Debugging and Verification Tools for LINGUA FRANCA in GEMOC Studio

Julien Deantoni
and João Cambeiro
Université Cote d'Azur
CNRS/I3S/INRIA Kairos, France
Email: {first.last}@univ-cotedazur.fr

Soroush Bateni
UT Dallas
Texas, USA
Email: soroush@utdallas.edu

Shaokai Lin
and Marten Lohstroh
UC Berkeley
California, USA
Email: {shaokai, marten}@berkeley.edu

***Abstract*—LINGUA FRANCA (LF) is a polyglot coordination language designed for the composition of concurrent, time-sensitive, and potentially distributed reactive components called reactors. The LF coordination layer facilitates the use of target languages (*e.g.,* C, C++, Python, TypeScript) to realize the program logic, where each target language requires a separate runtime implementation that must correctly implement the reactor semantics. Verifying the correctness of runtime implementations is not a trivial task, and is currently done on the basis of regression testing. To provide a more formal verification tool for existing and future target runtimes, as well as to help verify properties of LF programs, we recruit the use of GEMOC Studio—an Eclipse-based workbench for the development, integration, and use of heterogeneous executable modeling languages. We present an operational model for LF, realized in GEMOC Studio, that is primed to interact with a rich set of analysis and verification tools. Our instrumentation provides the ability to navigate the execution of LF programs using an omniscient debugger with graphical model animation; to check assertions in particular execution runs, or exhaustively, using a model checker; and to validate or debug traces obtained from arbitrary LF runtime environments.**

I. INTRODUCTION

Cyber-physical systems (CPSs) are becoming increasingly complex, built using an intricate composition of complicated software, hardware, and networked systems. This complexity in design inherently makes CPS development difficult and error-prone. Consequently, the challenges associated with managing concurrency, timing, and distributed computing also multiply proportionally. To tackle these challenges, very recently, a comprehensive solution is proposed in the form of LINGUA FRANCA (LF) [1], a polyglot coordination language designed for the deterministic composition of concurrent, time-sensitive, and potentially distributed reactive components called reactors. LF is aimed at simplifying the design and implementation of CPSs through a programming model with sophisticated runtime support that delivers deterministic behavior under quantifiable assumptions.

LF provides a coordination layer based on reactors [2], [3], a model of computation that combines aspects from actors, discrete-event systems, and synchronous languages. This coordination layer is meant to complement mainstream programming languages. So far, LF supports C, C++, Python, and TypeScript as target languages, each of which requires a dedicated code generator and a supporting runtime library. As LF becomes more popular and adds support for more target languages, two issues arise.

First, the underlying code generators and runtime libraries can become tedious to debug. Currently, each code generator and runtime library is subjected to an extensive suite of regression tests that verify strict compliance with the reactor semantics and the LF syntax. However, if a regression test fails, there is not much to go by other than the end result of the test. During the development phase of a new target (or while adding new language features) it may be helpful to the developers to have an easy way to compare the execution trace of a failing test against the set of valid traces (*i.e.,* linearizations of one partial-order trace) permitted by the reactor model. Such capability can help reveal *how* and *why* a certain violation of the reactor semantics has occurred.

Second, from a higher-level user perspective, debugging an LF program is currently only possible on the basis of generated code, by executing it using the debugging facilities of the target language (e.g., gdb for C), or with the help of print statements. But this tends to be cumbersome—particularly for lower-level languages like C—because the generated code obfuscates the high-level structure of the program.

To enable a more sophisticated debugging functionality in LF, one could look at existing approaches that are developed for other domain-specific languages (DSLs) that establish a mapping between the DSL and the implementation language [4], [5]. However, since LF is polyglot, the substantial task of creating such a mapping would have to be repeated for each individual target language. Hence, we pursue a more flexible design that is nonetheless capable of integrating into the LF IDE.

Contributions. As a starting point for our debugging framework, we use GEMOC Studio [6]. We then implement the two following methods to debug an LF program:

*a) **Interactive debugger**:* We directly weave an abstract execution model of LF into its syntactic definition, so that LF reaction code (assumed to be written in Groovy for our proof-of-concept implementation) can be interpreted through this abstract model. More precisely, we use the GEMOC MOCCML approach, which relies on a *clock calculus* to support the definition of concurrent and timed operational semantics.

author version

Based on this, we implement a rich debugging environment with the possibility to go forward and backward in time, as well as to inspect the state of the LF program directly at the LF level of abstraction. Our developed tooling also provides model animation and supports exhaustive simulation of the program.

b) ***Trace debugger****:* We add the possibility to inject execution traces obtained from the execution of compiled LF programs directly in the development environment. This enables target runtime developers to validate execution traces that are obtained through a standardized tracing interface against the LF semantics, and detect any deviation thereof from the correct behavior. Unlike the interactive debugger, which requires reaction code to be interpreted, this feature is compatible with any target that has a runtime capable of producing execution traces.

