Efficient Verification of Lingua Franca Programs

ANONYMOUS AUTHOR(S)

In this paper we present a rewriting logic semantics for the LINGUA FRANCA (LF) coordination language for concurrent cyber-physical systems developed at UC Berkeley. Our semantics is executable and therefore enables automatic formal analysis with the Maude and Real-Time Maude tools. In contrast to other verification approaches to LF, we capture the intended big-step "discrete-event" semantics of LF. Not only does this give us the desired semantics and therefore the correct analysis results, but our semantics also provides significantly faster verification than those based on an interleaving "event-based" semantics. We capture larger subsets of LF than previous verification approaches, and provide all analysis methods provided by them, including "mixed" (data-flow and state) properties, and LTL and timed CTL model checking. Benchmarking on the LF VERIFIER benchmark suite shows that our analyses drastically outperform those of LF VERIFIER.

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

Additional Key Words and Phrases: Cyber-physical systems, formal executable semantics, simulation, reachability analysis, LTL model checking, timed CTL model checking, rewriting logic, Maude, Lingua Franca

1 Introduction

LINGUA FRANCA (LF) [9, 11, 13] is a coordination language for cyber-physical systems, developed by the group of Edward A. Lee at UC Berkeley, that provides *deterministic concurrency*: the only source of nondeterminism are "external actions." LF eliminates race conditions by construction, makes it easy to specify timed behavior, and removes the need to perform manual synchronization [9].

In LF, *reactions* execute programs when triggered. Reaction executions are considered logically instantaneous, so that many reactions can be executed at the same logical time. For example, one reaction's output may trigger another reaction's execution *at the same logical time*. LF has a timed *synchronous* (or "discrete-event") semantics ([11, pp. 7 and 20], [8, p. 4], [13, pp. 5, 20, and 24]): *all* reactions executed at the same logical time can be seen to happen in *one* step.

The goal of our paper is to formalize this intended semantics of LF and to support automatic formal analysis of LF programs based on this semantics; these are known to be challenging tasks: *"formally modeling the behavior of [discrete-event systems] using a conventional operational model [...] is difficult because one needs to specify properties globally over execution traces, a task that cannot be easily achieved at the transition level"* [8, p. 2].

We are aware of three efforts providing formal semantics and verification for LF; they all fall short of capturing its discrete-event semantics or covering realistic programs:

(1) Sirjani, Lee, and Khamespanah [20] translate three LF programs into an extension of the actor-based language Timed Rebeca [1], and perform reachability analysis on the models.
(2) In [8], the UC Berkeley group encodes *bounded* model checking of a subset of LF as an SMT problem. Their LF VERIFIER tool can model check a fragment of metric temporal logic, so that it is sufficient to consider behaviors up to time $h(\phi)$ (essentially the sum of the upper bounds of the temporal operator intervals), by computing an upper bound on the number of reaction executions that may happen within time $h(\phi)$, and performing SMT-based model checking up to so many steps.
(3) In [12], Marin et al. provide a rewriting logic [14] semantics and Maude [5] model checking for LF models.

Efforts (1) and (2) provide an "event-based" semantics, where each reaction execution corresponds to a step ("In the proposed axiomatic semantics, a *reaction invocation* defines a transition, which matches the *event-based semantics* in Sirjani et al." [8, p. 8]). These efforts do not cover external events ("physical actions"), so they model check what LF guarantees are deterministic models. Not only do they fail to capture the intended discrete-event semantics of LF, but, since they cover all interleavings in a "big" (synchronous) step, their verification is very slow.

Effort (3) aims at capturing the discrete-event semantics of LF, but does not support: (i) reactions with multiple triggers (which is a significant restriction), (ii) analyzing metric/timed temporal logic properties, which are the ones model checked in [8], or (iii) *dataflow* (or "action") properties, which are central in [8]. For all these reasons, Marin et al. [12] cannot cover most systems and/or desired properties in the LF Verifier benchmark suite.

In this paper we build on the work in [12] to provide the desired discrete-event semantics for a larger subset of LF than supported by other verification efforts, including *both* nondeterministic physical actions and multiple reaction triggers, and to provide many more analysis methods than other such efforts. We support both unbounded and time-bounded simulation, reachability analysis, LTL model checking, and timed CTL model checking, where the basic properties can be state properties, dataflow properties, and "mixed" properties. By cleverly handling the tags in the event queues, the reachable state spaces is our executions in many cases remain finite, so that we have terminating *unbounded* reachability analysis and LTL and timed CTL model checking (see Section 4).

In contrast to [12], we have also extended the LF compiler to (i) automatically generate a Maude model from an LF model, and (ii) to automatically perform the Maude analysis of the LF model, annotated with intuitive properties to verify, so that the LF user can analyze her model without any knowledge of Maude.

We give background to LF and Maude in Section 2, and formalize the semantics of LF in Section 3. Section 4 shows how we can perform a wide range of formal analyses of LF programs in Maude and Real-Time Maude. Since we now support all LF Verifier benchmarks, and their desired properties, Section 5 benchmarks all those 22 LF programs. As expected, our analyses drastically outperform those in [8]: they take less than 0.1 seconds, whereas the analysis in [8] can take more than 20 minutes. We introduce our LF-mc tool for automatically verifying LF models in Maude in Section 6, and discuss other related work in Section 7, and give some concluding remarks in Section 8.

2 Preliminaries

2.1 Lingua Franca

Lingua Franca (LF) [9, 11, 13] is a coordination language for cyber-physical systems that supports *reactor-oriented* programming [10]: the components of a system are modeled by *reactors*, which may have state variables, ports, actions, timers, and an ordered list of *reactions* which are invoked in response to a *trigger*. Such a trigger can be (the presence of an event at) an input port, a timer, or an action. A timer is used to generate periodic events, whereas a logical action is used to schedule future events with a time delay specified by the program. *Physical actions* represent external events from the physical environment, the timing and values of which are not controlled by the program.

A *reaction* has the form `reaction(`*triggers*`) ->` *effects* `{=` *body* `=}`, where *triggers* is a set of triggers, *effects* is the ports and actions that the reaction *may* write to (resp., schedule), and *body* specifies the reaction's behavior in a *target language* supported by LF (currently C, C++, TypeScript, Python, and Rust); hence "Lingua Franca." A reaction is invoked whenever at least one of its triggers is present, even if some other triggers are absent; however, the reaction is only executed when it is known for each trigger whether it will be present or absent at this logical time (see below). Depending on the values of state variables and inputs/actions, a reaction invocation may or may not produce declared outputs/actions. Reaction executions are considered to be *logically instantaneous*.

Ports are connected by *immediate* and *delayed connections*. An event travels *logically instantaneously* from an output port to the input ports connected to it by immediate connections.

Events are time-stamped with tags $\langle t, m \rangle$ consisting of a *time value* t and an integer $m \in \mathbb{N}$, called *micro-step* index. An event may also carry a value that will be passed as an argument to triggered reactions. LF distinguishes between events from the physical environment and events produced under the control of the program by considering two time lines: *logical time* and *physical time*.

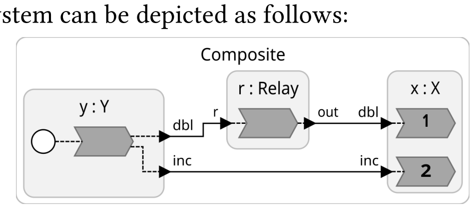
Since reaction executions (or "invocations") are considered to be logically instantaneous, the execution of a reaction r may produce an output that travels along an *immediate* connection and therefore triggers another reaction r' *at the same logical time*. The execution of r' may trigger another reaction r'' at this same logical time, and so on. LF has a *discrete-event* ("timed synchronous-reactive") semantics: these reactions execute at the same logical time in one "big step."

The execution of reactions triggered *at the same logical time* must satisfy: (i) a reaction r_1 with a declared output that is connected with an immediate connection to an input that may trigger r_2 must be executed before r_2; and (ii) if r_3 and r_4 are reactions in the *same reactor*, then they must be executed in their order of declaration (to ensure determinism and mutual exclusion, since they access the same state variables). These constraints can be expressed as a directed graph, called the *acyclic precedence graph* (APG) [13], with reactions as nodes and the above constraints as edges; any such APG must be acyclic for a program to be a *valid* LF program.

Example 2.1. The following shows an LF program taken from [11]:

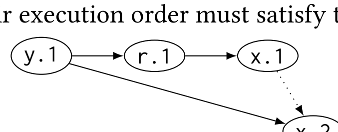
```
reactor X {
  input dbl:int;   input inc:int;
  state s:int = 1;
  reaction(dbl) {= self->s *= 2;   =}
  reaction(inc) {= self->s += inc; =}
}
reactor Relay {
  input r:int;     output out:int;
  reaction(r) -> out {=
    lf_set(out, r->value); =}
}
reactor Y {
  output dbl:int;  output inc:int;
  reaction(startup) -> dbl, inc {=
    lf_set(dbl, 1); lf_set(inc, 1); =}
}
main reactor {
  x = new X();
  r = new Relay();
  y = new Y();
  y.dbl -> r.r;
  r.out -> x.dbl;
  y.inc -> x.inc;
}
```

The architecture of this system can be depicted as follows:



The system has three reactors `y`, `r`, and `x`, only immediate connections, and reactions (depicted as chevrons). The white circle is the predefined logical action `startup`, which triggers at startup.

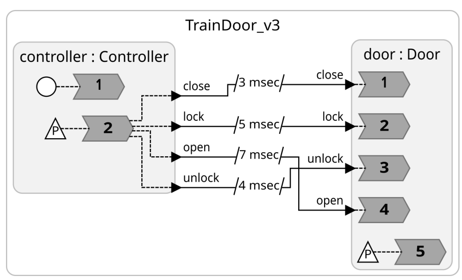
Since all connections are instantaneous, all the reactions are triggered at the same logical time by the `startup` action. Since their execution order must satisfy the constraints in the APG



the execution of reaction **2** in `x` must wait until reaction **1** in `x` has finished executing. The system is therefore deterministic: the final value of the variable `s` of `x` is 3. Although it does not happen with the reaction code in this example, *if* some invocations of the reaction in `relay` does not produce output, in those cases the execution of **2** obviously cannot wait for **1** to finish. □

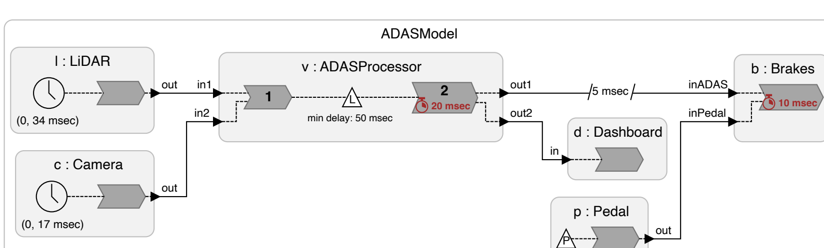
Example 2.2. The following shows the LF specification and the architecture of the third train door controller system in [20], with a `Controller` reactor `controller` and a `Door` reactor `door`:

```
reactor Controller {
 output lock: bool
 output open: bool
 output unlock: bool
 output close: bool
 physical action external: bool
 reaction(startup) {=
  /* initialize system */ =}
 reaction(external) -> close, lock,
      open, unlock {=
  if (external->value == true)
  { lf_set(close, true);
    lf_set(lock, true); }
  else { lf_set(open, true);
    lf_set(unlock, true); } =}
}
reactor Door {
 input lock: bool
 input unlock: bool
 input open: bool
 input close: bool
 state locked: bool = false
 state isOpen: bool = false
 physical action extOpen: bool
 reaction(close) {=
  self->isOpen = false; =}
 reaction(lock) {=
  if (self->isOpen == false)
   self->locked = true; =}
 reaction(unlock) {=
  self->locked = false; =}
 reaction(open) {=
  if (self->locked == false)
   self->isOpen = true; =}
 reaction(extOpen) {=
  if (self->locked == false)
   self->isOpen = true; =}
}
main reactor {
 c = new Controller()
 d = new Door()
 c.lock -> d.lock after 5 msec
 c.unlock -> d.unlock after 4 msec
 c.open -> d.open after 7 msec
 c.close -> d.close after 3 msec
}
```



The controller controller has a Boolean physical action (marked 'P') external modeling the driver wanting to either close-and-lock the door (external -> value == true), which produces output to its close and lock output ports, or to unlock-and-open the door, which produces outputs to its ports open and unlock. All connections are *delayed*: the delay to lock is greater than to close, which should ensure that the door is closed before it is (tried to be) locked; and vice versa for unlocking and opening the door. The variables locked and isOpen model the state of the door. The Door reactor door has a physical action extOpen, modeling a passenger pushing the "Open Door" button. ◻

Example 2.3. The following shows the architecture of the advanced driver assistance system (ADAS), which is the running example in [8].



