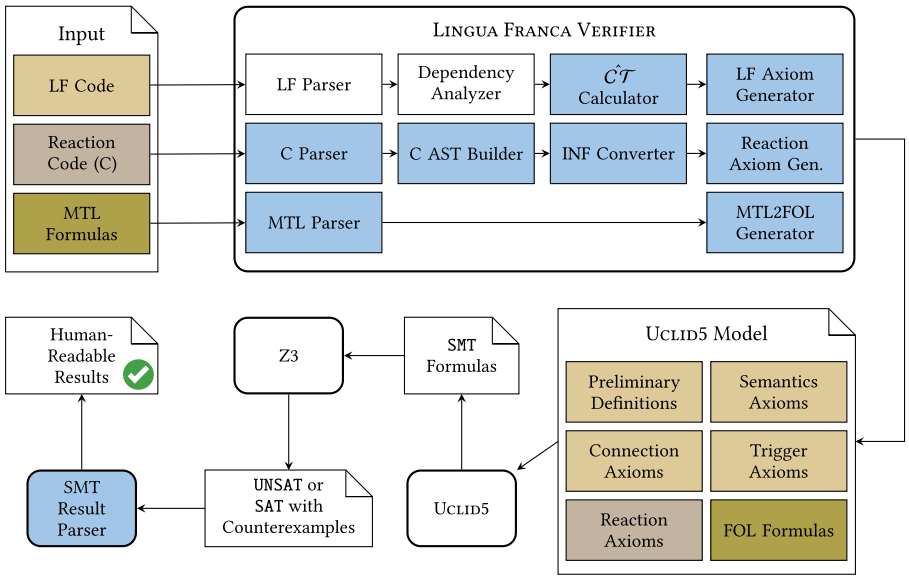

Fig. 2. Verification workflow of LF programs and the architecture of the LF Verifier. Blue represents software modules implemented in this work. The other matching colors indicate correspondence between the input program and the generated model.

$$\overline{indices} = (1, 2, ..., k) \qquad val \in vals = V^{|S|} \qquad \pi_i = (rxn, tag, val, trig, sch, pld)$$
$$rxn \in rxns = N \cup \{\epsilon\} \qquad trig \in trigs = \mathbb{B}^{|T|} \qquad \bar{\pi} = (\pi_1, \pi_2, ..., \pi_k) \in \Pi$$
$$t, m \in \mathbb{N} \qquad sch \in schs = \mathbb{B}^{|A|} \qquad indices = \overline{indices} \cup (k{+}1, ..., k{+}|N|)$$
$$tag = (t, m) \in tags \qquad pld \in plds = \mathbb{V}^{|A|} \qquad \pi = \bar{\pi} \cup (\pi_{k+1}, ..., \pi_{k+|N|}) \in \Pi$$

Fig. 3. Logical time and path definitions.

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.

(1) 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. *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. $\mathcal{G}$) and italicized words (e.g. 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**}).

(2) Logical time and path. The definitions covered in this segment are shown in Figure 3. A logical time tag is defined as $tag := (t, m)$, 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 $\mathcal{T} : tags \rightarrow \mathbb{N}$ for mapping a time tag to its time value and $\mathcal{M} : tags \rightarrow \mathbb{N}$ for mapping a time tag to its microstep index. To compare two tags, we define an ordering relation

$$\prec = \{(tag_1, tag_2) \in tags^2 \mid \mathcal{T}(tag_1) < \mathcal{T}(tag_2) \\ \vee \ (\mathcal{T}(tag_1) = \mathcal{T}(tag_2) \wedge \mathcal{M}(tag_1) < \mathcal{M}(tag_2))\}$$

and a relation $\preceq = \{(tag_1, tag_2) \in tags^2 \mid tag_1 \prec tag_2 \vee tag_1 = tag_2\}$.

We use R to denote the set of reactors and N the set of reactions defined in an LF program. In the axiomatic model, a reaction invocation is defined as $rxn \in rxns = N \cup \{\epsilon\}$, where ϵ 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 $val \in vals = V^{|S|}$, 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 $trig \in trigs = \mathbb{B}^{|T|}$. We further let $A \subseteq T$ denote the set of *actions* specifically. A boolean vector denoting whether actions are scheduled is defined as $sch \in schs = \mathbb{B}^{|A|}$. This is different than the same actions in *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 $pld \in plds = \mathbb{V}^{|A|}$.

A *path* $\bar{\pi} = (\pi_1, \pi_2, \ldots, \pi_k)$ 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 $\overline{indices}$ set as a sequence of natural numbers from 1 to the bound k, i.e., $\overline{indices} = (1, 2, \ldots, k)$. 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 $indices = \overline{indices} \cup (k{+}1, \ldots, k{+}|N|)$, $\pi = \bar{\pi} \cup (\pi_{k+1}, \ldots, \pi_{k+|N|}) \in \Pi$, and the length of the padding is set to $|N|$. Note that Safety MTL formulas are evaluated on $\bar{\pi}$ and not on π, the padded version, which is used here to prevent the axioms from being trivially unsatisfiable at the path boundary.