We provide a brief overview of LF and the GEMOC MOCCML approach in Section II. We discuss the operational semantics of LF and its implementation in GEMOC in Section III. We demonstrate the usage of our tooling in Section IV. Finally, we position our contributions with respect to related work in Section V and conclude in Section VI.

II. BACKGROUND

A. Lingua Franca

LINGUA FRANCA (LF) [1] is a new polyglot coordination language aimed to simplify the design and implementation of concurrent, time-sensitive, and distributed software. It provides supported target languages with a semantic notion of time, deterministic concurrency, and transparent exploitation of parallelism. LF is based on the reactor model [2], [3], in which components—homonymously called *reactors*—are composed under a discrete-event semantics. Reactors follow much of the style of actors [7], with the crucial difference that communication between reactors is timestamped, and deterministic unless specified otherwise. The goal of LF is to relegate the many difficulties associated with concurrent programming [8] as much as possible to the supporting runtime system, away from the program logic.

Reactors have ports which allow them to be connected to other reactors. Beside connections, reactors can also be composed hierarchically (through containment). The functionality of a reactor is expressed through its reactions, which execute in response to the presence of one or more triggers. A triggering event is always associated with a tag that denotes the current logical time. Any outputs that a reaction produces will have the same tag of the event that triggered it. Hence, reactions are logically instantaneous, following principles of synchronous programming [9].

In LF, reactions are written in verbatim target code. Each of these target code segments is treated by the LF compiler as a "black box." The compiler merely inserts them into the appropriate locations within the target code it generates on the basis of the composition of reactors specified by the LF program. All that matters to the compiler and the runtime system are the dependencies between reactions, which inform the orders in which reactions can be executed without breaking the reactor semantics. This approach allows the runtime system to transparently exploit parallelism, without any engineering effort or hints in the form of pragmas needed from the application developer.

B. The GEMOC MoCCML approach

GEMOC studio [6] is an Eclipse project[1] that provides a set of metalanguages for the definition of the operational semantics of programming languages. Specifically, GEMOC studio supports languages that are specified using the Eclipse Modeling Framework (EMF) [10], which is a modeling framework and code generation facility for building language tools and applications based on a structured data model called an Ecore model. The current implementation of LF is based on Xtext[2], a framework that automatically generates an Ecore model based on a textual language definition (an Xtext grammar). In GEMOC studio, the Ecore model, which merely captures the syntactic aspects of the language, can be complemented with a Model of Concurrency and Communication (MOCC) [11], [12], [13] to define an execution semantics with formal support of concurrency and timing. In this approach, the operational semantics is defined by four different artifacts:

1) The runtime program state, which represents the state of the program during execution. Inspired by the approach of attribute grammars [14], the runtime state is woven directly into the concepts of the languages (*i.e.,* into the Ecore model) by using Kermeta 3 (K3)[3]—an action language (*i.e.,* a language for specifying state transitions systems) primarily designed for implementing the execution semantics of Ecore metamodels.
2) A set of rewriting rules, which manipulate the runtime program state. These are also woven into the Ecore model and defined imperatively using K3.
3) A set of Domain Specific Events (DSEs), which are events of interest in the concurrent and timed execution semantics of the language. They are part of the MOCC and define the observable atomic actions in an execution trace. Some of these events are associated with a rewriting rule and act as handle on the rewriting rules call. Like other artifacts, they are also woven into the Ecore model by using a mapping provided by the MOCCML language [15]. The mapping also contains behavioral invariants, which define ordering constraints that must be respected for any execution, and use constraints as defined in the next item.
4) A set of formal domain-specific constraints, put in conjunction, to be enforced during execution. These constraints induce a partial order of the DSE instances. They are specified by using CCSL constraints as defined originally [16], or by using constraint automata introduced in MOCCML [12]. Such constraints allows expressing mixtures of synchronous and asynchronous

[1]https://eclipse.org/gemoc/
[2]http://eclipse.org/xtext/
[3]https://diverse-project.github.io/k3/

author version

ordering requirements on DSEs; enabling the symbolic specification of partially ordered sets of clock ticks, which is well-suited for the description of large sets of model control flow [17].