A Camera and a LiDAR are connected to an ADASProcessor, which has a delayed connection to the Brakes. The LiDAR and Camera components are triggered periodically by *timers* with offset 0 and periods 34 and 17. Reaction **1** in the ADASProcessor schedules an action (for reaction **2**) with delay 50. The driver can press the brake pedal at any time; this is modeled by a physical action in the Pedal component. See [8] and Section 5 for details. ◻

2.2 Rewriting Logic and Maude

Rewriting logic [14] is a computational logic where data types are defined by algebraic equational specifications and local state changes are modeled by (possibly conditional) labeled rewrite rules. Rewriting logic is suitable for modeling distributed systems in an object-oriented style. Maude [4, 5] is a specification language and high-performance simulation, reachability analysis, and linear temporal logic (LTL) model checking tool for rewriting logic.

A Maude module $M = (\Sigma, E, R)$ specifies a *rewrite theory*, with:

- Σ an algebraic *signature*; i.e., a set of *sorts*, *subsorts*, and *function symbols*.
- (Σ, E) a *membership equational logic* [15] theory, with E a set of possibly conditional equations and membership axioms, specifying the data types of the system.
- R a collection of *labeled conditional rewrite rules* $[l] : t \longrightarrow t'$ **if** *cond*, for a *label* l, and terms t and t', specifying the system's local transitions.

We summarize the syntax of Maude and refer to [5] for details. A function f is declared `op` f : $s_1 \dots s_n$ `->` s, where $s_1 \dots s_n$ denotes the sorts of its arguments, and s its sort. We can declare that the function is a constructor (`ctor`) of elements of sort s; and binary function symbols can be declared to be associative (`assoc`), commutative (`comm`), and/or have an identity element t (`id:` t, or `right id:` t for a right identity element t), so that computation is performed *modulo* such properties. Underbars ('_') in function names denote argument positions in "mix-fix" notation.

Equations and rewrite rules are introduced with, respectively, keywords `eq`, or `ceq` for conditional equations, and `rl` and `crl`. They are implicitly universally quantified by the mathematical variables appearing in them; such variables are declared with the keywords `var` and `vars`, or can have the form *var*:*sort* and be introduced on the fly. An equation $f(t_1, \dots, t_n) = t$ marked `owise` ("otherwise") can be applied to $f(\dots)$ only if no other equation with left-hand side $f(u_1, \dots, u_n)$ can be applied. A module *imports* another module M using `including` M.

In *object-oriented modules* (`omod` M `is` ...`endom`), a declaration `class` C | att_1 : s_1 , ..., att_n : s_n declares a *class* C of objects with attributes att_1 to att_n of sorts s_1 to s_n. An *object instance* of class C is represented as a term of the form < O : C | att_1 : val_1 , $\cdots$, att_n : val_n >, where O, of sort `Oid`, is the object's *identifier*, and val_1 to val_n are the values of the attributes att_1 to att_n. A state is modeled as a term of the sort `Configuration`, and is a *multiset* of objects and messages. The dynamic behavior of a system is axiomatized by specifying its transitions by rewrite rule; e.g., the rule

```
rl [l] : < O : C | a1 : Y,    a2 : O2 >  < O2 : C2 | b : X >
     =>  < O : C | a1 : Y + X, a2 : O2 >  < O2 : C2 | b : 0 > .
```

defines a family of transitions in which an object `O` of class `C` adds the value of the attribute `b` (which is then set to `0`) of the object `O2` of class `C2` to its attribute `a1`. Attributes whose values do not change and do not affect the next state need not be mentioned.

Formal Analysis. The command `red` *expr* reduces the expression *expr* to its normal form using the equations E. The rewrite command `rew [`n`]` *init* simulates at most n steps of one behavior from the initial state *init* by applying rewrite rules. Given a state pattern *pattern* and an (optional) condition *cond*, Maude's `search` command searches the reachable state space from *init* for all (or optionally a given number of) states that match *pattern* such that *cond* holds:

`search` *init* `=>*` *pattern* [`such that` *cond*] .

The search command can have arguments denoting the maximal number of desired solutions and/or the maximal depth of the search tree. The command `red modelCheck(`*init*, ϕ`)` checks whether the LTL formula ϕ is satisfied by the initial state specified by the term *init*. Atomic propositions in the formula ϕ are user-defined terms of sort `Prop`, and the function `op _|=_ : State Prop -> Bool` specifies which states satisfy a given proposition. LTL formulas are built from state formulas, Boolean connectives, such as `~` (negation), `/\` (conjunction), `\/` (disjunction), and `->` (implication), and the temporal logic operators `[]` ("always"), `<>` ("eventually"), and `U` ("until").

Real-Time Systems. Real-time systems can be modeled as real-time rewrite theories [19], where ordinary rewrite rules model instantaneous change, and where *tick* rewrite rules `crl [`*tick*`] : {`t`} => {`t'`} in time` τ `if` *cond* model time advance: the system may evolve from state `{`t`}` to state `{`t'`}` in time τ. The states have the form `{`u`}`, so that time advances uniformly in all parts of the state. As shown in this paper, real-time rewrite theories can be specified and analyzed in Maude. Time-specific analysis command, such as *timed* (or "metric") CTL model checking [7], are provided by the Real-Time Maude tool [18], which currently only runs on the older version 2 of Maude.

3 Rewriting Logic Semantics for LF with Multiple Reaction Triggers

This section presents our executable rewriting logic semantics of the subset of Lingua Franca described in Section 3.1. Section 3.2 explains how a Lingua Franca model can be represented as a Maude term, and Sections 3.3 to 3.6 describe how Lingua Franca models can be executed in Maude.

3.1 Subset of Lingua Franca Considered

Like other approaches to Lingua Franca verification [8, 12, 20] we do not treat physical time, and only consider logical time. We therefore do not cover physical actions in their fullest generality, with deadlines, which relate model time and physical time. In [8, 20], physical actions are *not* covered (see, e.g., [8, p. 6]; each physical action in the LF model is translated into one with a *fully deterministic* behavior in the Timed Rebeca model in [20]). In contrast to [8, 20] we support a fairly general nondeterministic model of physical actions: each physical actions has a "period" and a finite range of possible "values." We then periodically choose nondeterministically whether or not the physical action takes place; if it takes place, its value is also chosen nondeterministically.

In contrast to [12], we support multiple triggers of reactions, which makes it significantly harder to formalize the semantics of LF.[1]

The reaction code is given in a target language (C, Rust, etc.). As explained in Section 6, we have extended the LF compiler to *automatically* transform reaction code in the fragment of C targeted by Lin et al. [8] into Maude.

3.2 Representing LF models in Maude

Following [12], we formalize LF programs and their semantics in an object-oriented style in Maude. An LF program is modeled as a term where each reactor r is modeled as an object

```
< r : Reactor | inports : inputPorts,  outports : outputPorts, state : variablesAndValues,
                timers : timerObjects, actions : actionObjects, reactions : listOfReactions >
```

inputPorts and *outputPorts* are sets of port objects of the form `<` *name* `: Port | value : [`*val*`] >`, *variablesAndValues* is a term of the form x_1 `|-> [`v_1`] ; ... ;` x_n `|-> [`v_n`]`, where v_i is the value of the state variable x_i. *timerObjects* and *actionObjects* are sets of, respectively, `Timer` and `PhysicalAction` and `LogicalAction` objects. *listOfReactions* is a *list* of reactions (since their order matters), where each LF reaction `reaction(`*triggers*`) ->` *effects* `{=` *code* `=}` is modeled by a Maude term `reaction when` *triggers* `-->` *effects* `do {`*code*`}`, where '`-->` *effects*' can be omitted if no effects are declared.

An immediate connection is modeled as a term `(`r `:` *outPortName* `-->` r' `:` *inPortName*`)` and a delayed connection is modeled as a term `(`r `:` *outPortName* `--` *delay* `-->` r' `:` *inPortName*`)`.

[1]The ADAS system in Theorem 2.3 is the running example in both [8] and [12]. However, Lin at al. [8] remove the Pedal component in their verification, since they cannot handle physical actions, while Marin et al. [12] remove the Camera component, since they cannot handle multiple reaction triggers.

Example 3.1. The following term `init` represents the LF model in Example 2.2:[2]

```
eq init =
< controller : Reactor | inports : none,  state : empty,  timers : none,
    outports : < lock : Port | value : [false] >  < open : Port | value : [false] >
               < unlock : Port | value : [false] >  < close : Port | value : [false] >,
    actions : < startup : LogicalAction | minDelay : 0, minSpacing : 0,
                                          policy : defer, payload : [false] >
            < external : PhysicalAction | minDelay : 0, ..., payload : [false] >,
    reactions : (reaction when startup do {skip})
                 reaction when external --> close ; lock ; open ; unlock do {if ... fi} >
< door : Reactor |  outports : none,  timers : none,
    inports : < lock : Port | value : [false] >  < unlock : Port | value : [false] >
              < open : Port | value : [false] >  < close : Port | value : [false] >,
    state : (locked |-> [false])  ; (isOpen |-> [false]) ; (counter1 |-> [0]),
    actions : < extOpen : PhysicalAction | minDelay : 0, ..., payload : [false] >,
    reactions : (reaction when close do {isOpen := [false]})
                (reaction when lock do {if isOpen === [false] then locked := [true] fi})
                (reaction when unlock do {locked := [false]})
              (reaction when (extOpen ; open) do {if locked === [false] then isOpen := [true] fi}) >
(controller : lock -- 5 --> door : lock)       (controller : unlock -- 4 --> door : unlock)
(controller : open -- 7 --> door : open)       (controller : close  -- 3 --> door : close)
```
□

Defining the data types of these terms is straight-forward; e.g., the class `Reactor` is defined

```
class Reactor | inports : Configuration,   outports : Configuration,
                state : ReactorState,      reactions : ReactionList,
                timers : Configuration,    actions : Configuration .
```

Terms of sort `ReactionList` are lists (assoc) of terms of sort `Reaction`:

```
sorts ReactionList Reaction .  subsort Reaction < ReactionList .
op nil : -> ReactionList [ctor] .
op __ : ReactionList ReactionList -> ReactionList [ctor assoc id: nil] .
```

The sort `Reaction` is defined[3]

```
op reaction when_-->_do`{_`} : OidSet OidSet ReactionBody -> Reaction [ctor] .
op reaction when_do`{_`} : OidSet ReactionBody -> Reaction .
eq reaction when OS do {RB} = reaction when OS --> none do {RB} .
```

where `OidSet` is the sort of *sets* of object identifiers; in this case the identifiers of the triggering actions, timers, and input ports. The sort `ReactionBody` defines the grammar of the reaction code:

```
sort ReactionBody .
op skip : -> ReactionBody [ctor] .
op _;_ : ReactionBody ReactionBody -> ReactionBody [ctor assoc id: skip] .
op _:=_ : VarId Expr -> ReactionBody [ctor] .
op if_then_fi : BoolExpr ReactionBody -> ReactionBody [ctor] .
op if_then_else_fi : BoolExpr ReactionBody ReactionBody -> ReactionBody [ctor] .
op while_do_done : BoolExpr ReactionBody -> ReactionBody [ctor] .
op _<-_ : PortId Expr ->  ReactionBody [ctor] . --- write to output port
op schedule : ActionId IntExpr Expr -> ReactionBody [ctor] .
```

3.3 Execution States

Extending Marin et al. [12], with, e.g., invoked reactions, the states of our executions have the form

[2]Parts of Maude terms and output will be replaced by '. . .'.

[3]We usually do not show variable declarations, but follow the convention that variables are written in (all) capital letters.

```
{< queue : EventQueue | queue : eventQueue >
 < r_1 : Reactor | ... >  ...  < r_n : Reactor | ... >
 < env : Environment | physicalActions : physicalActionObjects >
 < rxns : Invoked | reactions : reactionsExecutedInLastStep >}
```

The queue object maintains the (tag-ordered) event queue of the system, whose entries are terms

`events_1 at tag(t_1,n_1) :: events_1 at tag(t_2,n_2) :: ...`

where $events_i$ is a *set* of *events*, with each event represented as a term event(*reactor*, *trigger*, *value*). t_i is the time *remaining* until $events_i$ should trigger reactions. We do *not* store the actual time values of the tags, since those could grow beyond any bound, making the reachable state space infinite (and hence unbounded model checking nonterminating) even when the reachable "untagged state space" would be finite. n_i is the microstep index of the events $events_i$.

The `< r_i : Reactor | ... >` objects model the reactors as explained in Section 3.2; in particular, they contain the current values of the reactor's state variables.

The env object maintains data about physical actions, each of which is modeled as an object

```
< r . a : PhysAct | leftOfPeriod : t, period : p,
                    possibleValues : rangeOfValues, timeNonDet : b >
```

where r is the reactor, a the action name, t the time remaining until the action may happen next, p is its "period," *rangeOfValues* is the finite set of values the action can have, and b is a flag denoting whether the physical action is time-nondeterministic (i.e., does *not* have to take place each period).

To be able to express and analyze also dataflow and "mixed" properties, we also store the set of reactions that were executed in the most recent step as the value *reactionsExecutedInLastStep*.