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

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

(3) 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:

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

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.

$$\forall i, j \in indices.\ \mathcal{G}(\pi_i) = \mathcal{G}(\pi_j) \wedge \mathfrak{P}(\mathcal{N}(\pi_i)) < \mathfrak{P}(\mathcal{N}(\pi_j)) \implies i < j, \tag{ReactionOrder}$$

where $\mathfrak{P} : N \to \mathbb{N}$ 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:

$$\forall i, j \in \mathit{indices}.\ (\mathcal{N}(\pi_i) = \mathcal{N}(\pi_j) \wedge i \neq j) \implies \mathcal{G}(\pi_i) \neq \mathcal{G}(\pi_j). \quad \text{(Uniqueness)}$$

Next, we place constraints on the time tags, where both the timestamp and the microstep need to be non-negative.

$$\forall i \in \mathit{indices}.\ \mathcal{T}(\mathcal{G}(\pi_i)) \geq 0 \wedge \mathcal{M}(\mathcal{G}(\pi_i)) \geq 0. \quad \text{(ValidTag)}$$

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

$$\forall j \in \mathit{indices}.\ \mathcal{N}(\pi_j) \neq \epsilon \implies (\forall i \in \mathit{indices}, i < j.\ \mathcal{N}(\pi_i) \neq \epsilon) \quad \text{(EmptyLast)}$$

(4) 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 $val_u, val_d \in \mathit{val}$ and their presence in $trig_u, trig_d \in \mathit{trig}$. We use a variable $\mathit{physical} \in \mathbb{B}$ to denote whether the connection is a *physical* connection, and we define a variable $\mathit{delay} \in \mathbb{N}$ to indicate the logical delay carried by the connection. We also define a *schedule* operator $\oplus : \mathit{tags} \times \mathbb{N} \to \mathit{tags}$ as $g \oplus t = g'$ s.t. $\mathcal{T}(g') = \mathcal{T}(g) + t$ and $\mathcal{M}(g') = 0$ if $t > 0$ and $\mathcal{M}(g') = \mathcal{M}(g) + 1$ if $t = 0$. The behavior of a connection is specified as follows.

$$\begin{aligned}\forall i \in \overline{\mathit{indices}}.\ (\mathit{trig}_u(\pi_i) \implies (\exists j \in \mathit{indices}.\ j > i \wedge \mathit{trig}_d(\pi_j) \\ \wedge\ \mathit{val}_d(\pi_j) = \mathit{val}_u(\pi_i) \wedge (\neg \mathit{physical} \implies \mathcal{G}(\pi_j) = \mathcal{G}(\pi_i) \oplus \mathit{delay}))) \quad \text{(SendToDownstream)}\end{aligned}$$

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.

$$\begin{aligned}\forall i \in \mathit{indices}.\ (\mathit{trig}_d(\pi_i) \implies (\exists j \in \mathit{indices}.\ j < i \wedge \mathit{trig}_u(\pi_j) \\ \wedge\ \mathit{val}_u(\pi_j) = \mathit{val}_d(\pi_i) \wedge (\neg \mathit{physical} \implies \mathcal{G}(\pi_i) = \mathcal{G}(\pi_j) \oplus \mathit{delay}))) \quad \text{(Correspondence)}\end{aligned}$$

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.

$$\forall i \in \mathit{indices}.\ \mathit{trig}_d(\pi_i) = \mathit{false} \implies (\mathit{val}_d(\pi_i) = \mathit{val}_d(\pi_{i-1})) \quad \text{(SameIfAbsent)}$$

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.

$$\exists i \in \mathit{indices}.\ \mathcal{N}(\pi_i) = v \wedge \mathcal{G}(\pi_i) = (0,0) \wedge \neg(\exists j \in \mathit{indices}.\ \mathcal{N}(\pi_j) = v \wedge j \neq i) \quad \text{(StartupTrigger)}$$

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 $\mathit{val}_a \in \mathit{vals}$, its presence stored in $\mathit{trig}_a \in \mathit{trigs}$, and a logical delay $\mathit{delay}_a \in \mathbb{N}$. We further let $\mathit{reactions}_a$ denote a

set of reactions that can schedule the action a. We can state an axiom, ScheduleAction, to encode the semantics of actions.

$$\exists i \in indices. \bigvee_{n \in reactions_a} trig_a(\pi_i) = true \implies (\exists j \in indices.\ j < i \ \wedge \ \mathcal{N}(\pi_j) = n$$
$$\wedge\ \mathcal{G}(\pi_i) = \mathcal{G}(\pi_j) \oplus delay_a \wedge sch_a(\pi_j) = true \wedge val_a(\pi_i) = pld_a(\pi_j)) \quad \text{(ScheduleAction)}$$

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 $trig_t \in trigs$. Recall also that k is the largest index in $indices$. Then, a timer must start firing at the pre-defined offset, as stated in TimerOffset.

$$((\mathcal{T}(\mathcal{G}(\pi_k)) \geq offset) \implies \exists i \in \overline{indices}.\ trig_t(\pi_i) = true \wedge \mathcal{G}(\pi_i) = (offset, 0))$$
$$\wedge\ ((\mathcal{T}(\mathcal{G}(\pi_k)) < offset) \implies \forall j \in \overline{indices}.\ \mathcal{N}(\pi_j) \neq \epsilon) \quad \text{(TimerOffset)}$$

Intuitively, if the last state of $\bar{\pi}$ (i.e., π_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, TimerPeriod and TimerSpacing to encode this behavior.

$$\forall i \in \overline{indices}.\ trig_t(\pi_i) = true \implies$$
$$(\exists j \in indices.\ j > i \wedge trig_t(\pi_j) = true \wedge \mathcal{G}(\pi_j) = \mathcal{G}(\pi_i) \oplus period) \quad \text{(TimerPeriod)}$$

$$\forall i \in \overline{indices}.\ trig_t(\pi_i) = true \implies$$
$$((\exists x \in \mathbb{N}.\ x \geq 0 \wedge \mathcal{T}(\mathcal{G}(\pi_i)) = offset + x \times period) \wedge \mathcal{M}(\mathcal{G}(\pi_i)) = 0) \quad \text{(TimerSpacing)}$$

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

$$\forall i \in indices.\ (\exists t \in triggers_n.\ t = true) \iff \mathcal{N}(\pi_i) = n \quad \text{(ReactionTriggers)}$$

(5) 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 Section 7.