For a specific program conforming to the abstract syntax, the DSEs (and the associated constraints) are instantiated in a classical object-oriented way: for each instance of a concept in a program, the DSEs that were woven in the concept are instantiated. MoCCML and CCSL are based on the notion of logical time [18], [19], which was originally designed for distributed and concurrent systems, but which was also used in synchronous languages. They generalize different descriptions of time, based on the notion of *clocks*—a clock being an ordered set of instants. A DSE instance is then a *clock*.

Solving an instance of a MoCCML model (*i.e.,* doing a run) results in a *schedule*. A schedule over a set of clocks is a possibly infinite sequence of *steps*, where a step is a set of ticking clocks. For each step, one or several clock(s) can tick depending on the imposed constraints. Note that since MoCCML describes partial orders, there possibly exist various schedules for a single specification, which represent different acceptable interleavings of actions (*i.e.,* reaction invocations in the LF program). The operational semantics of a MoCCML model [12] specifies how to construct acceptable schedules step-by-step, and is given as a mapping to a Boolean expression on Boolean variables in bijection with the clocks. In a step, a boolean variable valuated to *true* means that the clock ticks; and if valuated to *false* that the clock is absent. Each time a constraint is added to the specification, it adds Boolean constraints, which depend on the definition of the constraint and its internal state. Boolean expressions are put in conjunction so that each added constraint reduces the set of acceptable schedules (also referred to as a trace containment according to [20]).

Instances of MoCCML models are inputs to the TimeSquare tool suite [21], which supports their simulation and is part of the Gemoc engine in the MoCCML approach. Each time a clock ticks, the associated rewriting rule is applied. If various clocks tick during a single step, then the rewriting rule application order is undefined and should consequently not be significant. It is possible to define a data dependent concurrency model (which amounts to the symbolic partial order being pruned at runtime according to the program runtime state [22]). This is used, for instance, to specify the control flow of a classical `if-then-else` statement. Depending on the evaluation of the branch predicate (the outcome of which depends on the model state), either the `then` or the `else` branch is taken.

To make a complete definition of the concurrency model, it is also possible to define *priorities* between the different DSEs [23]. Intuitively, for a specific program, if (and only if) according to the constraints, two clocks cannot tick together in the same step, then the clock with the highest priority is chosen at that step.

From the definition of the concurrent and timed operational semantics in MoCCML, Gemoc studio offers an omniscient debugger [24] (to go forward and backward in the execution trace and explore new interleaving) and an assertion checker (to check if some constraints are enforced or not during the execution). Extensions realized in the context of this paper also provides an integration with the CADP model checker[25]. Finally any Gemoc engine can use *addons* and we developed one for graphical model animation (to help understand the execution).

III. Operational Semantics of Lingua Franca

A. Intuitive Description

The semantics of LF is a hybrid between 1) a discrete event model with support for superdense time [26] and 2) a dataflow model to handle concurrency and potential interleaving. Consequently, a run of an LF program evolves around the following two notions.

1) temporal dependencies: These are induced by timed concepts such as *timers* or *actions* and *connections* that have a specified delay. Each synchronous-reactive tick is associated with a logical time represented by a *tag* (*i.e.,* a pair of integers that represent a *time value* and a *microstep*). A new event is enqueued with a tag strictly greater than the current tag. All events are handled in tag order. Each event is tied to a particular trigger that determines which reaction(s) it activates. Time advancement (*i.e.,* reassignment of the current tag such that it is equal to the tag of the event on top of the event queue) cannot occur before all events for the current tag have been handled, and all triggered reactions have executed. Reaction code can schedule a future event to occur after a specified delay through invocation of the `schedule` routine (*e.g.,* `schedule(action1, 10msec)` where `action1` has to be part of the same reactor and has to be declared as an effect of the reaction). Note that an action can also be scheduled with zero delay, in which case, the resulting event will have a tag with the same time value as the current logical time, but the microstep will be incremented by one.