Example 3.2. A state during the execution of the train door system is

```
{< controller : Reactor | inports : none,  state : empty,   timers : none,
     outports : (< close : Port | value : [true] > < lock : Port | value : [true] >
                 < unlock : Port | value : [false] > < open : Port | value : [false] >),
     reactions : (reaction 1 when startup --> none do {skip}
                  reaction 2 when external --> close ; lock ; unlock ; open do {if ... fi}),
     actions : (< startup : LogicalAction | ..., payload : [false] >
                < external : PhysicalAction | ..., payload : [true] >) >
 < door : Reactor | inports : ..., outports : none, state : (isOpen |-> [false] ; locked |-> [true]),
     reactions : ..., timers : none, actions : < extOpen : PhysicalAction | ..., payload : [false] > >
 < env : Environment | physicalActions : (
       < controller . external : PhysAct | period : 10, leftOfPeriod : 10,
                                           possibleValues : ([true] : [false]),
                                           timeNonDet : false >
     < door . extOpen : PhysAct | period : 11, leftOfPeriod : 1, possibleValues : [true],
                                  timeNonDet : true >) >
 < queue : EventQueue |
     queue : (event(door, close, [true]) at tag(3, 0)) :: event(door, lock, [true]) at tag(5, 0) >
 < rxns : Invoked | reactions : (controller . 2) >
 (controller : close -- 3 --> door : close)   (controller : lock -- 5 --> door : lock)
 (controller : unlock -- 4 --> door : unlock)  (controller : open -- 7 --> door : open)} in time 10
```

This shows the state at time 10. The key variable values are that the door's isOpen value is false, and its locked value is true. The physical action external (modeling the driver) takes place (timeNonDet : false) every 10 time units (period : 10) and selects nondeterministically whether its value is true or false (possibleValues : ([true] : [false])). The physical action extOpen (modeling the passenger) nondeterministically chooses every 11 time units whether or not (timeNonDet : true) take place with value true. The (only) reaction that took place in the (big) step leading to this state was the second reaction of the controller (controller . 2). □

Defining these classes and data types is straight-forward; event queues can be defined as follows:

```
class EventQueue | queue : EQueue .
sort EQueue .    subsort TaggedEvents < EQueue .    op empty : -> EQueue [ctor] .
op _::_ : EQueue EQueue -> EQueue [ctor assoc id: empty] .

sort Event Events TaggedEvents .      subsort Event < Events .
op event : ReactorId ActionTrigger Value -> Event [ctor] .
op noEvent : -> Events [ctor] .
op __ : Events Events -> Events [ctor assoc comm id: noEvent] .
op _at_ : Events Tag -> TaggedEvents [ctor] .
```

3.4 Physical Actions

Each physical action is represented by an object of the class

```
class PhysAct | leftOfPeriod : TimeInf,  period : TimeInf,
                possibleValues : ValueSet,  timeNonDet : Bool .
```

where the sort `ValueSet` is a declared as a set of values:

```
sort ValueSet .    subsort Value < ValueSet .    op noValue : -> ValueSet [ctor] .
op _:_ : ValueSet ValueSet -> ValueSet [ctor assoc comm id: noValue] .
```

The following rule models that a physical action "happens" when its `leftOfPeriod` timer is 0:[4]

```
rl [actionHappens] :
   < (RI . AI) : PhysAct | leftOfPeriod : 0, period : P,  possibleValues : (V : VS) >
=> < (RI . AI) : PhysAct | leftOfPeriod : P >
   scheduleAction(event(RI, AI, V)) .

msg scheduleAction : Event -> Msg .
eq < O : Environment | physicalActions : CONF scheduleAction(EVENT) >
   < O2 : EventQueue | queue : QUEUE >
 = < O : Environment | physicalActions : CONF >
   < O2 : EventQueue | queue : schedule(EVENT, 0, QUEUE) > .
```

Since the possible values is a set (the set union operator `:` is declared to be associative and commutative), the selected value `V` can be *any* value in the set. The `leftOfPeriod` timer is reset, and the corresponding event `event(RI,AI,V)` is inserted into the event queue with delay `0`.

When `timeNonDet` is `true`, we select nondeterministically whether the action takes place when its timer expires; the following rule models that it does not:

```
rl [noAction] :
   < O : PhysAct | leftOfPeriod : 0, period : P, timeNonDet : true >
=> < O : PhysAct | leftOfPeriod : P > .
```

3.5 Execution

We define the dynamic behaviors of a system, whose states are given in Section 3.3, using three rewrite rules, in addition to those for physical actions shown above:

(1) a rule `step` defines the execution of one (big) step the system; and
(2) two `tick` rules advance time until the first events in the event queue are ready to be executed, or until a physical action timer expires.

The following rewrite rule performs a step in the system; i.e., executes all the reactions when the events `EVENTS` at the head (i.e., beginning) of the event queue have *remaining* tag `tag(0, N)`:

```
crl [step] :
    {< E : Environment | physicalActions : CONF1 >
     REACTORS-AND-CONNECTIONS
```

[4]We assume that the periods are larger than the minimum spacing between actions, but can enforce minimum spacing by resetting `leftOfPeriod` to `max(P, `ms`)` or `max(P, `$\lceil ms/\mathtt{P} \rceil * \mathtt{P}$`)`, where ms is the value of the action's `minSpacing` attribute.

```
     < Q : EventQueue | queue : (EVENTS at tag(0, N)) :: QUEUE >
     < RXNS : Invoked | >}
=>
    {< E : Environment | physicalActions : CONF1 >
     NEW-NETWORK
     < Q : EventQueue | queue : NEW-QUEUE >
     < RXNS : Invoked | reactions : INVOKED >}
 if   smallestTimer(CONF1) > 0
    /\ networkQueueRxns(NEW-NETWORK, NEW-QUEUE, INVOKED) :=
         executeStep(EVENTS, REACTORS-AND-CONNECTIONS, QUEUE) .
```

The reactions triggered by the events `EVENTS` at the head of the event queue are executed, which could trigger many other reactions at the same logical time. All those reactions are executed by the *function* `executeStep`, which takes as arguments these `EVENTS`, the current state of the reactors and the connections (`REACTORS-AND-CONNECTIONS`), and the *remaining* event queue `QUEUE`. This function returns a triple `networkQueueRxns(NEW-NETWORK, NEW-QUEUE, INVOKED)`, where `NEW-NETWORK` is the updated network (with the values of the state variables updated), `NEW-QUEUE` is the resulting event queue after the future events generated by the executed reactions have been added to `QUEUE`, and, `INVOKED` is the set of reactions that were executed in this step. The condition `smallestTimer(CONF1) > 0` forces the rule `actionHappens` to be taken before `step` when a physical action timer expires.

The tick rewrite rule advances time until either the remaining delay of the events at the head of the queue becomes 0, or until the next physical action expires (`min(T1, smallestTimer(CONF1))`):

```
crl [tick] :
     {< E : Environment | physicalActions : CONF1 >
      REACTORS-AND-CONNECTIONS
      < Q : EventQueue | queue : (EVENTS at tag(T1, N)) :: QUEUE >
      < RXNS : Invoked | >}
  => {< E : Environment | physicalActions : decreaseTimers(CONF1, T) >
      REACTORS-AND-CONNECTIONS
      < Q : EventQueue | queue : (EVENTS at tag(T1 monus T, N)) :: decreaseTags(QUEUE, T) >
      < RXNS : Invoked | reactions : none >}  in time T
     if T := min(T1, smallestTimer(CONF1)) /\ T > 0 .
```

This rule advances time until the next events in the queue become “ripe” or until some physical action timer expires. All physical action timer values decrease by `T` (`decreaseTimers(CONF1, T)`) and all “remaining time” values in the tags in the event queue decrease by `T` (`decreaseTags(QUEUE, T)`). A similar tick rule applies when the queue is empty, but there are physical actions in the system.

3.6 Executing a Single Step

The function

```
op executeStep : Events Configuration EQueue -> Network+Queue+Reactions .
```

executes a single step; i.e., all reactions that are triggered by the set of triggering events, *and* those triggered at the same logical time by executing these reactions, and so on. We must also take various constraints into account. For example, in Example 2.1, reaction **2** in `x` cannot execute before reaction **1** has executed, *if* reaction **1** executes at this logical time. Whether or not a reaction execution produces output (and, if so, what) depends on the values of its state variables and inputs.

Our idea is to *dynamically* maintain a *runtime acyclic precedence graph* (R-APG) during the execution of a big step. Given a set of events, we construct an R-APG whose nodes are all the reactions that *possibly could* execute at the same logical time as those triggered by the events. The edges in our R-APG correspond to those in the LF APG. In addition, we mark each node in the

R-APG with a status, which could be either executed (the reaction has already been executed in this step), absent (we know that the reaction will not be triggered at this logical time), unknown (it is too early to tell whether the reaction will be triggered or not at this logical time), or present (at least one of the reaction's triggers is/will be present, but the reaction has not yet been executed). Updating the R-APG and executing reactions in a big step must go hand in hand.

The function executeStep therefore starts by building the initial R-APG for EVENTS:

```
eq executeStep(EVENTS, NETWORK, QUEUE)
 = executeStep(generateAPG(EVENTS, NETWORK),
              addEventsToPorts(EVENTS, NETWORK), QUEUE, none) .
```

This second function executeStep maintains the R-APG as its first argument, the network is the second argument, the event queue is the third argument, and the reactions executed is the fourth argument. The key idea behind defining this function is to use an equation

```
eq executeStep(rApg, network, queue, reactionsInvoked)
 = executeStep(newRApg, newNetwork, newQueue, newReactionsInvoked)
```

that executes *some* reaction r in the *rApg* marked present, *if* all its predecessors in *rApg* are marked either executed or absent. After that execution we also know what (if any) outputs the executed reaction has produced, and update the *rApg* with the additional knowledge we obtain. This is somewhat cumbersome, since a reaction may have many triggers, and it is sufficient that one of them is eventually present to trigger the reaction. If, as a result of executing a reaction r, we know that all inputs of reaction r' will be absent, not only must r' update its status to absent, but the information that r' will not produce any outputs in this step must be propagated throughout the R-APG; this again could lead to other nodes going from unknown to absent, and so on.

Executing a reaction r may also schedule future actions and/or output values to ports that are sources of *delayed* connections. Such future events must be added to the updated event queue *newQueue*. After executing the reaction, the state variables of the reactors may have changed, and *newNetwork* is the new network (state). Finally, we add the executed reaction r to the set of reactions invoked in this step (*newReactionsInvoked*).

When we have reached a fixed point, and no reaction can be invoked, we return the new network (state), the new updated event queue, and the set of reactions invoked in this step:

```
eq executeStep(GRAPH,NETWORK,QUEUE,INVOKED) = networkQueueRxns(NETWORK,QUEUE,INVOKED) [owise] .
```

It is worth remarking that the main equation executes *some* reaction r that is enabled. There could be multiple such reactions r that could be executed. The key thing is that the semantics of Lingua Franca is deterministic, so that the order of execution of the reactions in a big step does not matter, as long it respects the constraints represented by the APG.

The above scheme can be seen as a *general scheme* for executing a set of "reactions" subject to a (possibly dynamically changing, as here) "constraint" or "strategy" on those reaction executions.

A node in the R-APG is formalized as an object of the following class

```
class APGNode | triggers : TriggerStatus,   pre : ReactionIdSet,
                succ : ReactionIdSet,       status : ExecutionStatus .

sorts ExecutionStatus TriggerStatus .  op empty : -> TriggerStatus [ctor] .
ops executed absent unknown present : -> ExecutionStatus [ctor] .
op _|->_ : ActionTrigger ExecutionStatus -> TriggerStatus [ctor] .
op _;_ : TriggerStatus TriggerStatus -> TriggerStatus [ctor assoc comm id: empty] .
```

For *each* trigger of the reaction, the attribute triggers stores whether the trigger is unknown, present, or absent. The attributes pre and succ denote the predecessor and successor nodes in the R-APG.