5 BOUNDING THE COMPLETENESS THRESHOLD FOR BMC

Recall that the *completeness threshold*, denoted as $\mathcal{CT}$, is an attribute of a system M with respect to a property p and a translation scheme such that checking a trace up to $\mathcal{CT}$ 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 (Section 4) and the translation of a Safety MTL formula to a First-Order Logic formula over a timed word $\bar{\pi}$ based on the semantics of MTL (Definition 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 $\hat{\mathcal{CT}}$. 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 $\hat{\mathcal{CT}}$ 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 Figure 4.

Formally, a state space diagram is a directed graph (V, E) where $V \subseteq tags \times \mathcal{P}(rxns) \times \mathcal{P}(events)$ and $E \subseteq V \times V$, where $\mathcal{P}$ denotes a power set. The symbols *tags* and *rxns* are defined previously in Figure 3. Here we introduce a new set *events*, which is defined as $events = tags \times trigs$, where *trigs* is also defined in Figure 3. To extract a tag, a set of reactions, or a set of events from a node, we define projection functions $tag_n : V \to \mathcal{P}(events)$, $rxns : V \to \mathcal{P}(rxns)$, and $events : V \to \mathcal{P}(events)$. Similarly, to extract a tag or a trigger from an event, we define projection functions $tag_e : events \to \mathcal{P}(events)$ and $trigs : events \to \mathcal{P}(rxns)$.

Constructing a state space diagram requires a light-weight simulation of the *worst-case* execution until some time horizon, denoted as $horizon \in \mathbb{N}$ is reached, which is the time interval specified by a Safety MTL formula. We provide a precise definition of a theoretical worst-case execution below.

Definition 5.1 (Theoretical Worst-case Execution of an LF Program). A theoretical worst-case execution of an LF program is a execution path π permitted by the LF program structure such that

$$\forall \pi' \in \Pi.\ \forall g \in tags.\ |\{\pi_i \in \pi \mid \mathcal{G}(\pi_i) = g \wedge \mathcal{N}(\pi_i) \neq \epsilon\}| \geq |\{\pi'_j \in \pi' \mid \mathcal{G}(\pi'_j) = g \wedge \mathcal{N}(\pi'_j) \neq \epsilon\}|$$

Intuitively, the theoretical worst-case execution path has the most possible number of reaction invocations at each logical tag among all possible execution paths defined by the LF program structure.

During the simulation, the algorithm keeps track of three quantities (as defined earlier): a current time tag (denoted as *currentTag*), a list of reactions invoked at the current tag (denoted as *reactionsInvoked*), and a priority queue of pending events ordered by time tags (denoted as *pendingEvents*). The algorithm is listed in Algorithm 1 with specific implementation details abstracted away.

The main exploration logic happens inside a while loop 5. Before the actual exploration begins, the algorithm requires a *horizon* argument and identifying a set of *initial events* from which the execution unfolds. The *horizon* argument is derived based on the given MTL formula as follows. Let ψ be an MTL formula in Negation Normal Form, and let $h : \text{MTL} \to \mathbb{R}$ be a function mapping an MTL formula to its horizon. The function h can then be defined recursively.

$$h(\psi) = \begin{cases} \max(h(\psi_1), h(\psi_2)) & \text{if } \psi \equiv \psi_1 \wedge \psi_2 \text{ or } \psi \equiv \psi_1 \vee \psi_2 \\ \max(I) + h(\psi_1) & \text{if } \psi \equiv \mathbf{G}_I\psi_1 \text{ or } \psi \equiv \mathbf{F}_I\psi_1 \text{ or } \psi \equiv \mathbf{X}_I\psi_1 \\ \max(I) + \max(h(\psi_1), h(\psi_2)) & \text{if } \psi \equiv \psi_1 \mathbf{U}_I \psi_2 \text{ or } \psi \equiv \psi_1 \widetilde{\mathbf{U}}_I \psi_2 \\ 0 & \text{otherwise} \end{cases}$$

To determine the set of initial events, the algorithm identifies 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., $|pendingEvents| = 0$);
(2) the simulation completes the time horizon (i.e., $\mathcal{T}(currentTag) > horizon$);
(3) a *cycle* is found in the state space diagram.

ALGORITHM 1: Explore the state space and build a state space diagram

```
1:  procedure EXPLORE(horizon, findLoop)
2:      previousTag, currentTag ← ϵ, 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
```

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.