2) data dependencies: Defined between *reactions*, they impose ordering constraints on the invocation of reactions during a particular synchronous-reactive tick. A data dependency between two reactions can either be specified explicitly by ordering the reactions inside a single reactor (allowing them to operate on shared state), or by establishing a connection between two ports of distinct reactors (one of which being an *effect* of the first reaction, the other being a *source* of the second reaction). These statically known data dependencies are conservative approximations of the actual data dependencies that exist between reactions at runtime; whether reaction code invokes `set` on a port may depend on the reactor's state and current inputs. A reaction r_1 with an effect y that is connected to a downstream port x known as the trigger of another reaction r_2 will cause r_2 to be triggered if it calls `set` on y. But r_1, even if triggered, has no obligation to produce an output (and thus r_2 may not execute, even if r_1 does). This is why the reactor model can also be characterized as a "sparse synchronous" model [27].

Author Version

While temporal dependencies are concerned with ordering events on a logical timeline, data dependencies are concerned with partially ordering reactions that occur at the same tag. Time advancement acts as a synchronization barrier for the execution of reactions at a particular tag. As such, we could say that temporal dependencies apply at a coarser grain than data dependencies. Another way to view the interaction between the two is that the preservation of data dependencies has a *higher priority* than time advancement.

B. Encoding in GEMOC

This section outlines the encoding of the LF operational semantics in terms of the four artifacts introduced in section II-B. The entire encoding can be obtained from GitHub[4].

1) Runtime Program State: The runtime state σ of an LF program is determined by the following tuple: $\{P_{pcv}, S_{scv}, A_{buf}, \mathcal{Q}_E\}$ where P_{pcv} is the set of current values pcv associated with elements in P, the set of ports in the program; S_{scv} is the set of current values scv associated with elements in S, the set of state variables in the program; A_{buf} is the data buffer associated with elements in A, the set actions in the program (including those implied by connections with a specified delay); and $\mathcal{Q}_E$ is a global priority queue.

In our K3 code, $\mathcal{Q}_E$ is referred to as `eventQ`, and its elements are `STA` (scheduled time advancement) objects. An `STA` instance represents a scheduled event, identified by a reference to a timed concept (either an action, a timer or a timed connection) and the tag (relative to the current time) at which it must be released. Scheduled events are always stored in an increasing tag order. Furthermore, pcv is represented by `currentValue`, which is woven into the `Variable` aspect, a superclass of `Port` (see Listing 1). When `currentValue` equals `null`, it is considered *absent* (and *present* otherwise). The type of `currentValue` is `ClonableObject`, so that it can be copied. This is required by the framework to allow the omniscient debugger to do backward navigation in time and to construct the state space of the program.

```
1  @Aspect(className=Variable) class VariableAspect{
2    public ClonableObject currentValue
3  }
```

Listing 1. Implementation of the pcv runtime state

A similar mechanism is used to encode S_{scv} and A_{buf}. $\mathcal{Q}_E$ is woven into the root concept of the program (*i.e.,* `Model`) and encoded as an `EventQ`, an extension of `List<STA>`. The current tag is also woven in the root concept of the program (lines 4 and 6 of Listing 2). However, this runtime data should be excluded from the state space (to avoid infinite state space since time increases monotonically)—this is the purpose of the `@NotInStateSpace` annotation. The GEMOC framework automatically generates the appropriate getters and setters for the elements of the runtime state of the program that are needed for omniscient debugging and state space construction.

```
1  @Aspect(className=Model) class ModelAspect {
2    public var eventQ eventQ=new EventQ()
3    @NotInStateSpace
4    public var Integer currentTime = 0
5    @NotInStateSpace
6    public var Integer currentMicroStep = 0
7      [...]
8  }
```

Listing 2. Implementation of the $\mathcal{Q}_E$ runtime state

[4]https://github.com/jdeantoni/LinguaFrancaInGemoc/

2) Rewriting Rules: The rewriting rules typically modify the runtime program state. For instance, there are two *schedule* rewriting rules in charge of adding a timed action at the appropriate place in the event queue:

$$\frac{\langle\langle rt, tc\rangle \in \mathcal{Q}_E, \sigma\rangle \Rightarrow false}{\sigma \rightsquigarrow (\sigma \biguplus \mathcal{Q}_E \cup \{\langle rt, 0, tc\rangle\})} \quad (1)$$

$$\frac{\langle\langle rt, tc\rangle \in \mathcal{Q}_E, \sigma\rangle \Rightarrow true}{\sigma \rightsquigarrow (\sigma \biguplus \mathcal{Q}_E \cup \{\langle rt, max\mu s_{\langle rt,tc\rangle} + 1, tc\rangle\})} \quad (2)$$

where $\langle rt, tc\rangle$ is a tuple for a timed concept tc to be scheduled at the (relative) tag rt; where $\langle rt, X, tc\rangle$ is the scheduled time advancement of tc at the tag rt and microstep X and where $max\mu s_{\langle rt,tc\rangle}$ is the maximum microstep index of the existing scheduled time advancement in $\mathcal{Q}_E$ corresponding to the timed concept tc at time rt.