The above main equation can then be defined in Maude as follows:

```
ceq executeStep(< (REACTOR . N) : APGNode | triggers : TRIGGERS-STATUS, status : present,
                                            pre : PRE >   GRAPH,
                NETWORK
                < REACTOR : Reactor | reactions : REACTIONS1
                                                  reaction N when TRIGGERS --> OS do BODY
                                                  REACTIONS2 >,
                QUEUE,  INVOKED)
  = executeStep(< (REACTOR . N) : APGNode | status : executed >
                updateGraph(GRAPH, REACTOR, OS, NETWORK < REACTOR : Reactor | >, OUTPUTS),
                propagateImmediateOutputs(OUTPUTS, NETWORK OBJECT),
                scheduleDelayedInputs(OUTPUTS, NETWORK, RESULT-QUEUE),
                INVOKED ; (REACTOR . N))
 if presetOK(PRE, GRAPH)   /\  allTriggersDecided(TRIGGERS-STATUS)  /\
      result(OBJECT, OUTPUTS, RESULT-QUEUE) :=
        executeReaction(< REACTOR : Reactor |  >, N, TRIGGERS-STATUS, QUEUE) .
```

In this equation, the reaction (REACTOR . N) has status present in the R-APG, each of its predecessor nodes in the R-APG has either been executed or will not be executed at this logical time (presetOK(PRE, GRAPH)), *and* the status of each of its triggers has been decided (since all inputs must be consumed/read when a reaction is executed). The reaction (REACTOR . N) can therefore be executed by the function executeReaction, which returns the new state OBJECT of the reactor REACTOR, the outputs OUTPUTS generated by executing (REACTOR . N), and the event queue resulting from inserting the events scheduled in the execution of the reaction (RESULT-QUEUE). The outputs OUTPUTS can be connected to *immediate* or *delayed* connections. The latter must be added to the event queue (scheduleDelayedInputs(OUTPUTS, NETWORK, RESULT-QUEUE)), the former must be added to the corresponding input ports since they will trigger executions in this step (propagateImmediateOutputs(OUTPUTS,NETWORK OBJECT)). The R-APG must also be updated by the results of executing the reaction (updateGraph(...)).

The specification of executeReaction, propagateImmediateOutputs, and scheduleDelayedInputs is fairly straight-forward (see the Maude code for details). The function updateGraph is more subtle, and also uses a "fixed-point" style definition. The following equation specifies the case when there is an immediate connection (REACTOR3 : PORTID3 -> REACTOR2 : PORTID2) and we know that reaction (REACTOR3 . N3) will not execute. Since it is this reaction that would generate output to (REACTOR3 : PORTID3), we know that REACTOR2's PORTID2 will be absent, and this *could* change the status of (REACTOR3 : PORTID3):

```
eq updateGraph(< (REACTOR2 . N2) : APGNode | triggers : ((PORTID2 |-> unknown) ; TRIGGERS-STATUS),
                                             status : STATUS > GRAPH
                < (REACTOR3 . N3) : APGNode | status : absent >,
               REACTOR1, OS,
               NETWORK
               < REACTOR3 : Reactor |
                     reactions : (RL1 (reaction N3 when OS1 --> (PORTID3 ; OS2) do RB) RL2) >
               (REACTOR3 : PORTID3 --> REACTOR2 : PORTID2),
               EVENTS)
 = updateGraph(< (REACTOR2 . N2) : APGNode | triggers : ((PORTID2 |-> absent) ; TRIGGERS-STATUS),
                                             status : (if STATUS == unknown and
                                          allTriggersDecided((PORTID2 |-> absent) ; TRIGGERS-STATUS)
                                                         then absent else STATUS fi) >
               < (REACTOR3 . PORTID3) : APGNode | > GRAPH, REACTOR1, OS,
               NETWORK  < REACTOR3 : Reactor | >  (REACTOR3 : PORTID3 --> REACTOR2 : PORTID2), EVENTS) .
```

4 Formal Analysis of LF Models in Maude

Our formalization of the "logical-time" semantics of LF is executable, so that LF models can be analyzed using Maude and Real-Time Maude. Existing formal analysis methods for LF are:

(1) Sirjani et al. [20] provide state-based reachability analysis.
(2) Lin et al. [8] provide SMT-based *bounded model checking* for the restricted fragment of *safety MTL* (a subset of MTL, a linear temporal logic (LTL) where each temporal operator is annotated with a time interval) where each temporal operator is annotated with a *finite* intervals; e.g., $\Box_{[5,8]}(a \longrightarrow \Diamond_{[3,6]}b)$. Given such a formula ϕ and model M, Lin et al. compute a bound $\mathcal{CT}_{M,\phi}$, so that bounded model checking up to $\mathcal{CT}_{M,\phi}$ steps solves the model checking problem: ϕ holds in M if and only if ϕ holds in all behaviors of length $\mathcal{CT}_{M,\phi}$. Lin et al. [8] support three kinds of properties: (i) *state properties* consider the values of reactor variables, ports, and actions; (ii) *dataflow properties* are properties about *reactor invocations*; and (iii) *mixed properties* include both state and dataflow properties.
(3) Marin et al. [12] provide unbounded and step-bounded and time-bounded reachability analysis and ("untimed") LTL model checking, but only of *state properties.*

In this paper we fill the gaps and provide more than all the analysis methods above, including:

- unbounded and time-bounded reachability analysis for state, dataflow, and mixed properties;
- unbounded and time-bounded LTL model checking of all three kinds of properties; and
- full *timed* CTL model checking, using Real-Time Maude, of all three kinds of properties.

Relation to Lin et al.'s completeness threshold. Since Lin et al. base their analysis on SMT solving, they (can) only perform *bounded* model checking. However, they claim completeness of their analysis by computing a *completeness threshold* $\mathcal{CT}_{M,\phi}$ so that ϕ holds in all behaviors if and only if ϕ holds in all behaviors of length $\mathcal{CT}_{M,\phi}$. The reason is that Lin et al. use a fragment of safety MTL where it is sufficient to consider behaviors up to time $h(\phi)$, which is less than or equal to the sum of the upper bounds of the intervals of the temporal operators in ϕ. For example, for the formula $\Box_{[5,8]}(a \longrightarrow \Diamond_{[3,6]}b)$ it is sufficient to consider all behaviors up to time 14. Since Lin et al. consider an "event-based" semantics, and since they do step-bounded instead of time-bounded analysis, they must compute an upper bound on how many reactions could be be invoked within time $h(\phi)$.

Since we can do time-bounded analyses directly, we can obtain terminating and complete analyses for the same formula ϕ by just performing time-bounded analysis with time bound $h(\phi)$.

4.1 Initial States

Section 3.2 explains how we represent an LF program as a Maude term. Our states during executions must also include the additional infrastructure needed during executions, as explained in Section 3.3. The initial states of our analyses are defined as follows in general:

```
op initSystem : -> GlobalSystem .
eq initSystem =
    {< env : Environment | physicalActions : physActions >
     addReactionIndices(init)
     < queue : EventQueue | queue : addInitialTimers(init, addStartup(...,empty)) >
     < rxns : Invoked | reactions : none >} .
```

where *physActions* contains a `PhysicalAction` object for each physical action. We need a way to identify reactions in the R-APG and to express dataflow properties; `addReactionIndices(init)` adds numbers to each reaction in `init`. The function `addInitialTimers` adds the initial timer events to the event queue, and `addStartup` adds desired actions that should trigger (at) the start of the system.

Example 4.1. The initial execution state for the train door example, whose behaviors are triggered by physical actions, is

```
eq initSystem =
{< env : Environment | physicalActions :
    < (controller . external) : PhysAct | leftOfPeriod : 0,  period : 10,  timeNonDet : false,
                                     possibleValues : ([true] : [false]) >
    < (door . extOpen) : PhysAct | leftOfPeriod : 0,  period : 11,
                              possibleValues : ([true]),  timeNonDet : true >  >
 addReactionIndices(init)
 < queue : EventQueue | queue : empty >   < rxns : Invoked | reactions : none >} .
```

□

4.2 Analysis Methods

By including different predefined versions of the `tick` rewrite rule, we can perform unbounded, time-bounded, and/or "step-bounded" simulation and reachability analysis, as well as unbounded and time-bounded (untimed) LTL model checking directly in Maude. We have "ported" our semantics to Real-Time Maude, which provides full *timed* CTL model checking, so that we can subject LF models to unbounded and time-bounded timed CTL model checking, as well as other time-specific analyses, such finding the earliest and latest time needed to reach a desired state.

4.2.1 Unbounded and Time-bounded Analysis. The tick rules in Section 3.5 are "standard" rules for *specifying* real-time rewrite theories. In Real-Time Maude, these are the only ones we need, except for *time-bounded* timed CTL model checking. Our tick rules are suitable for step-bounded simulation: in the train example, the command `rew [10] initSystem` returned the state shown in Example 3.2. However, these tick rules add a "system clock," whose value may grow beyond any bound in nonterminating systems, to the state, making the reachable "clocked" state space infinite even when the reachable "unclocked" state space is finite.

For *unbounded* analysis we should use tick rules that do *not* add a clock to the state. We have different versions of the tick rules in different modules. For *unbounded* analysis, the module `UNBOUNDED-ANALYSIS-DYNAMICS` should be imported; it replaces each original tick rule `crl [tick] :` `{`t`} => {`u`} in time T if` *cond* (in the module `SIMULATION-DYNAMICS`) with the rule `crl [tick] : {`t`}` `=> {`u`} if` *cond*, *without* the "`in time T`" part.

For *time-bounded* analysis, we keep the *system clock*, and use the module `TIME-BOUNDED-DYNAMICS`, which replaces each tick rule `crl [tick] : {`t`} => {`u`} in time T if` *cond* in our original semantics with the corresponding rule

```
crl [tick] : {t} in time T2 => {u} in time T2 + T if T2 + T <= timeBound /\ cond .
```

We must then define the value of `timeBound`, and use the initial state `initSystem in time 0`.

In this way, we can do time-bounded and unbounded analysis by just including different modules:

Example 4.2. In the train door system, the module `TEST-TRAIN` defines the initial state. The module `SIMULATION-TRAIN` has our original rules, the module `UNCLOCKED-TRAIN` should be used for *unbounded* analysis, and `TIME-BOUNDED-TRAIN` should be used for *time-bounded* analysis:

```
omod TEST-TRAIN is
  including TRAINDOOR-V3 .
  including DYNAMICS-WITHOUT-TICK .
  ops env queue rxns : -> Oid [ctor] .
  op initSystem : -> GlobalSystem .  eq initSystem = ...    --- initial state as above
endom

omod SIMULATE-TRAIN is
  including TEST-TRAIN .
```

```
  including SIMULATION-DYNAMICS .
endom

omod UNCLOCKED-TRAIN is
  including TEST-TRAIN .
  including UNBOUNDED-ANALYSIS-DYNAMICS .
endom

omod TIME-BOUNDED-TRAIN is
  including TEST-TRAIN .
  including TIME-BOUNDED-DYNAMICS .
  eq timeBound = 150 .
endom
```
□

In Maude, we can specify the module in the analysis command (e.g., `search in UNCLOCKED-TRAIN : initSystem =>* ...`). Below we do not specify the module in which the analysis is done, but assume that the "current" module is the appropriate one for the analysis performed.

4.2.2 Reachability Analysis. We can define search patterns on the execution state in search commands, including values of state variables, the invoked reactions in a state, the content of the event queues, and so on. The following *unbounded* search command checks whether it is possible to reach a state in which the state variables `isOpen` and `locked` in the `door` reactor both have the value `true`:

```
Maude> search [1] initSystem =>* {REST:Configuration < door : Reactor | ATTS:AttributeSet,
                               state : (locked |-> [true]) ; (isOpen |-> [true]) ; RS:ReactorState >} .

No solution.
```

The variables `REST:Configuration`, `ATTS:AttributeSet`, and `RS:ReactorState` capture, respectively, the other objects in the state, the other attributes in the object `door`, and the other variables in `door` (superfluous in this case). It is worth noting that the reachable state space, with event queues, is finite in this case, so that a search for an unreachable state terminates.

To illustrate time-bounded reachability analysis and *mixed* properties, we perform a *time-bounded* search (in `TIME-BOUNDED-TRAIN`) to check whether the variable `locked` could be `false` after a step in which the second reaction (the one triggered by `lock`) in `door` has been executed:

```
Maude> search [1] initSystem in time 0 =>* {REST:Configuration
             < door : Reactor | state : (locked |-> [false]) ; RS:ReactorState, ATTS:AttributeSet >
            < rxns : Invoked | reactions : (door . 2) ; R:ReactionIdSet >} in time T:Time .

Solution 1 (state 414)
REST:Configuration --> ...    ATTS:AttributeSet --> ...    R:ReactionIdSet --> ...
RS:ReactorState --> isOpen |-> [true]    T:Time --> 35
```

The Maude output shows that such an unexpected (?) situation could happen at time 35: the door is open when the `lock` signal arrives. The Maude command `show path 414` shows the whole behavior.

4.2.3 LTL Model Checking. We can use Maude's linear temporal logic (LTL) model checker to perform both unbounded and time-bounded LTL model checking. We define the following *generic* atomic propositions; the user may in addition define her own model-specific propositions:

```
subsort ClockedSystem < State .  var REST : Configuration .  var REACTORID : ReactorId .  var TAG : Tag .
var RIDS : ReactionIdSet . var VAR : VarId . var VAL : Value . var RS : ReactorState . var EVENT : Event .
var T : Time .  var O : Oid .  var REACTION : ReactionId .  var PROP : Prop .  vars EQ1 EQ2 : EQueue .

op _in_is_ : VarId ReactorId Value -> Prop [ctor] .
eq {REST < REACTORID : Reactor | state : (VAR |-> VAL) ; RS >} |= VAR in REACTORID is VAL = true .
```

```
op _isInQueue : Event -> Prop [ctor] .
eq {REST < 0 : EventQueue | queue : (EQ1 :: (EVENT at TAG) :: EQ2) >} |=  EVENT isInQueue = true .

op _invoked : ReactionId -> Prop [ctor] .
eq {REST < 0 : Invoked | reactions : (REACTION ; RIDS) >} |=  REACTION invoked = true .

ceq {REST} in time T |= PROP = true if {REST} |= PROP .
```

var `in` *reactor* `is` *value* holds if the current value of the state variable *var* in the reactor *reactor* is *value*. *e* `isInQueue` holds if the event *e* is in the event queue, and *reaction* `invoked` holds if the reaction *reaction* was invoked in the step leading to the current state. The last equation extends the definition of all propositions to "clocked" states, for time-bounded LTL model checking.