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. □

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 Figure 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

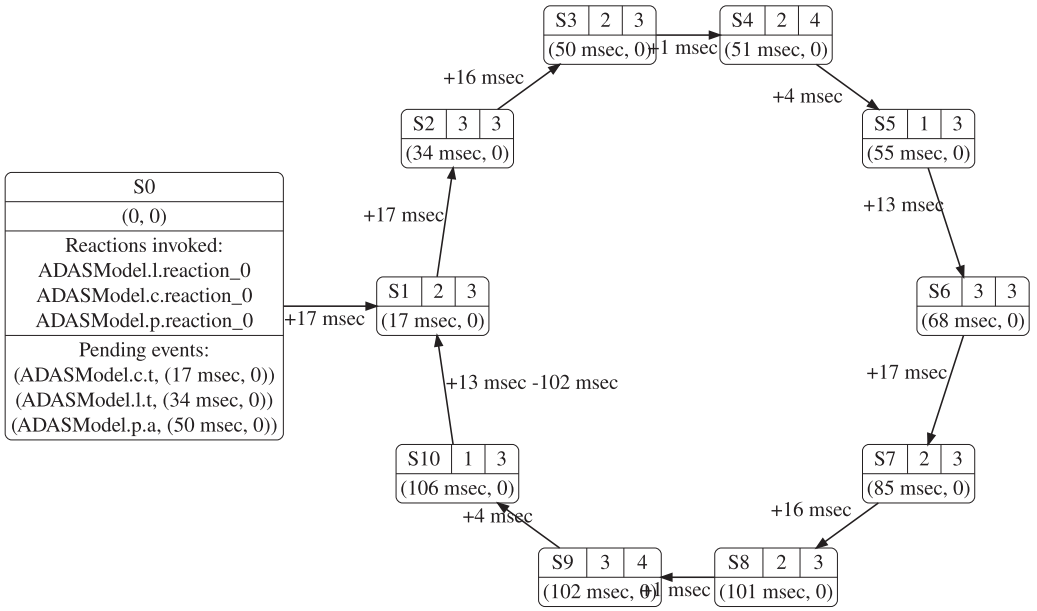

Fig. 4. An abstract view of the state space diagram for the running example in Figure 1. In this abstract view, except for S0, which is shown in full, each node consists of (from left to right) an identifier, the number of reactions invoked at the current tag, the number of events pending, and (at the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

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

(1) $rxns(n_1) = rxns(n_2)$;
(2) $\{trigs(e_1) \subseteq trigs \mid e_1 \in events(n_1)\} = \{trigs(e_2) \subseteq trigs \mid e_2 \in events(n_2)\}$;
(3) $\{tag_e(e_1) - tag_n(n_1) \mid e_1 \in events(n_1)\} =$
$\{tag_e(e_2) - tag_n(n_2) \mid e_2 \in events(n_2)\}$.

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

From diagram to $\hat{CT}$. Once the state space diagram is generated, we can then compute $\hat{CT}$ from it. For this, we state two theorems.

Theorem 5.3 ($\hat{CT}$ 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 $\hat{CT}$ such that*

$$\hat{CT} = \sum_{n \in V} \mid rxns(n) \mid.$$

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., $\hat{CT}$, can be computed by summing up the number of reaction invocations of each node. We then obtain the desired expression. □

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

Theorem 5.4 ($\hat{CT}$ 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 $\hat{CT}$ such that*

$$\hat{CT} = \sum_{n'' \in V'' \cup \{n_c\}} |\, rxns(n'')\,| \; + \sum_{n' \in V'} |\, rxns(n')\,| * iterations,$$

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, $iterations = \lceil (horizon - \mathcal{T}(tag_n(n_c)))/\rho \rceil$, and ρ 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 $\hat{CT}$ 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., $horizon - \mathcal{T}(tag_n(n_c))$, dividing it by the period of the cycle, ρ, 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.[1] 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 Figure 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: (1) 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 Figure 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).

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 Figure 2 as "C Parser" and "MTL Parser"), generating a parse tree for C and another parse tree for MTL.

[1] GitHub link: https://github.com/lf-lang/lingua-franca

$$
\begin{aligned}
\mathit{mtl} &::= \mathit{formula} \mid \mathit{unaryOp}\ \mathit{formula} \mid \\
&\quad\ \mathit{formula}\ \mathit{binaryOp}\ \mathit{formula} \\
\mathit{unaryOp} &::= \texttt{`!'} \mid \texttt{`X'}\ \mathit{interval} \mid \\
&\quad\ \texttt{`G'}\ \mathit{interval} \mid \texttt{`F'}\ \mathit{interval} \\
\mathit{binaryOp} &::= \texttt{`U'}\ \mathit{interval} \\
\mathit{interval} &::= (\texttt{`('} \mid \texttt{`['})\ \mathit{time}\ \texttt{`,'}\ \mathit{time}\ (\texttt{`)'} \mid \texttt{`]'}) \mid \\
&\quad\ \texttt{`['}\ \mathit{time}\ \texttt{`]'} \\
\mathit{time} &::= \text{int (id)?} \\
\mathit{formula} &::= \mathit{prop} \mid \mathit{equivalence} \mid \mathit{implication} \mid \\
&\quad\ \mathit{disjunction} \mid \mathit{conjunction} \mid \mathit{mtl}
\end{aligned}
\qquad
\begin{aligned}
\mathit{prop} &::= \texttt{`true'} \mid \texttt{`false'} \mid \\
&\quad\ \mathit{expr}\ \mathit{relOp}\ \mathit{expr} \\
\mathit{expr} &::= \text{id} \mid \texttt{`('}\ \mathit{sum}\ \texttt{`)'} \mid \text{int} \mid \\
&\quad\ \mathit{expr}\ \mathit{arithOp}\ \mathit{expr} \\
\mathit{relOp} &::= \texttt{`=='} \mid \texttt{`!='} \mid \texttt{`<'} \mid \\
&\quad\ \texttt{`<='} \mid \texttt{`>'} \mid \texttt{`>='} \\
\mathit{arithOp} &::= \texttt{`+'} \mid \texttt{`-'} \mid \texttt{`*'} \mid \texttt{`/'} \\
\mathit{equivalence} &::= \mathit{mtl}\ \texttt{`<==>'}\ \mathit{mtl} \\
\mathit{implication} &::= \mathit{mtl}\ \texttt{`==>'}\ \mathit{mtl} \\
\mathit{disjunction} &::= \mathit{mtl}\ \texttt{`||'}\ \mathit{mtl} \\
\mathit{conjunction} &::= \mathit{mtl}\ \texttt{`\&\&'}\ \mathit{mtl}
\end{aligned}
$$

Fig. 5. Syntax for the Safety MTL fragment supported by the LF Verifier.