These two rewriting rules are encoded as shown in Listing 3. The $max\mu s_{\langle rt,tc\rangle}$ is stored in `lastMS` (declared on line 7) and computed along the loop starting on line 8 in case $\langle rt, tc\rangle \in \mathcal{Q}_E$ (tested on line 10). The remainder of the code implements the $\mathcal{Q}_E \cup \{\langle rt, X, tc\rangle\}$ ensuring the priority order in the queue and the computation of X.

```
1  def void schedule(TimedConcept tc, int rt) {
2    if(_self.eventQ.isEmpty) {
3          // Insert new scheduled time advancement
4          _self.eventQ.add(new STA(tc, rt, 0))
5      return
6    }
7    var int lastMS = -1;
8    for(var int i = 0; i < _self.eventQ.size; i++) {
9      var sta= _self.eventQ.get(i);
10     if(sta.timedConcept==tc && sta.releaseTag==rt) {
11       // Already exists
12       lastMS = staAtI.microStep
13     }
14     if(sta.releaseTag > rt) {
15       var actualMS = (lastMS == -1) ? 0 : lastMS+1;
16       _self.eventQ.add(i, new STA(tc, rt, actualMS))
17       return
18     }
19   }
20   _self.eventQ.add(new STA(tc, rt, 0))
21   return
22 }
```

Listing 3. The schedule rewriting rule, merging (1) and (2)

Following the same principle, the other rewriting rules have been developed, and most of them are simple. One rewriting rule is a bit more elaborate: the one in charge of executing the code defined in a reaction. Note that since this is encoded in a rewriting rule, it is seen as atomic from a MoCC point of view. Listing 4 provides the code implementing the reaction execution rewriting rule. The reaction code is encoded in the Groovy language (https://groovy-lang.org/). The rewriting rule is in charge of putting the appropriate context variables in the scope of the Groovy code (lines 3 to 5) and to actually execute the code (lines 7 to 9). Note that in addition to the code, we prefix it with the definition of LF specific functions (line 9) to make them available in the reaction code (*e.g.,* the `set` or the `schedule` routines). Finally, we check (from lines 11 to 18)

author version

what output ports have been set and what actions have been scheduled. Finally, on line 19, triggers are reset to `null` (*i.e.,* absent).

```
1  @Aspect(className=Reaction) class ReactionAspect {
2    def void exec(){
3      var Binding binding = new Binding()
4      binding.setVariable("self", _self )
5      var Context c = bindReactionContext(binding)
6
7      val ucl = ReactionAspect.classLoader
8      val shell = new GroovyShell(ucl,binding)
9      shell.evaluate(lfGroovyFunctions+_self.code.body)
10
11     for(VarRef vRef : _self.effects) {
12       if (c.outAssignements.containsKey(vRef.variable)){
13         vRef.variable.currentValue = c.outAssignements.get(vRef.
               variable)
14       } else { vRef.variable.currentValue = null }
15       if (c.newSchedules.containsKey(vRef.variable)){
16         (vRef.variable as Action).nextSchedule = c.newSchedules.
               get(vRef.variable) as Integer
17       }
18     }
19     resetTriggerValuesToNull();
20   (...)
```

Listing 4. The reaction execution rewriting rule

In addition to the rewriting rules, we define some predicate functions on the runtime state, which are used in the MoCC to define data-dependent control. For instance, in Listing 5, the `isPresent()` routine is defined in the context of a particular port or action (instance of `Variable`) and reports whether it is present or absent.

```
1  @Aspect(className=Variable) class VariableAspect{
2    def Boolean isPresent(){
3      return _self.currentValue !== null
4    }
5  }
```

Listing 5. Part of the program runtime state

We have defined a total of 16 rewriting rules. It is important to notice that there is no control flow between these rewriting rules. The two following subsections define the DSEs that act as handles on the invocations of rewriting rules and MoCCML constraints that define the partial order(s) in which rewriting rules must be invoked.