Example 4.3. We check the mixed property whether the weird situation where the door is unlocked even though the second `door` reaction was just invoked can always happen:

```
Maude> red modelCheck(initSystem, <> (locked in door is [false] /\ (door . 2) invoked)) .
```

This command returns a counterexample, since the controller *could* always want to open the door.

The main property of interest is under what circumstances a very "pushy" passenger can keep the doors open forever, from some time on, no matter what the driver does. The easiest way to analyze this property is to change the `timeNonDet` flag of the physical action `door . extOpen` to `false` and check whether this guarantees that the doors will eventually stay open forever:

```
Maude> red modelCheck(initSystem, <> [] (locked in door is [false])) .
```

This command returns `true` when the passenger pushes the button *every two time units*; otherwise there is a counterexample to this property: the controller *can* avoid being stuck with an unlocked door forever if the user cannot push the button more often than every three time units. □

4.2.4 Timed CTL Model Checking. We could easily port our specifications to Real-Time Maude, and can therefore also perform Real-Time Maude analyses on LF models. For example, we can use Real-Time Maude's timed CTL model checker to check whether the door *always* will open *within seven time units* after the driver wants them opened ($\forall\Box(\textit{driverPushOpen} \longrightarrow \forall\Diamond_{\leq 7}\ \textit{doorOpen})$):

```
Maude> (mc-tctl initSystem |= AG ((event(controller, external, [false]) isInQueue)
                                 implies AF[<= than 7] (isOpen in door is [true])) .)

Property satisfied
```

The property does not hold for bounds smaller than seven.

We can also use other time-specific Real-Time Maude commands, such as finding the shortest and longest time needed to reach some state; for example the "mixed" state where the door is unlocked even though the reaction receiving a `lock` signal was just invoked:

```
Maude> (find earliest initSystem =>* {REST:Configuration
                 < door : Reactor | state : (locked |-> [false]) ; RS:ReactorState, ATTS:AttributeSet >
                  < rxns : Invoked | reactions : (door . 2) ; R:ReactionIdSet >} .)

Result: {...} in time 35

Maude> (find latest initSystem =>* {REST:Configuration
              < door : Reactor | state : (locked |-> [false]) ; RS:ReactorState, ATTS:AttributeSet >
             < rxns : Invoked | reactions : (door . 2) ; R:ReactionIdSet >}  in time < 50 .)

Result: there is a path in which the pattern is not reachable in time < 50
```

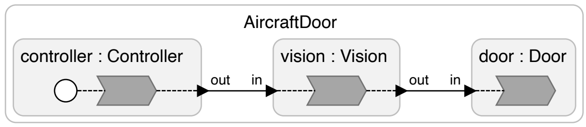


Fig. 1. Architecture of the Aircraft Door controller.

```
target C;
reactor Controller {
  output out:int;
  reaction(startup) -> out {=
    lf_set(out, 1);         =}
}
reactor Vision {
  input in:int;
  output out:int;
  state ramp:int(0);
  reaction(in) -> out {=
    if (self->ramp == 1) {
      lf_set(out, 0);
    } else {
      lf_set(out, 1);
    }                  =}
}
reactor Door {
  input in:int;
  state doorOpen:int;
  reaction(in) {=
    if (in->value == 1)
      self->doorOpen = 1;
    else if (in->value == 0)
      self->doorOpen = 0;        =}
}
main reactor AircraftDoor {
  controller = new Controller();
  vision = new Vision();
  door = new Door();
  controller.out -> vision.in;
  vision.out -> door.in;
}
```

Fig. 2. LF specification of the Aircraft Door controller.

Since Real-Time Maude assumes that the tick rules are "simulation" tick rules, *time-bounded* TCTL model checking can be obtained by adding an object < `tb : Bound | timeLeft :` *timeBound* > to the state, and modify the tick rules to not advance time beyond when the `timeLeft` value reaches 0.

5 Case Studies and Benchmarking

In [8], a benchmark suite of 22 LF programs, drawn from real-world applications and other benchmarks suits, is used to evaluate the LF Verifier. We have successfully verified the desired—or stronger—properties of all programs in this suite. This section presents five representative examples (Sections 5.1 to 5.5), demonstrating how nontrivial properties involving physical actions, multiple reaction triggers, and timing constraints can be efficiently verified. We also compare our analyses against the LF Verifier (Section 5.6), which shows that our analyses drastically outperform theirs.

5.1 An Aircraft Door Controller

We consider a simple aircraft door, which is either open or closed. Figure 1 shows the LF model for an aircraft door controller, with the LF code in Figure 2. The `Controller` reactor sends a signal to the `Vision` reactor, which checks its internal state variable `ramp` before sending either an open or close command to the `Door` reactor. Upon receiving an input, the `Door` reactor updates the `doorOpen` state variable to the received value. The corresponding Real-Time Maude code is shown below:

```
(tomod AIRCRAFT-DOOR is including RUNTIME-APG .
  ops ramp doorOpen : -> IVarId [ctor] .     ops controller vision door : -> ReactorId [ctor] .
  ops in out : -> IPortId [ctor] .           op startup : -> IActionId [ctor] .
  op init : -> Configuration .
  eq init =
    < controller : Reactor | inports : none, outports : < out : Port | value : [0] >,
      state : empty, timers : none,
      actions : < startup : LogicalAction | minDelay : 0, minSpacing : 0, policy : defer, payload : [0] >,
      reactions : reaction when startup --> out do {out <- [1]} >
    < vision : Reactor | inports : < in : Port | value : [0] >, outports : < out : Port | value : [0] >,
      state : ramp |-> [0], timers : none, actions : none,
      reactions : reaction when in --> out do {if (ramp === [1]) then out <- [0] else out <- [1] fi} >
    < door : Reactor | inports : < in : Port | value : [0] >, outports : none,
      state : doorOpen |-> [0], timers : none, actions : none,
```

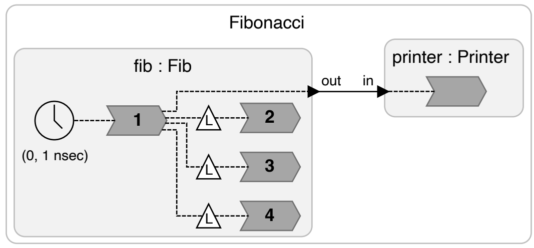


Fig. 3. Architecture of the Fibonacci system

```
      reactions : reaction when in do {doorOpen := in} >
    (controller : out --> vision : in)  (vision : out --> door : in) .
endtom)
```

LF VERIFIER verifies the following ("mixed") safety property: if the ramp variable of the Vision reactor initially is 0, then *in time 0* it should always be the case that *if* the first reaction (numbered '0' in LF VERIFIER) is invoked, then the door's doorOpen variable should be 1:

```
(AircraftDoor_vision_ramp == 0) ==>
    (G[0 sec](AircraftDoor_door_reaction_0 ==> (AircraftDoor_door_doorOpen == 1))))
```

Since this formula only concerns time 0, it can be easily checked in many ways. The following command analyzes the property in timed CTL in around 0.1 seconds:

```
Maude> (mc-tctl initSystem |= (valueOf ramp in vision is [0])   implies
                   AG[<= than 0] ((reaction (door . 1) invoked)  implies  (valueOf doorOpen in door is [1])) .)

rewrites: 28135 in 102ms cpu (107ms real) (274737 rewrites/second)
Property satisfied
```

5.2 Fibonacci Example

Figure 4 shows an LF program that generates Fibonacci numbers using two reactors (Figure 3). The Fib reactor is responsible for generating the next Fibonacci number. The Printer reactor receives the result of each computation and stores it in a state variable. A timer, which first fires at time 0, triggers the first reaction in Fib every nanosecond. This reaction sends the current Fibonacci number to its output port and schedules other logical actions. The corresponding Maude code is:

```
omod FIBONACCI is including LF-REPR . protecting NAT-LF-TIME .
  ops n result lastResult secondLastResult : -> RVarId [ctor] .
  ops fib printer : -> ReactorId [ctor] .
  ops in out : -> RPortId [ctor] .
  ops incrementN saveLast saveSecondLast : -> RActionId [ctor] .
  op t  : -> TimerId [ctor] .
  op init : -> Configuration .
  eq init =
    < fib : Reactor | inports : none, outports : < out : Port | value : [0] >,
      state : (n |-> [0]) ; (result |-> [0]) ; (lastResult |-> [0]) ; (secondLastResult |-> [0]),
      timers : < t : Timer | offset : 0, period : 1 >,
      actions : < incrementN : LogicalAction | minDelay : 0, minSpacing : 0, policy : defer, payload : [0] >
                < saveLast : LogicalAction | minDelay : 0, minSpacing : 0, policy : defer, payload : [0] >
                < saveSecondLast : LogicalAction | minDelay : 0, minSpacing : 0, policy : defer, payload : [0] >,
      reactions :
        (reaction when t --> (out ; incrementN ; saveLast ; saveSecondLast) do {
          if (n < [2]) then (result := [1]) else (result := lastResult + secondLastResult) fi ;
          (out <- result) ; schedule(incrementN, [0], [0]) ; schedule(saveLast, [0], [0]) ;
          schedule(saveSecondLast, [0], [0]) })
        (reaction when incrementN  do {(n := n + [1])})
        (reaction when saveSecondLast do {(secondLastResult := lastResult)})
          (reaction when saveLast  do {lastResult := result}) >
```

```
target C
reactor Fib {
  output out:int
  timer t(0, 1 nsec);
  logical action incrementN
  logical action saveLast
  logical action saveSecondLast
  state N:int(0)
  state result:int(0)
  state lastResult:int(0)
  state secondLastResult:int(0)
  reaction(t) -> out, incrementN,
      saveLast, saveSecondLast {=
    if (self->N < 2) {
      self->result=1;
    } else {
      self->result = self->
       lastResult + self->
       secondLastResult; }
    lf_set(out,self->result);
    lf_schedule(incrementN,0);
    lf_schedule(saveLast,0);
    lf_schedule(saveSecondLast,0);
  =}
  reaction(incrementN) {=
    self->N += 1;       =}
  reaction(saveSecondLast) {=
    self->secondLastResult=self->
     lastResult; =}
  reaction(saveLast) {=
  self->lastResult = self->result; =}
}
reactor Printer {
  input in:int
  state result:int
  reaction(in) {=
    self->result = in->value;  =}
}
main reactor {
  fib = new Fib()
  printer = new Printer()
  fib.out -> printer.in
}
```

Fig. 4. LF specification of the Fibonacci system

```
    < printer : Reactor | inports : < in : Port | value : [0] >, outports : none,
      state : (result |-> [0]), timers : none, actions : none,
      reactions : (reaction when in do {result := in}) >
    fib : out --> printer : in .
endom
```

LF Verifier checks the following ("mixed") property: at 10 nanoseconds, the `Printer`'s state variable holds the eleventh Fibonacci number (89), where `G[10 nsec]` specifies the moment when 10 nanoseconds have elapsed:

```
G[10 nsec](Fibonacci_printer_reaction_0 ==> Fibonacci_printer_result == 89)
```

This property can by analyzed using the following Maude command to search for an error state at time 10. Finding no solution demonstrates that the invariant holds.

```
Maude> search initSystem in time 0 =>*
              {< printer : Reactor | state : (result |-> [N]), ATTRS >
               < rxns : Invoked | reactions : (printer . 1) ; RIDS > REST} in time 10 such that N =/= 89  .

No solution.
states: 33  rewrites: 4033 in 9ms cpu (10ms real) (415045 rewrites/second)
```

5.3 Elevator Example

This example models a simple elevator that moves between two floors (Figure 5). It has three buttons (to select a floor or to stop) and three sensors (to detect the current floor and whether the door is closed). A key requirement is that when the stop button is pressed, all other commands are ignored until the stop button is released. Figure 6 shows the LF code for the Elevator example, and the corresponding Maude code is shown below:

```
omod ELEVATOR is including RUNTIME-APG .
  ops floor doorIsOpen stopPressed direction doorState counter1 : -> IVarId [ctor] .
  ops control simulator : -> ReactorId [ctor] .
  ops call1 call2 stop reachFloor doorStatus motorUp motorDown doorCommand : -> IPortId [ctor] .
  ops motorDone checkDoor resetDirection : -> IActionId [ctor] .
  ops call1Pressed call2Pressed simStopPressed : -> TimerId [ctor] .
  op init : -> Configuration .
  eq init =
    < control : Reactor | inports : < call1 : Port | value : [0] >  < call2 : Port | value : [0] >
        < stop : Port | value : [0] >   < reachFloor : Port | value : [0] > < doorStatus : Port | value : [0] >,
      outports : < motorUp : Port | value : [0] >  < motorDown : Port | value : [0] >
        < doorCommand : Port | value : [0] >,
```