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 implemention 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 Section 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 (Figure 1) effectively turn into

```
1 /* (C) Reaction body 1 */
2 if (true) lf_schedule(a, 0);
3 if (true) self->requestStop = 1;
4 /* (C) Reaction body 2 */
5 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 $C\hat{\mathcal{T}}$. 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 "$C\hat{\mathcal{T}}$ Calculator" which executes Algorithm 1 in Section 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 ms]` operator requires 10 ms while the inner `F[0, 55 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 (Figure 4) shows that within an interval of 65 ms, the program reaches six states (S0-S5). $C\hat{\mathcal{T}}$, i.e., the maximum sum of reaction invocations along the six states, is $C\hat{\mathcal{T}} = 3 + 2 + 3 + 2 + 2 + 1 = 13$. The "$C\hat{\mathcal{T}}$ 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 Section 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].

```
/* (Uclid5) Preliminary definitions */
const START     : integer = 0;    // The start index of the trace
const END       : integer = 13;   // The end index of the trace (ie, CT bound)
const END_TRACE : integer = 19;   // The end index of the trace with padding
type  state_t = {rxn_t, tag_t, val_t, trig_t, sch_t, pld_t};    // A state is a 6-tuple.
type  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 ==> (
     ((finite_exists (j : integer) in indices :: (j > i && j <= END_TRACE) && (
       ADASModel_p_a_is_present(t(j))
       && tag_same(g(j), tag_schedule(g(i), (50000000+0)))
       && ADASModel_p_a(s(j)) == 0
     )) // Closes finite_exists
     && ADASModel_p_a_scheduled(d(i)))))
     && (true ==> ((ADASModel_p_requestStop(s(i)) == 1)
))))));

/* (Uclid5) Axioms of reaction 2 of ADASProcessor */
axiom(finite_forall (i : integer) in indices :: (i > START && i <= END) ==> (
 (ADASModel_p_reaction_1(rxn(i))) ==> (((
   (ADASModel_p_requestStop(s(i)) == 1) ==> (((ADASModel_p_out1(s(i)) == 1) && (
        ADASModel_p_out1_is_present(t(i)))))
)))));
```

As mentioned in Section 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))
  ))))))))));

/* (Uclid5) BMC: given an initial condition, the property needs to hold at step 0. */
property bmc_responsive : initial_condition() ==> p(0);
```