3) Domain Specific Events: The DSEs represent the observable atomic actions in the traces produced by the semantics. As such, it encompass small/big steps semantics. An excerpt of the DSE definition is given in listing 6. For instance, in the context of a `Reaction` (line 1 to 3), we can observe the start and the finish of the execution. We can also notice that the `start` DSE is associated with the `exec()` rewriting rule explained in the previous subsection. Looking at the `Variable` DSE, we reified `present`, which occurs when the variable has been set, and `absent`, which occurs when it has not been set. Finally, `updates` occurs each time a variable is present or absent. We can see that there exists an `isPresent` DSE, which is used to define how the MoCC is pruned depending on the actual runtime state of the system. It calls the `isPresent()` predicate, and according to the result forbids either `present` or `absent` (lines 8 to 11). A similar mechanism determines whether a timed concept can be released or not (not shown in the listing). Finally, we reified two types of actions at the root concepts (see lines 13 to 14 of listing 6): `atFutureTag`, whose instance occurrences represent a call to the `atFutureTag()` rewriting rule when a timed concept releases (see III-A1); `atCurrentTag`, which represents events that happen at the current tag, such as ports being set and reactions being invoked (see III-A2). This reification has been done to allow the definition of priorities between them, defining the high level order of actions in a LF program, *i.e.,* `atCurrentTag` events have a higher priority than `atFutureTag` events.

```
1  context Reaction
2    def: startExecution  : Event = self.exec()
3    def: finishExecution : Event = self
4  context Variable
5    def : updates : Event = self.updates()
6    def : absent  : Event = self
7    def : present : Event = self
8    def if (self.isOutputPort()): isPresent : Event = self.
         isPresent()
9      [result] switch
10       case (result = true)  forbid absent  until updates;
11       case (result = false) forbid present until updates;
12 context Model
13   def : atFutureTag : Event = self.atFutureTag()
14   def : atCurrentTag : Event = self
```

Listing 6. Some of the DSEs for LF

4) Model of Concurrency and Communication: After the DSEs have been defined on the language, we specified how they are constrained. This is done by reusing existing constraints as defined by CCSL and by defining some domain-specific constraints based on MoCCML automata. Constraining the DSEs is done by defining behavioral invariants in MoCCML, which define when and how to apply the constraints.

In the following, we reuse the CCSL `Causes` constraint, which takes two DSEs (clocks c_1 and c_2) as arguments and specifies that $\forall k \in \mathbb{N}$, the k^{th} instant of c_1 to coincide with or precede the k^{th} instant of c_2.

In Listing 7, the `Causes` constraint is applied in the context of a `Connection` (line 1) in case it has no delay specified (line 3). In this case, the first argument of the `Causes` constraint is the presence of data in the left port (*i.e.,* the source port), and the presence of a data on the right port (*i.e.,* the target port), as defined in line 3. The same pattern is used to specify that the absence of data is propagated through a `Connection`.

```
1  context Connection
2    inv ConnectionSourceCausesTargetForPresent:
3      (self.delay = null) implies
4      Relation Causes(self.leftPorts.variable.present, self.
           rightPorts.variable.present)
5    inv ConnectionSourceCausesTargetForAbsent:
6      (self.delay = null) implies
7      Relation Causes(self.leftPorts.variable.absent, self.
           rightPorts.variable.absent)
```

Listing 7. Causalities definition for untimed Connections

In case the `Connection` has a non-zero delay, meaning that it goes through the event queue instead of propagating downstream directly[5], we use combination of a CCSL constraint and a MoCCML automaton, which is defined as represented in Figure 1. In the initial state (idle) the connection can be started, which results in the scheduling of its associated action (see the `schedule()` function in Listing 3). `starts` can be done several times. Then, when a time advancement

[5]In LF, a connection with a delay is syntactic sugar for a reactor that outputs a given input with the specified delay by scheduling an action that it subsequently reacts to.