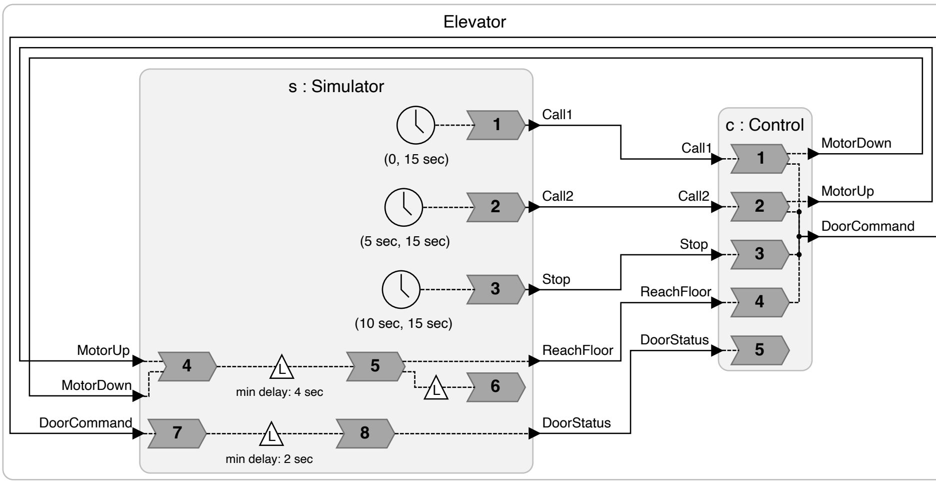


Fig. 5. The Elevator model

```
  state : (floor |-> [0]) ; (doorIsOpen |-> [0]) ; (stopPressed |-> [0]) ; (direction |-> [0]) ;
    (counter1 |-> [0]),
  timers : none,   actions : none,
  reactions :
    (reaction when call1 --> (motorDown ; doorCommand) do { (counter1 :=  [78]) ;
      if ((stopPressed ==/= [1]) && (doorIsOpen ==/= [1]) && (floor ==/= [1]) && (direction === [0]))
      then ( (counter1 := [99]) ; (doorCommand <- [0]) ; (motorDown <- [1])) fi})
    (reaction when call2 --> (motorUp ; doorCommand) do {
      if ((stopPressed ==/= [1]) && (doorIsOpen ==/= [1]) && (floor ==/= [2]) && (direction === [0]))
      then ((doorCommand <- [0]) ; (motorUp <- [1])) fi})
    (reaction when stop --> doorCommand do {
      if (stopPressed === [0]) then ((stopPressed := [1]) ; (doorCommand <- [1]))
      else ((stopPressed := [0]) ; (doorCommand <- [0])) fi })
    (reaction when reachFloor --> doorCommand do {  (floor := reachFloor) ; (doorCommand <- [1]) })
    (reaction when doorStatus do {doorIsOpen := doorStatus})                     >
< simulator : Reactor | inports : < motorUp : Port | value : [0] >  < motorDown : Port | value : [0] >
             < doorCommand : Port | value : [0] >,
  outports : < call1 : Port | value : [0] >  < call2 : Port | value : [0] > < stop : Port | value : [0] >
             < reachFloor : Port | value : [0] > < doorStatus : Port | value : [0] >,
  state : (direction |-> [0]) ; (doorState |-> [0]),
  timers : < call1Pressed : Timer | offset : 0, period : 15 >
           < call2Pressed : Timer | offset : 5, period : 15 >
           < simStopPressed : Timer | offset : 10, period : 15 >,
  actions : < motorDone : LogicalAction | minDelay : 5, minSpacing : 0, policy : defer, payload : [0] >
            < checkDoor : LogicalAction | minDelay : 2, minSpacing : 0, policy : defer, payload : [0] >
            < resetDirection : LogicalAction | minDelay : 0,    minSpacing : 0,
                                             policy : defer,  payload : [0] >,
  reactions :
    (reaction when call1Pressed --> call1 do { (call1 <- [1]) })
    (reaction when call2Pressed --> call2 do { call2 <- [1] })
    (reaction when simStopPressed --> stop do { stop <- [1] })
    (reaction when (motorUp ; motorDown) --> motorDone do {
      if (isPresent(motorUp) && (! isPresent(motorDown)))
      then ((direction := [1]) ; (schedule(motorDone, [0], [0])))
      else (if (isPresent(motorDown) && (! isPresent(motorUp)))
        then ((direction := [2]) ; (schedule(motorDone, [0], [0]))) fi) fi })
    (reaction when motorDone --> (reachFloor ; resetDirection) do {
      if (direction === [1]) then (reachFloor <- [2])
      else if (direction === [2]) then (reachFloor <- [1]) fi fi ;
      schedule(resetDirection, [0], [0])  })
    (reaction when resetDirection do {direction := [0]})
    (reaction when doorCommand --> checkDoor do {(doorState:=doorCommand) ; (schedule(checkDoor, [0], [0]))})
    (reaction when checkDoor --> doorStatus do { doorStatus <- doorState })     >
(control : motorUp --> simulator : motorUp)
```

```
target C

reactor Control {
    input Call1:int
    input Call2:int
    input Stop:int
    input ReachFloor:int
    input DoorStatus:int

    output MotorUp:int
    output MotorDown:int
    output DoorCommand:int

    state Floor:int
    state DoorIsOpen:int
    state StopPressed:int
    state direction:int

    reaction(Call1) -> MotorDown, DoorCommand {=
        if ((!(self->StopPressed == 1) && !(self->DoorIsOpen == 1))
            && self->Floor != 1
            && self->direction == 0)
        {
            lf_set(DoorCommand, 0);
            lf_set(MotorDown, 1);
        }
    =}

    reaction(Call2) -> MotorUp, DoorCommand {=
        if ((!(self->StopPressed == 1) && !(self->DoorIsOpen == 1))
            && self->Floor != 2
            && self->direction == 0)
        {
            lf_set(DoorCommand, 0);
            lf_set(MotorUp, 1);
        }
    =}

    reaction(Stop) -> DoorCommand {=
        if (self->StopPressed == 0) {
            self->StopPressed = 1;
            lf_set(DoorCommand, 1);
        }
        else {
            self->StopPressed = 0;
            lf_set(DoorCommand, 0);
        }
    =}

    reaction(ReachFloor) -> DoorCommand {=
        self->Floor = ReachFloor->value;
        lf_set(DoorCommand, 1);
    =}

    reaction(DoorStatus) {=
        self->DoorIsOpen = DoorStatus->value;
    =}
}

reactor Simulator {
    output Call1:int
    output Call2:int
    output Stop:int
    output DoorStatus:int
    output ReachFloor:int

    input MotorUp:int
    input MotorDown:int
    input DoorCommand:int

    timer Call1Pressed(0, 15 sec)
    timer Call2Pressed(5 sec, 15 sec)
    timer StopPressed(10 sec, 15 sec)

    logical action MotorDone(4 sec)
    logical action CheckDoor(2 sec)
    logical action ResetDirection

    state direction:int(0)
    state doorStatus:int(0)

    reaction(Call1Pressed) -> Call1 {=
        lf_set(Call1, 1);
    =}

    reaction(Call2Pressed) -> Call2 {=
        lf_set(Call2, 1);
    =}

    reaction(StopPressed) -> Stop {=
        lf_set(Stop, 1);
    =}

    reaction(MotorUp, MotorDown) -> MotorDone {=
        if (MotorUp->is_present && !MotorDown->is_present) {
            self->direction = 1;
            lf_schedule(MotorDone, 0);
        }
        else if (MotorDown->is_present && !MotorUp->is_present) {
            self->direction = -1;
            lf_schedule(MotorDone, 0);
        }
    =}

    reaction(MotorDone) -> ReachFloor, ResetDirection {=
        if (self->direction == 1)
            lf_set(ReachFloor, 2);
        else if (self->direction == -1)
            lf_set(ReachFloor, 1);

        lf_schedule(ResetDirection, 0);
    =}

    reaction(ResetDirection) {=
        self->direction = 0;
    =}

    reaction(DoorCommand) -> CheckDoor {=
        self->doorStatus = DoorCommand->value;
        lf_schedule(CheckDoor, 0);
    =}

    reaction(CheckDoor) -> DoorStatus {=
        lf_set(DoorStatus, self->doorStatus);
    =}
}

main reactor {
    c = new Control()
    s = new Simulator()

    c.MotorUp -> s.MotorUp
    c.MotorDown -> s.MotorDown
    c.DoorCommand -> s.DoorCommand
    s.Call1 -> c.Call1
    s.Call2 -> c.Call2
    s.Stop -> c.Stop
    s.ReachFloor -> c.ReachFloor
    s.DoorStatus -> c.DoorStatus
}
```

Fig. 6. LF representation of the Elevator model.

```
    (control : motorDown --> simulator : motorDown)
    (control : doorCommand --> simulator : doorCommand)
    (simulator : call1 --> control : call1)
    (simulator : call2 --> control : call2)
    (simulator : stop --> control : stop)
    (simulator : reachFloor --> control : reachFloor)
    (simulator : doorStatus --> control : doorStatus)    .
endom
```

One of the key safety properties in the Elevator example is that the elevator moves only when it is safe (i.e., the door is closed and the stop button is not pressed). This is represented by the following Safety MTL formula and was verified in [8]:

```
G[0, 15 sec]((Elevator_s_reaction_3 && (Elevator_s_direction != 0)) ==>
            (Elevator_s_doorStatus == 0 && Elevator_c_StopPressed == 0))
```

Since the above property is essentially a time-bounded invariant, we consider a *stronger* invariant invariant without a time bound, encoded as the following unbounded reachability command to search for an error state. Note that the text in blue represents the premise of the invariant, and the condition in `such that` represents the negation of its conclusion. The analysis took around 0005 seconds, exploring 5 synchronous states.

```
Maude> search [1] initSystem =>*
   {< simulator : Reactor | state : ((direction |-> [NZ:NzNat]) ; RS1), AS1 >
    < control : Reactor | AS2, state : (RS2 ; (stopPressed |-> [N3:Nat]) ; (doorIsOpen |-> [N2:Nat])) >
    < rxns : Invoked | reactions : (simulator . 4) ; RIDS > REST} such that N2:Nat =/= 0 or N3:Nat =/= 0 .

No solution.
states: 53  rewrites: 8801 in 4ms cpu (4ms real) (2200250 rewrites/second)
```

5.4 The ADAS Example

The advanced driver assistance system (ADAS), introduced Theorem 2.3, involves both physical actions (the `Pedal` component) and multiple reaction triggers (the `Camera` and `LiDAR` components), and therefore cannot be handled by previous work [8, 12]. Figure 7 shows the LF model,[5] and the corresponding Maude code is shown in Figure 8.

The key (“mixed”) property is that when a stop is requested, the breaks are applied within 55 ms. The LF Verifier formula is as follows, where `ADASModel_l_reaction_0` refers to the invocation of the reaction in `LiDAR`, `ADASModel_p_requestStop` is the `requestStop` variable of the `ADASProcessor` reactor, and `ADASModel_b_brakesApplied` is the `brakesApplied` variable of the `Brakes` reactor:

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

We set `timeBound` to 55, and check the following *stronger* LTL property: whenever `LiDAR` is invoked and `requestStop` in `adasProcessor` is 1, then `brakesApplied` in `brakes` becomes 1 *within* the time bound. This property is stronger than the original formula, since any counterexample to the original is also a counterexample to this property. The following command for time-bounded LTL model checking returns `true` in less than 0.01 seconds, having examined 184 states:

```
Maude> red modelCheck(initSystem in time 0,
                      [] (  (reaction (lidar . 1) invoked /\ requestStop in adasProcessor is [1])
                           -> <> brakesApplied in brakes is [1])  ) .

rewrites: 16085 in 8ms cpu (7ms real) (2010625 rewrites/second)
result Bool: true
```

[5] https://github.com/lf-lang/lf-verifier-benchmarks/blob/main/benchmarks/src/ADASModel.lf

```
target C;
reactor Camera {
  output out: int
  state frame: int = 0
  timer t(0, 17 msec)
  reaction(t) -> out {=
    self -> frame++;
    lf_set(out, frame); =}
}
reactor Lidar {
  output out: int
  state frame: int = 0
  timer t(0, 34 msec)
  reaction(t) -> out {=
    self -> frame++;
    lf_set(out, frame); =}
}

reactor AdasProcessor {
  input in1: int
  input in2: int
  output out1: int
  output out2: int
  state requestStop: bool = false
  logical action a(50 msec): int
  reaction(in1, in2) -> a {=
    self -> requestStop = true;
    lf_schedule(a, 0);  =}
  reaction(a) -> out1, out2 {=
    if (self -> requestStop)
      lf_set(out1, 1);
    else
      lf_set(out2, 4); =}
}
reactor Dashboard {
  input in: int
  state received: bool = false
  reaction(in) {=
    self -> received = true; =}
}
reactor Pedal {
  output out: int
  physical action a
  reaction(a) -> out {=
    lf_set(out, 1); =}
}
reactor Brakes {
  input inAdas: int
  input inPedal: int
  state brakesApplied: int = 0
  reaction(inAdas) {=
    self -> brakesApplied = inAdas
      -> value;
  =}
  reaction(inPedal) {=
    self -> brakesApplied = inPedal
      -> value;
  =}
}
main reactor {
  camera = new Camera()
  lidar = new Lidar()
  adasProcessor = new AdasProcessor
      ()
  brakes = new Brakes()
  dashboard = new Dashboard()
  pedal = new Pedal()
  lidar.out -> adasProcessor.in1
  camera.out -> adasProcessor.in2
  adasProcessor.out1 -> brakes.
      inAdas after 5 msec
  adasProcessor.out2 -> dashboard.
      in
  pedal.out -> brakes.inPedal
}
```

Fig. 7. LF specification of ADAS.