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 Section 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`.

```
73 // l.out  -> v.in1                // Originally connected
74 c.out  -> v.in2
75 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]. Thirdteen out of the 22 are safe examples and 9 are unsafe examples. The benchmark focuses on verifying

the three types of properties introduced in Section 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.[2]

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 implemention 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 Section 8.

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. Figure 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.

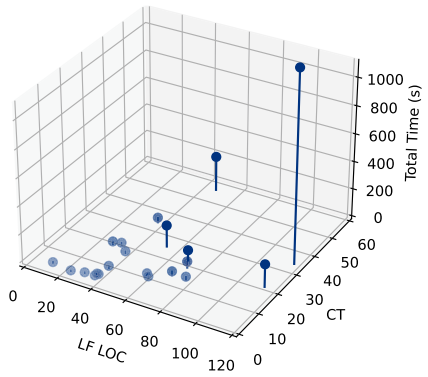

Fig. 6. Distribution of benchmark results.

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].

[2]GitHub link: https://github.com/lf-lang/lf-verifier-benchmarks

Table 1. Benchmark Results

Program	Type	#Rxns	$\hat{CT}$	Safe?	LOC		TIME (seconds)			
					LF	Uclid5	GenUclid	GenSMT	Z3	Total
ADASModel [75]	III	6	13	yes	78	875	3.35	17.72	1.72	22.79
AircraftDoor [53]	III	3	3	yes	37	402	3.59	2.98	0.02	6.59
Alarm [33]	III	2	2	no	24	338	2.93	2.43	0.01	5.37
CoopSchedule [24]	I	2	18	no	29	659	2.75	18.35	0.31	21.41
Elevator [16]	III	13	33	yes	118	1899	3.09	322.05	1038.02	1363.16
Election [17]	I	9	18	yes	73	1045	3.47	38.18	3.27	44.92
Election2 [17]	I	6	9	no	61	700	3.55	8.66	0.17	12.38
Factorial [16]	III	3	33	yes	38	461	3.73	17.73	13.15	34.61
Fibonacci [16]	III	5	55	yes	48	649	2.81	92.15	153.49	248.45
PingPong [36]	I	4	16	no	39	510	3.69	9.73	0.14	13.56
Pipe [22, 40]	I	5	24	no	54	2059	2.89	139.52	12.15	154.56
ProcessMsg [24]	I	4	19	yes	33	487	3.75	9.48	0.74	13.97
ProcessSync [24]	I	1	3	yes	12	284	2.78	2.21	0.01	5.00
Railroad [2]	I	10	20	yes	115	2333	2.93	155.21	4.33	162.47
Ring [22]	II	4	8	yes	39	897	3.68	12.15	0.07	15.9
RoadsideUnit [28]	I	8	21	yes	70	1536	3.78	70.29	0.62	74.69
SafeSend [22]	II	4	4	yes	37	479	2.70	3.78	0.03	6.51
Subway [33]	II	10	13	no	70	1097	2.85	27.11	0.31	30.27
Thermostat [49]	I	6	10	yes	59	652	2.80	10.61	0.14	13.55
TrafficLight† [16]	I	4	14	–	69	–	–	–	–	–
TrainDoor [71]	II	3	3	no	31	398	2.62	3.03	0.01	5.66
UnsafeSend [22]	II	4	4	no	38	478	2.84	3.66	0.02	6.52

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

Furthermore, we hope to support other LF features including physical actions, deadlines, modal models, etc. We leave these improvements for future work.

REFERENCES

[1] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 36, 2 (2014), 1–74.

[2] Rajeev Alur. 2015. *Principles of Cyber-physical Systems.* MIT press.

[3] Charles André. 2009. *Syntax and Semantics of the Clock Constraint Specification Language (CCSL).* Research Report RR-6925. INRIA. 37 pages. https://hal.inria.fr/inria-00384077

[4] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. 2009. Bounded model checking of software using SMT solvers instead of SAT solvers. *International Journal on Software Tools for Technology Transfer* 11, 1 (2009), 69–83.

[5] Roy Armoni, Limor Fix, Ranan Fraer, Scott Huddleston, Nir Piterman, and Moshe Y. Vardi. 2005. SAT-based Induction for Temporal Safety Properties. *Electronic Notes in Theoretical Computer Science* 119, 2 (2005), 3–16. https://doi.org/10.1016/j.entcs.2004.12.021

[6] Kyungmin Bae, Peter Csaba Ölveczky, Thomas Huining Feng, and Stavros Tripakis. 2009. Verifying ptolemy II discrete-event models using real-time maude. In *Formal Methods and Software Engineering*, Karin Breitman and Ana Cavalcanti (Eds.). Springer, Berlin, 717–736.

[7] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. 2021. Satisfiability modulo theories. In *Handbook of Satisfiability* (2nd ed.), Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh (Eds.). IOS Press, Chapter 33, 1267–1329.

[8] Soroush Bateni, Marten Lohstroh, Hou Seng Wong, Rohan Tabish, Hokeun Kim, Shaokai Lin, Christian Menard, Cong Liu, and Edward A. Lee. 2022. Xronos: Predictable coordination for safety-critical distributed embedded systems. *arXiv preprint arXiv:2207.09555* (2022).

[9] Albert Benveniste and Gérard Berry. 1991. The synchronous approach to reactive and real-time systems. *Proc. IEEE* 79, 9 (1991), 1270–1282.

[10] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. 2013. *Systems and Software Verification: Model-checking Techniques and Tools.* Springer Science & Business Media.

[11] Lorenzo Bettini. 2016. *Implementing Domain-specific Languages with Xtext and Xtend*. Packt Publishing Ltd.
[12] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. 1999. Symbolic model checking without BDDs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 193–207.
[13] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. 2009. Bounded model checking. *Handbook of Satisfiability* 185, 99 (2009), 457–481.
[14] Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. 2002. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions *(LNCS 2404)*, E. Brinksma and K. G. Larsen (Eds.). 78–92.
[15] Christos G. Cassandras and Stéphane Lafortune. 2008. *Introduction to Discrete Event Systems*. Springer.
[16] Adrien Champion, Alain Mebsout, Christoph Sticksel, and Cesare Tinelli. 2016. The kind 2 model checker. In *International Conference on Computer Aided Verification*. Springer, 510–517.
[17] Ernest Chang and Rosemary Roberts. 1979. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Commun. ACM* 22, 5 (1979), 281–283.
[18] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.