author version

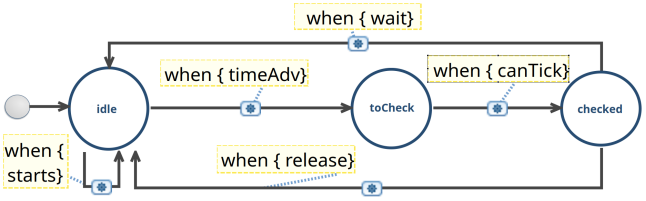

Fig. 1. MoCCML automaton for `Connection` life cycle

occurs, the connection enters the toCheck state, waiting to check if it can be released or not[6]. When the check is performed, the connection goes in the checked state. There, two outgoing transitions are possible. Either the connection releases, or it waits. These transitions are to be put in relation with the connection between the MoCC and the runtime state of the system. In short, the *canTick* event is associated function to a predicate that checks in the event queue if the connection can release or not. This forbids the related DSE, allowing only one of the checked state outgoing transitions to be taken. This constraint is applied in Listing 8, lines 2 to 4. This automaton is not sufficient and causalities must be defined between some of the DSEs used in the automaton and the ones from the source and target of the connection. This is done by again using the `Causes` constraint (see lines 5 to 14). In short, the presence of a data in the source port starts the connection (schedules it), when the connection releases it, this causes the presence of a data in the target port and when the source port is absent, or, when the connection waits, it causes an absence in the target port.

```
1  context Connection
2  inv TimedConnectionConstraints
3    (self.delay <> null) implies
4    Relation Connection(self.starts, self.canRelease, self.wait,
         self.releases, theModel.atFutureTag)
5  inv TimedConnectionStartsWithSourcePresence:
6      (self.delay <> null) implies
7      Relation Causes(self.leftPorts.variable.present, self.starts)
8  inv TimedConnectionReleaseCausesTargetPresence:
9      (self.delay <> null) implies
10     Relation Coincides(self.releases, self.rightPorts.present)
11 inv TimedConnectionAbsentOrWaitCausesTargetAbsent:
12   (self.delay <> null) implies
13   let waitOrAbsent : Event = Expression Union(self.wait, self.
         leftPorts.variable.absent) in
14   Relation Causes(waitOrAbsent, self.rightPorts.variable.absent)
```

Listing 8. Causalities definition for timed Connections

Other constraints follow the same pattern. In total, we define 56 behavioral invariants to specify the scheduling of LF rewriting rules.

IV. Debugging and Verification Tools

We employ a simple example, depicted in the bottom portion of Figure 2, to illustrate the tooling resulting from the encoding of LF operational semantics in GEMOC Studio. More examples are provided in the GitHub repository mentioned in Section III. The sensor fusion LF program defines three reactors. Two reactors (Sensor1 and Sensor2) produce values periodically every 20ms and 10ms respectively. A Fusion reactor has a reaction, Fusion.1, that is triggered by (either of) the sensor outputs, performs the data fusion, and schedules an action that triggers reaction Fusion.2 only if both sensor values were present. This schedule is done with a zero delay, causing Fusion.2 to be triggered at the next microstep. This example is simple but illustrative since it contains concurrency, interleaving between the sensor reactions, triggering of a reaction with multiple triggers, as well as reaction code dependent scheduling of actions.

Interactive debugger. First, without modifying the existing tooling, we obtain a development environment where one can see the runtime state of the model directly on the graphical representation of the model, can step by step choose a specific scheduling of the reactions among the one accepted by the semantics, and can go back in time to explore new scheduling, provided that the reaction code is written in Groovy. Figure 2 shows a screenshot of a part of the environment. On the top left, the logical steps that can be executed at the current step are proposed. On top right a view of the multi-branch execution trace is provided. It represents different explorations of an execution: blue dots are executed steps, green dots are steps that were possible but not chosen, orange dots are current possible steps in a specific branch and the different lines represent places where the user decided to go back in time to explore a different interleaving. On the bottom of the picture, the existing graphical representation of the LF program is annotated to give a view over the runtime state of the model at the current execution step. We believe these different views help the developer to better understand both the underlying semantic model and the behavior of their particular model.

Finally, if one wants to verify temporal properties of a LF program, we propose two non-exclusive possibilities. First, one can add new MoCCML constraints as *assertions* in the specification. The assertion constraints are not enforced but checked during a specific debug session. Second, if the state space of the program is finite, one can ask for an exhaustive simulation which creates the whole state space of the program in a format suitable to be used as input of the CADP model checker [25] and amenable to verification of temporal properties written in the Model Checking Language (MCL [28]). The diagnosis obtained from the CADP tool can then be injected into the debugging environment. The small example introduced in this section has a finite state space of 56 states and 110 transitions computed in 10 seconds.