5.5 Train Door Example

We consider a simple train door controller (Figure 10), which consists of three reactors. The goal is to ensure that the door is locked while the train is moving. The LF code is shown in Figure 9, and the corresponding Maude code is provided below.

```
omod TRAINDOOR is including LF-REPR . protecting NAT-LF-TIME .
  ops received : -> RVarId [ctor] .
  ops controller train door : -> ReactorId [ctor] .
  ops in out1 out2 : -> RPortId [ctor] .
  ops startup : -> RActionId .
  op init : -> Configuration .
  eq init =
     < controller : Reactor | inports : none,
       outports : < out1 : Port | value : [0] > < out2 : Port | value : [0] >,
       state : empty, timers : none,
       actions : < startup : LogicalAction | minDelay : 0, minSpacing : 0, policy : defer, payload : [0] >,
       reactions : reaction when startup --> out1 ; out2 do { (out1 <- [1] ) ; (out2 <- [2])}   >
     < train : Reactor | inports : < in : Port | value : [0] >, outports : none,
        state : (received |-> [0]), timers : none, actions : none,
        reactions : reaction when in do { received := in }       >
     < door : Reactor | inports : < in : Port | value : [0] >, outports : none,
        state : (received |-> [0]), timers : none, actions : none,
        reactions : reaction when in do { received := in }            >
     (controller : out1 -- 1 --> train : in)
     controller : out2 -- 1 --> door : in .
endom
```

The key safety property to check in this system is that the train does not move until the door is closed. This is represented by the following formula, which was shown to be false in [8]:

```
(!TrainDoor_t_reaction_0)U[0, 1 sec](TrainDoor_d_reaction_0)
```

```
omod ADAS is including LF-REPR . protecting NAT-LF-TIME . protecting RUNTIME-APG .
  ops frame received brakesApplied requestStop : -> RVarId [ctor] .
  ops lidar camera adasProcessor dashboard pedal brakes : -> ReactorId [ctor] .
  ops in in1 in2 out out1 out2 inAdas inPedal : -> RPortId [ctor] .
  op a : -> RActionId .                  op t : -> TimerId [ctor] .
  op init : -> Configuration .
  eq init =
    < lidar : Reactor | inports : none,  outports : < out : Port | value : [0] >,
      state : (frame |-> [0]),  actions : none, timers : < t : Timer | offset : 0, period : 34 >,
      reactions : reaction when t --> out do {(frame := frame + [1]) ; (out <- frame)} >
    < camera : Reactor | inports : none, outports : < out : Port | value : [0] >,
      state : frame |-> [0],  actions : none, timers : < t : Timer | offset : 0, period : 17 >,
      reactions : reaction when t --> out do {(frame := frame + [1]) ; (out <- frame)} >
    < adasProcessor : Reactor | inports : < in1 : Port | value : [0] >  < in2 : Port | value : [0] >,
      outports : < out1 : Port | value : [0] > < out2 : Port | value : [0] >,
      state : requestStop |-> [0],  timers : none,
      actions : < a : LogicalAction | minDelay : 50, minSpacing : 0, policy : defer, payload : [0] >,
      reactions :
        (reaction when (in1 ; in2) --> a do {(requestStop := [1]) ; schedule(a, [0], [0])})
        reaction when a --> out1 ; out2 do {if requestStop === [1] then (out1 <- [1]) else (out2 <- [4]) fi} >
    < dashboard : Reactor | inports : < in : Port | value : [0] >, outports : none,
      state : received |-> [0],  timers : none,  actions : none,
      reactions : reaction when in do {received := in} >
    < pedal : Reactor | inports : none, outports : < out : Port | value : [0] >,
      timers : none,  state : empty,
      actions : < a : PhysicalAction | minDelay : 0, minSpacing : 0, policy : defer, payload : [0] >,
      reactions : reaction when a --> out do {out <- [1]} >
    < brakes : Reactor | inports : < inAdas : Port | value : [0] > < inPedal : Port | value : [0] >,
      outports : none,  timers : none,  actions : none, state : (brakesApplied |-> [0]),
      reactions : (reaction when inAdas do {brakesApplied := inAdas})
                  reaction when inPedal do {brakesApplied := inPedal} >
     (lidar : out --> adasProcessor : in1)    (adasProcessor : out1 -- 5 --> brakes : inAdas)
     (camera : out --> adasProcessor : in2)   (adasProcessor : out2 --> dashboard : in)
     (pedal : out --> brakes : inPedal) .
endom
```

Fig. 8. Maude representation of ADAS.

```
target C;
reactor Controller {
  output out1:int;
  output out2:int;
  reaction(startup) -> out1, out2
    {= lf_set(out1, 1);
       lf_set(out2, 2);        =}
}
reactor Train {
  input in:int;
  state received:int;
  reaction(in) {=
    self->received = in->value; =}
}
reactor Door {
  input in:int;
  state received:int;
  reaction(in) {=
    self->received = in->value; =}
}
main reactor {
    c = new Controller();
    t = new Train();
    d = new Door();
    c.out1 -> t.in after 1 sec;
    c.out2 -> d.in after 1 sec;
}
```

Fig. 9. LF representation of the TrainDoor controller.

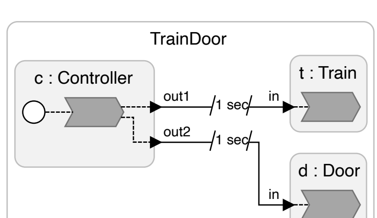


Fig. 10. The TrainDoor controller.

We use the Maude LTL model checker to perform a time-bounded analysis (by setting `timeBound` to 1). The following module defines a state proposition for reaction invocation.

```
omod MODEL-CHECK-TRAINDOOR is including MODEL-CHECKER . including TIME-BOUNDED-TRAINDOOR .
  subsort ClockedSystem < State .
  var REST : Configuration . var RID : ReactionId . var RIDS : ReactionIdSet . var T : Time . var O : Oid .
  op reaction_invoked : ReactionId -> Prop [ctor] .
  eq { REST < O : Invoked | reactions : (RID ; RIDS) > } in time T |=  reaction RID invoked = true .
endom
```

We run the following time-bounded LTL model checking command, which is equivalent to the original formula. The analysis took 0.001 seconds, explored 2 synchronous states, and returned a counterexample:

```
Maude> red modelCheck(initSystem, ~ (reaction (train . 1) invoked) U (reaction (door . 1) invoked)) .

rewrites: 120 in 0ms cpu (1ms real) (~ rewrites/second)
result ModelCheckResult: counterexample(... < rxns : Invoked | reactions : (controller . 1) > ...)
```

5.6 Experimental Results

We compare the performance of our analysis methods with LF Verifier on the LF Verifier benchmark suite. Due to technical issues, we did not re-run the experiments on the LF Verifier, but use the numbers reported in [8], which were conducted on a laptop running macOS version 11.7 with a 2.3 GHz 8-Core Intel Core i9 and 16GB RAM. For a fair comparison, we ran Maude on a laptop running macOS version 10.15.7, with a 2.6GHz dual-core Intel Core i5 and 8GB RAM.

Each example in the benchmark suite targets one of the three property types in safety MTL: state properties (Type I), dataflow properties (Type II), and mixed properties (Type III). We have analyzed equivalent or stronger properties using our framework in Maude, encoded as either unbounded/time-bounded reachability, unbounded/time-bounded LTL, or timed CTL.[6] For timed CTL model checking (e.g., AircraftDoor, Alarm, and Thermostat), we use Real-Time Maude, which runs on an older version of Maude (version 2.7.1).

Table 1 summarizes the experimental results. For the LF Verifier, "Gen." indicates the time taken to generate SMT encodings (including translation to UCLID5), and "Solving" reports the time spent on SMT solving. For Maude, "Time" shows the total analysis time, and "# State" reports the number of the (synchronous) states explored. All execution times are in seconds.

Our analyses drastically outperform those in [8]. Even when considering SMT solving times, Maude achieves much faster analysis than LF Verifier—especially for complex models such as Elevator. The reason is probably that we execute many reactions in one step, resulting in fewer states. Since LF is *deterministic*, it is enough to execute all reactions in a big step in *some* order consistent with the APG, whereas event-based model checkers seem to explore *all* such sequences of reaction executions. This can be exponential in the number of reactions executed.

6 The LF-mc Verification Tool

We have implemented a tool, LF-mc, by extending the LF compiler (`lfc`), which: (i) automatically generates a Maude model from an LF model annotated with intuitive queries, (ii) automatically generates the Maude analysis commands for the user queries, and (iii) invokes Maude to execute the Maude commands on the generated Maude model. LF-mc supports unbounded and time-bounded simulation, reachability analysis, and LTL model checking. LF-mc is available at https://tinyurl.com/mujjr8sz.

[6]In principle, the expressiveness of safety MTL is incomparable to that of (the union of) these property classes. However, all the properties considered in the benchmark suite [8] can be encoded in our setting—sometimes by strengthening the properties, as illustrated in Sections 5.1 to 5.5.

Program (Type)	LF Verifier		**Maude**		Program (Type)	LF Verifier		**Maude**	
	Gen.	Solving	Time	# State		Gen.	Solving	Time	# State
ADAS (III)	21.07	1.72	0.008	184	ProcessMsg (I)	13.23	0.74	0.001	18
AircraftDoor (III)	6.57	0.02	0.107	2	ProcessSync (I)	4.99	0.01	0.001	6
Alarm (III)	5.36	0.01	0.030	4	Railroad (I)	158.14	4.33	0.003	8
CoopSchedule (I)	21.1	0.31	0.002	4	Ring (II)	15.83	0.07	0.001	16
Elevator (III)	325.14	1,038.02	0.005	5	RoadsideUnit (I)	74.07	0.62	0.001	2
Election (I)	41.65	3.27	0.001	6	SafeSend (II)	6.48	0.03	0.001	4
Election2 (I)	12.21	0.17	0.001	6	Subway (II)	29.96	0.31	0.001	5
Factorial (III)	21.46	13.15	0.001	33	Thermostat (I)	13.41	0.14	0.037	2
Fibonacci (III)	94.96	153.49	0.010	33	TrafficLight (I)	-	-	0.001	420
PingPong (I)	13.42	0.14	0.002	22	TrainDoor (II)	5.65	0.01	0.001	2
Pipe (I)	142.41	12.15	0.001	14	UnsafeSend (II)	6.5	0.02	0.001	6

Table 1. LF Verifier vs. Maude

The LF-MC user defines her properties in an intuitive property specification language by decorating her LF program with annotations of the form

```
@maude(analysis="simulation", timeBound=t, rewrites=n)
@maude(analysis="reachability", timeBound=t, rewrites=n, goal="stateFormula")
@maude(analysis="ltl", timeBound=t, goal="LTLformula", type="*")
```

where `timeBound` and `rewrites` (bound on the number of rewrites) are optional. The reachability annotation can also be given an attribute `type="!"` to search for *deadlocked/final* states satisfying the state formula.

A *stateFormula* is a Boolean combination (using `!` or `~` for negation, `/\` for conjunction, `\/` for disjunction, etc.) of *state propositions*. An *LTLformula* extends such state formulas by allowing the user to specify LTL formulas using the temporal operators `[]`, `<>`, `U`, `W` (weak until), and `O` ("next").