[19] Julien Deantoni, João Cambeiro, Soroush Bateni, Shaokai Lin, and Marten Lohstroh. 2021. Debugging and verification tools for linga franca in GEMOC studio. In *2021 Forum on Specification & Design Languages (FDL)*. IEEE, 01–08.
[20] Julien Deantoni, Papa Issa Diallo, Joël Champeau, Benoit Combemale, and Ciprian Teodorov. 2014. *Operational Semantics of the Model of Concurrency and Communication Language*. Research Report RR-8584. INRIA. 23 pages. https://hal.inria.fr/hal-01060601
[21] Jack B. Dennis. 1974. *First Version Data Flow Procedure Language*. Report MAC TM61. MIT Laboratory for Computer Science.
[22] Emanuele D'Osualdo, Jonathan Kochems, and C.-H. Luke Ong. 2013. Automatic verification of erlang-style concurrency. In *International Static Analysis Symposium*. Springer, 454–476.
[23] Javier Esparza, Ruslán Ledesma-Garza, Rupak Majumdar, Philipp Meyer, and Filip Niksic. 2014. An SMT-based approach to coverability analysis. In *International Conference on Computer Aided Verification*. Springer, 603–619.
[24] Inc. Express Logic. [n. d.]. Measuring real-time performance of an RTOS.
[25] Moritz Eysholdt and Heiko Behrens. 2010. Xtext: Implement your language faster than the quick and dirty way. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications*. ACM, 307–309. https://doi.org/10.1145/1869542.1869625
[26] Miguel Felder, Dino Mandrioli, and Angelo Morzenti. 1994. Proving properties of real-time systems through logical specifications and petri net models. *IEEE Transactions on Software Engineering* 20, 2 (1994), 127–141.
[27] Anders Franzén. 2006. Using satisfiability modulo theories for inductive verification of lustre programs. *Electronic Notes in Theoretical Computer Science* 144, 1 (2006), 19–33.
[28] Brian Gallagher, Hidlehiko Akatsuka, and Hideaki Suzuki. 2006. Wireless communications for vehicle safety: Radio link performance and wireless connectivity methods. *IEEE Vehicular Technology Magazine* 1, 4 (2006), 4–24.
[29] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. 2013. CADP 2011: A toolbox for the construction and analysis of distributed processes. *International Journal on Software Tools for Technology Transfer* 15, 2 (2013), 89–107.
[30] A. H. Ghamarian, M. C. W. Geilen, T. Basten, and S. Stuijk. 2006. Throughput analysis of synchronous data flow graphs. *Proceedings - Design, Automation and Test in Europe (DATE'06)*. 116–121. https://doi.org/10.1109/DATE.2008.4484672
[31] Adwait Godbole, Yatin A. Manerkar, and Sanjit A. Seshia. 2022. Automated conversion of axiomatic to operational models: Theory and practice. In *Conference on Formal Methods in Computer-Aided Design–Fmcad 2022*. 331.
[32] George Hagen and Cesare Tinelli. 2008. Scaling up the formal verification of Lustre programs with SMT-based techniques. In *2008 Formal Methods in Computer-Aided Design*. IEEE, 1–9.
[33] Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel. 1992. Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *IEEE Transactions on Software Engineering* 18, 9 (1992), 785–793.
[34] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. 2001. Giotto: A time-triggered language for embedded programming. In *EMSOFT 2001*, Vol. LNCS 2211. Springer-Verlag, 166–184.
[35] Carl Hewitt, Peter Boehler Bishop, and Richard Steiger. 1973. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*. Standford, CA, USA, August 20-23, 1973. 235–245.
[36] Shams M. Imam and Vivek Sarkar. 2014. Savina-an actor benchmark suite: Enabling empirical evaluation of actor libraries. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control*. 67–80.

[37] Mohammad Mahdi Jaghoori, Ali Movaghar, and Marjan Sirjani. 2006. Modere: The model-checking engine of rebeca. *Journal of Computer and System Sciences - JCSS* 2, 1810–1815. https://doi.org/10.1145/1141277.1141704
[38] Jeff C. Jensen, Edward A. Lee, and Sanjit A. Seshia. 2012. An introductory lab in embedded and cyber-physical systems. *LeeSeshia.org, Berkeley, CA* (2012).
[39] Gilles Kahn. 1974. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*. North-Holland Publishing Co., 471–475.
[40] Naoki Kobayashi, Motoki Nakade, and Akinori Yonezawa. 1995. Static analysis of communication for asynchronous concurrent programming languages. In *Static Analysis*, Alan Mycroft (Ed.). Springer, Berlin, 225–242.
[41] Ron Koymans. 1990. Specifying real-time properties with metric temporal logic. *Real-time Systems* 2, 4 (1990), 255–299.
[42] Daniel Kroening and Ofer Strichman. 2003. Efficient computation of recurrence diameters. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 298–309.
[43] Daniel Kroening, Ofer Strichman, Thomas Wahl, and James Worrell. 2011. Linear completeness thresholds for bounded model checking. *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)*. 557–572.
[44] Daniel Kroening and Michael Tautschnig. 2014. CBMC–C bounded model checker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 389–391.
[45] Steven Lauterburg, Mirco Dotta, Darko Marinov, and Gul Agha. 2009. A framework for state-space exploration of Java-based actor programs. In *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 468–479.
[46] Edward Lee and David Messerschmitt. 1987. Synchronous data flow. 75, 9 (1987), 1235–1245.
[47] Edward A. Lee. 2008. Cyber physical systems: Design challenges. In *2008 11th IEEE International Symposium on Object and Component-oriented Real-time Distributed Computing (ISORC)*. IEEE, 363–369.
[48] Edward A. Lee, Soroush Bateni, Shaokai Lin, Marten Lohstroh, and Christian Menard. 2021. Quantifying and generalizing the CAP theorem. *arXiv:2109.07771 [cs.DC]* (September 16 2021). https://arxiv.org/abs/2109.07771
[49] Edward Ashford Lee and Sanjit Arunkumar Seshia. 2016. *Introduction to Embedded Systems: A Cyber-physical Systems Approach*. Mit Press.