State propositions include: *reactor* . *n* `invoked`, which holds if the *n*th [7] reaction in the reactor *reactor* was invoked in the big step leading to the current state; `event(`*reactor*`,` *trigger*`,` *value*`) inQueue`, which holds if the event queue currently contains the event `event(`*reactor*`,` *trigger*`,` *value*`)`; `event(`*reactor*`,` *trigger*`) inQueue`, which is similar but the value of the event is not considered; and *relational expressions* over reactor variables and input ports, which holds if the expression evaluates to *true*.

Relational expressions in state propositions are built from Boolean or integer constants, *var* `in` *reactor* (the value of the state variable *var* in the reactor *reactor*), and *port* `in` *reactor* (the value of the input port *port* in the reactor *reactor*), combined using arithmetic operators (such as + and *), and relational operators (such as ==, !=, >=, >, <=, <). LF-MC automatically checks that such expressions are well-formed: arithmetic and order comparisons (e.g., +, *, >=, >) are permitted only on integer-valued expressions, and equality and inequality comparisons (== and !=) are allowed for both Boolean and integer expressions.

The user should also annotate her LF program to specify how to simulate each physical action: its range of possible values, its period, and whether or not the physical action happens at the end of each period:

```
@maudePhysAct(name="action", inReactor="reactor", vals="values", period=p, timeNonDet=b)
```

[7]Note that LF Verifier indexes reactions beginning at 0, while our interpreter starts from 1

where *values* can be written as an integer interval, e.g., `2 .. 5`, or as a comma-separated list of values, and b is a Boolean value declaring whether the physical action is time-nondeterministic or not. `timeNonDet` is optional, with default value `false`.

Example 6.1. We annotate the LF model of the train door system in Theorem 2.2 with the following annotations:

```
@maudePhysAct(name="external", inReactor="c", vals="true, false", period=10, timeNonDet=true)
@maudePhysAct(name="extOpen", inReactor="d", vals="true", period=11, timeNonDet=true)
@maude(analysis="simulation", rewrites=10)
@maude(analysis="reachability", goal="(locked in d == true) /\ (isOpen in d == true)")
@maude(analysis="reachability", timeBound=150, goal="(locked in d == false) /\ (d.2 invoked)")
@maude(analysis="ltl", goal="<> ((locked in d == false) /\ (d.2 invoked))")
```

The first two define that the physical actions `external` and `extOpen` *may* (`timeNonDet=true`) happen every 10 and 11 time units, respectively, and the other annotations specify simulation, reachability analysis, and LTL model checking commands corresponding to the properties in Section 4. When we execute our `lfc` command with this file as parameter, all the analysis commands are executed on the Maude model which is also generated, and counterexamples are shown in an understandable form:[8]

```
linux> lfc lf-maude/examples/src/TrainDoor_v3.lf
[...]
rewrite [10] in SIMULATION-TRAINDOOR_V3 : initSystem timeBound INF .
rewrites: 411 in 0ms cpu (0ms real) (901315 rewrites/second)
result ClockedSystem: { ... } in time 22000000 timeBound INF
==========================================
search [1] in ANALYSIS-TRAINDOOR_V3 : initSystem timeBound INF =>* {C:Configuration}  timeBound TI:TimeInf
such that {C:Configuration} |= (d.sv.locked in d == [true] /\ d.sv.isOpen in d == [true]) = true .

No solution.
states: 702  rewrites: 46889 in 27ms cpu (27ms real) (1702269 rewrites/second)
==========================================
search [1] in ANALYSIS-TRAINDOOR_V3 : initSystem timeBound 150000000 =>* {C:Configuration} timeBound
TI:TimeInf such that {C:Configuration} |= ((d . 2) invoked /\ d.sv.locked in d == [false]) = true .

Solution 1 (state 188)
states: 189  rewrites: 12875 in 5ms cpu (6ms real) (2206134 rewrites/second)
C:Configuration --> ...
==========================================
reduce in MODELCHECKER-TRAINDOOR_V3 :
modelCheck(initSystem timeBound INF, <> ((d . 2) invoked /\ d.sv.locked in d == [false])) .
rewrites: 2242 in 1ms cpu (1ms real) (1231868 rewrites/second)
result ModelCheckResult: counterexample(
{ invoked: none   queue: empty
  c :[inports: empty  state: empty]
  d :[inports: ...     state: (d.sv.locked |-> [false]) ; d.sv.isOpen |-> [false]]}
===['extraTickRuleForPhysActs]===>
...
===['actionHappens]===>
{ invoked: none   queue: event(d, d.pa.extOpen, [true]) at tag(0, 1)
  c :[inports: empty  state: empty]
  d :[inports: ...     state: (d.sv.locked |-> [false]) ; d.sv.isOpen |-> [false]]}
===['step]===>
{ invoked: d . 5   queue: empty
  c :[inports: empty  state: empty]
  d :[inports: ...      state: (d.sv.locked |-> [false]) ; d.sv.isOpen |-> [true]]}
===['extraTickRuleForPhysActs]===>     ...)
```

7 Related Work

We have described the differences between our contribution and, to the best of our knowledge, all other work providing formal verification for LF [8, 12, 20] quite extensively elsewhere in this paper.

[8] LF-MC assumes that the time bounds in annotations are in milliseconds, and transforms them to nanoseconds, which is the time unit used by the LF compiler.

In [6], Deantoni et al. propose an operational semantics for LF based on Gemoc Studio. They identify domain-specific events of interest (DSEs), such as variable updates, start/finish events, etc., and then define CCSL constraints on these events that should be enforced during execution. They use their implementation primarily for interactive debugging, but if the state space is finite, they can also perform exhaustive simulation to generate all the traces that are amenable to model checking using the CADP model checker. Their operational semantics is defined in terms of sequences of DSEs and does not coincide with the intended "synchronous" semantics of LF.

Rewriting logic has been used to formalize the semantics and to provide formal analysis to a range of programming and modeling languages [3, 16, 17]. The closest to our work is the formalization of Ptolemy II discrete-event models in Maude [2], but not even that work (or any other we know of) execute actions in lockstep with maintaining and updating constraints like we do in this paper.

8 Concluding Remarks

Building on a prototype by Marin et al. [12], we have formalized in rewriting logic the intended "discrete-event" semantics of a large fragment of the Lingua Franca (LF) coordination language for cyber-physical systems. While Marin et al. [12] do not support multiple action triggers, and therefore do not capture most LF models, we capture a larger subset of LF than all previous verification approaches for LF [8, 12, 20]. This subset includes all 22 LF Verifier benchmarks and all examples in [8, 12, 20]. The work in [8, 20] does not consider (nondeterministic) physical actions. In such a setting, all models are deterministic (after all, the goal of LF is to provide "deterministic concurrency"). In contrast, we support a quite general nondeterministic model of physical actions.

Our formalization of the LF semantics is executable and can be used to analyze LF models using the Maude and Real-Time Maude tools. The work in [12] does not support *timed* (or *metric*) temporal logic or "dataflow" properties, and can therefore not perform the analyses in [8], which relied heavily on such properties. We support both unbounded and time-bounded simulation, reachability analysis, LTL model checking, and timed CTL model checking, for state properties, dataflow properties, and mixed properties, and can therefore analyze all LF verification benchmarks.

The efforts in [8, 20] provide "small-step" event-based semantics for LF, which (i) is not the desired semantics of LF, and (ii) while they model check only deterministic models, they analyze all interleavings of the small steps, leading to inefficient analyses. Since we capture the deterministic big-step semantics, our model checking of *physical-action-less* LF models only considers a single path. Benchmarking the 22 LF models shows the dramatic difference in performance: all our analyses take 0.1 seconds or less, whereas LF Verifier analysis can take more than 20 minutes.

This work shows the advantage of a computational logic such as rewriting logic—with *rewrite rules* defining transitions and *equations* defining functions (e.g., to define a single big step)—over pure event-based languages, such as Timed Rebeca, and over SMT encodings to formalize the semantics of what is a very complex language. The latter also only supports *bounded* model checking, whereas we provide both bounded and unbounded model checking.

Future work includes: (i) providing *symbolic* analysis methods, to support physical actions that may take place at *any* time; (ii) formalizing the two-timeline semantics of LF; and (iii) automate the translation from LF and integrate Maude verification of LF models into the LF tool chain.

Lifting our view above LF, our formalization seems to provide a *general technique* for executing a set of actions whose execution order must satisfy a "strategy" or "constraint" (like the APG for LF) which, furthermore, may change dynamically. Likewise, our way of enabling analyzing action-based (or "dataflow") properties is a simple general method.

References

[1] Luca Aceto, Matteo Cimini, Anna Ingólfsdóttir, Arni Hermann Reynisson, Steinar Hugi Sigurdarson, and Marjan Sirjani. 2011. Modelling and Simulation of Asynchronous Real-Time Systems using Timed Rebeca. In *10th International Workshop on the Foundations of Coordination Languages and Software Architectures (FOCLASA 2011) (EPTCS, Vol. 58)*. 1–19.

[2] Kyungmin Bae, Peter Csaba Ölveczky, Thomas Huining Feng, Edward A Lee, and Stavros Tripakis. 2012. Verifying hierarchical Ptolemy II discrete-event models using Real-Time Maude. *Science of Computer Programming* 77, 12 (2012), 1235–1271.

[3] Xiaohong Chen and Grigore Roşu. 2020. 𝕂—A Semantic Framework for Programming Languages and Formal Analysis. In *5th International School on Engineering Trustworthy Software Systems (SETSS 2019)*. Springer International Publishing, Cham, 122–158.

[4] Manuel Clavel, Francisco Durán, Steven Eker, Santiago Escobar, Patrick Lincoln, Narciso Martı-Oliet, José Meseguer, Rubén Rubio, and Carolyn Talcott. 2020. *Maude Manual (Version 3.1)*. Technical Report. SRI International, Menlo Park. http://maude.cs.illinois.edu/w/index.php/Maude_Manual_and_Examples

[5] M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, and C. Talcott. 2007. *All About Maude – A High-Performance Logical Framework*. Lecture Notes in Computer Science, Vol. 4350. Springer, Berlin, Heidelberg.

[6] Julien Deantoni, João Cambeiro, Soroush Bateni, Shaokai Lin, and Marten Lohstroh. 2021. Debugging and Verification Tools for Lingua Franca in Gemoc Studio. In *2021 Forum on specification & Design Languages (FDL)*. Antibes, France, 01–08. doi:10.1109/FDL53530.2021.9568383

[7] Daniela Lepri, Erika Ábrahám, and Peter Csaba Ölveczky. 2015. Sound and complete timed CTL model checking of timed Kripke structures and real-time rewrite theories. *Science of Computer Programming* 99 (2015), 128–192.

[8] 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 Transactions on Embedded Computing Systems* 22, 5s (2023), 1–24.

[9] Lingua Franca page [n. d.]. https://www.lf-lang.org/. Accessed October 16, 2025.

[10] Marten Lohstroh. 2020. *Reactors: A Deterministic Model of Concurrent Computation for Reactive Systems*. Ph. D. Dissertation. EECS Department, University of California, Berkeley.

[11] 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)* 20, 4 (2021), 1–27.

[12] Mircea Marin, Peter Csaba Ölveczky, Mario Reja, Mikheil Rukhaia, and Kyungmin Bae. 2025. Semantics and Formal Analysis of Lingua Franca CPS Specifications in Rewriting Logic. In *Rebeca for Actor Analysis in Action*. Lecture Notes in Computer Science, Vol. 15560. Springer.

[13] Christian Menard, Marten Lohstroh, Soroush Bateni, Matthew Chorlian, Arthur Deng, Peter Donovan, Clément Fournier, Shaokai Lin, Felix Suchert, Tassilo Tanneberger, et al. 2023. High-performance deterministic concurrency using Lingua Franca. *ACM Transactions on Architecture and Code Optimization* 20, 4 (2023), 1–29.

[14] José Meseguer. 1992. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theor. Comput. Sci.* 96, 1 (1992), 73–155.

[15] José Meseguer. 1997. Membership algebra as a logical framework for equational specification. In *Recent Trends in Algebraic Development Techniques (WADT'97) (LNCS, Vol. 1376)*. Springer, 18–61.

[16] José Meseguer and Grigore Rosu. 2007. The rewriting logic semantics project. *Theoretical Computer Science* 373, 3 (2007), 213–237.

[17] Peter Csaba Ölveczky. 2011. Semantics, Simulation, and Formal Analysis of Modeling Languages for Embedded Systems in Real-Time Maude. In *Formal Modeling: Actors, Open Systems, Biological Systems (Lecture Notes in Computer Science, Vol. 7000)*. Springer, 368–402.

[18] Peter Csaba Ölveczky. 2014. Real-Time Maude and Its Applications. In *Proc. WRLA'14 (LNCS, Vol. 8663)*. Springer.

[19] Peter Csaba Ölveczky and José Meseguer. 2002. Specification of real-time and hybrid systems in rewriting logic. *Theor. Comput. Sci.* 285, 2 (2002), 359–405.

[20] Marjan Sirjani, Edward A Lee, and Ehsan Khamespanah. 2020. Verification of cyberphysical systems. *Mathematics* 8, 7 (2020), 1068.