[50] Marten Lohstroh, Iñigo Incer Romeo, Andrés Goens, Patricia Derler, Jeronimo Castrillon, Edward A. Lee, and Alberto Sangiovanni-Vincentelli. 2019. Reactors: A deterministic model for composable reactive systems. *Model-Based Design of Cyber Physical Systems (CyPhy'19)* (2019).
[51] Marten Lohstroh, Christian Menard, Soroush Bateni, and Edward A. Lee. 2021. Toward a lingua franca for deterministic concurrent systems. *ACM Transactions on Embedded Computing Systems (TECS), Special Issue on FDL'19* 20, 4 (May 2021), Article 36. https://doi.org/10.1145/3448128
[52] Marten Lohstroh, Christian Menard, Soroush Bateni, and Edward A. Lee. 2021. Toward a lingua franca for deterministic concurrent systems. *ACM Trans. Embed. Comput. Syst.* 20, 4, Article 36 (may 2021), 27 pages. https://doi.org/10.1145/3448128
[53] Marten Lohstroh, Christian Menard, Alexander Schulz-Rosengarten, Matthew Weber, Jeronimo Castrillon, and Edward A. Lee. 2020. A language for deterministic coordination across multiple timelines. In *Forum for Specification and Design Languages (FDL)*. IEEE. https://doi.org/10.1109/FDL50818.2020.9232939
[54] Daniel Lustig, Geet Sethi, Margaret Martonosi, and Abhishek Bhattacharjee. 2016. COATCheck: Verifying memory ordering at the hardware-OS interface. *SIGARCH Comput. Archit. News* 44, 2 (mar 2016), 233–247. https://doi.org/10.1145/2980024.2872399
[55] Zohar Manna and Amir Pnueli. 1993. Verifying hybrid systems. In *Hybrid Systems*, Vol. LNCS 736. 4–35.
[56] Christian Menard, Andrés Goens, Marten Lohstroh, and Jeronimo Castrillon. 2020. Achieving determinism in adaptive AUTOSAR. In *Proceedings of the 2020 Design, Automation and Test in Europe Conference (DATE)* (Grenoble, France) *(DATE'20)*. EDA Consortium.
[57] Aleksandar Milicevic and Hillel Kugler. 2011. Model checking using SMT and theory of lists. In *NASA Formal Methods Symposium (NFM)*. Springer, 282–297.
[58] Walid A. Najjar, Edward A. Lee, and Guang R. Gao. 1999. Advances in the dataflow computational model. *Parallel Comput.* 25, 13-14 (December 1999), 1907–1929.
[59] Joël Ouaknine and James Worrell. 2006. Safety metric temporal logic is fully decidable. In *Tools and Algorithms for the Construction and Analysis of Systems*, Holger Hermanns and Jens Palsberg (Eds.). Springer, Berlin, 411–425.
[60] Joel Ouaknine and James Worrell. 2007. On the decidability and complexity of metric temporal logic over finite words. *Logical Methods in Computer Science* 3, 1 (feb 2007). https://doi.org/10.2168/lmcs-3(1:8)2007
[61] Joël Ouaknine and James Worrell. 2007. On the decidability and complexity of metric temporal logic over finite words. *arXiv preprint cs/0702120* (2007).
[62] Joël Ouaknine and James Worrell. 2008. Some recent results in metric temporal logic. In *International Conference on Formal Modeling and Analysis of Timed Systems*. Springer, 1–13.
[63] Terence Parr. 2013. The definitive ANTLR 4 reference. *The Definitive ANTLR 4 Reference* (2013), 1–326.

[64] Elizabeth Polgreen, Kevin Cheang, Pranav Gaddamadugu, Adwait Godbole, Kevin Laeufer, Shaokai Lin, Yatin A. Manerkar, Federico Mora, and Sanjit A. Seshia. 2022. UCLID5: Multi-modal formal modeling, verification, and synthesis. In *International Conference on Computer Aided Verification*. Springer, 538–551.

[65] Matteo Pradella, Angelo Morzenti, and Pierluigi San Pietro. 2013. Bounded satisfiability checking of metric temporal logic specifications. *ACM Trans. Softw. Eng. Methodol.* 22, 3, Article 20 (jul 2013), 54 pages. https://doi.org/10.1145/2491509.2491514

[66] Ragunathan Rajkumar, Insup Lee, Lui Sha, and John Stankovic. 2010. Cyber-physical systems: The next computing revolution. In *Design Automation Conference*. IEEE, 731–736.

[67] Arni Hermann Reynisson, Marjan Sirjani, Luca Aceto, Matteo Cimini, Ali Jafari, Anna Ingólfsdóttir, and Steinar Hugi Sigurdarson. 2014. Modelling and simulation of asynchronous real-time systems using timed rebeca. *Science of Computer Programming* 89 (2014), 41–68.

[68] Alceste Scalas, Nobuko Yoshida, and Elias Benussi. 2019. Effpi: Verified message-passing programs in Dotty. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Scala*. 27–31.

[69] Alexander Schulz-Rosengarten, Reinhard von Hanxleden, Marten Lohstroh, Soroush Bateni, and Edward A. Lee. 2023. Modal reactors. *arXiv preprint arXiv:2301.09597* (2023).

[70] Sanjit A. Seshia and Pramod Subramanyan. 2018. UCLID5: Integrating modeling, verification, synthesis, and learning. In *Proceedings of the 15th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*.

[71] Marjan Sirjani, Edward A. Lee, and Ehsan Khamespanah. 2020. Verification of cyberphysical systems. *Mathematics* 8, 7 (2020). https://doi.org/10.3390/math8071068

[72] Pramod Subramanyan and Sanjit A. Seshia. 2021. Getting started with Uclid5.

[73] Jonatan Wiik and Pontus Boström. 2017. Specification and automated verification of dynamic dataflow networks. In *International Conference on Software Engineering and Formal Methods*. Springer, 136–151.

[74] Shohei Yasutake and Takuo Watanabe. 2015. Actario: A framework for reasoning about actor systems. In *Workshop on Programming based on Actors, Agents, and Decentralized Control (AGERE)*.

[75] Adam Ziebinski, Rafal Cupek, Damian Grzechca, and Lukas Chruszczyk. 2017. Review of advanced driver assistance systems (ADAS). *AIP Conference Proceedings* 1906, 1 (2017), 120002. https://doi.org/10.1063/1.5012394 arXiv:https://aip.scitation.org/doi/pdf/10.1063/1.5012394

Received 23 March 2023; revised 2 June 2023; accepted 13 July 2023