Trace debugger. When a model is considered as correct by the developer (using Groovy to sketch out the logic of the reactions), they can specialize the reaction code for a specific target (*e.g.,* in the C language with two threads) and obtain an executable source code. In this case, to enable native debugging, it is also possible to ask the generated code for a *trace* that provides a standardized explanation of the program's execution (e.g., reaction triggering, etc.). In our method, we offer the possibility to inject these traces back into the simulation environment to replay what actually happened

[6]Intuitively, the semantics of the automaton is given as a mapping to a Boolean expression on the Boolean variable corresponding to the clocks used in the automaton. For instance, in the idle state, the resulting boolean constraint is $\neg(release \vee canTick \vee wait \vee) \wedge (starts \vee atFutureTag)$

author version

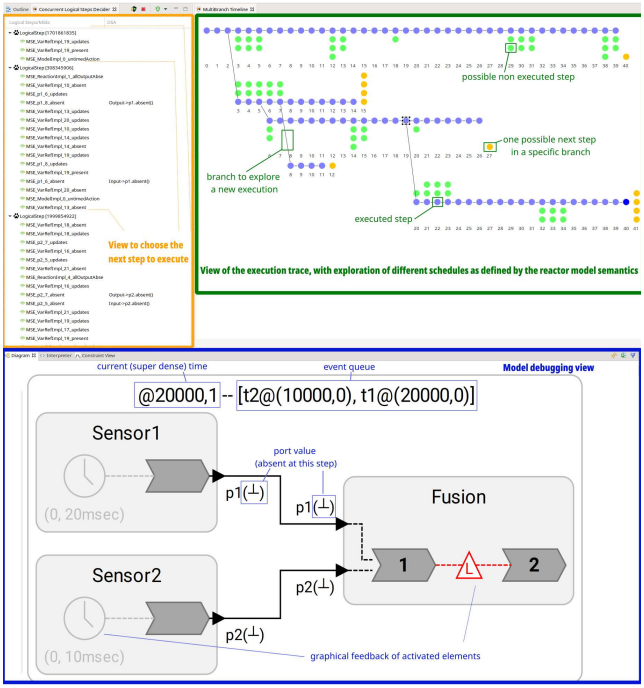

Fig. 2. An LF program under debugging

during the execution. First, when developing a new target, this method could be used to understand if the generated code behaves as expected. Second, it could help in understanding what happened during the actual execution on the target platform. This trace injection is generic because it relies on the generation of a generic representation of the execution of the program, in the form of a trace. Here we chose the dot format[7] and wrote an appropriate translator from the LF trace format[8]. More precisely, we generate a directed acyclic graph representing the reaction execution of a LF program over time from an execution trace. Graph nodes represent execution points occurring at specific super dense times. Graph edges between two nodes exist either if a time advancement occurs between these nodes, or if at least one reaction is executed between the two execution points, or if a value is present on a port. Edge labels indicate the reactions that are executed or the value that is present. If an edge label contains multiple reactions or multiple ports, it means they have been executed or set concurrently.

From the dot representation of the trace, a MoCCML scenario is automatically generated. First, it constrains the specific order of reaction execution and time advancement and second, to support data dependent control it modifies the RTD as retrieve from the trace. It is then possible to use the debugging environment for this specific trace.

Note that many reusable improvements of the GEMOC studio have been required to support the LF semantics, *e.g.,* the priority mechanism previously defined at the model level has been added in the MoCCML mapping meta language; a new MoCCML scenario language with support of RTD modification has been developed and tooled; the support of data dependent state spaces have been developed; a generic KLighD animator has been developed. These changes are available in a the continuous integration of the GEMOC studio but cannot be detailed in this paper.

[7]https://gephi.org/users/supported-graph-formats/graphviz-dot-format/
[8]https://wiki.lf-lang.org/tracing/

V. Related Work

There exist various alternative formalisms capable of specifying the concurrent and timed operational semantics of LF (*e.g.,* TESL [29], Timed Automata [30], synchronous languages [31], and Maude [32] to name a few). Using an external language to provide a language semantics usually requires a (possibly complex) mapping between the source and the target language(s). Doing so has two main drawbacks. First, the execution, debugging, or analysis are done on the target language which creates a semantic gap that may be difficult to fill by the developer or by the tooling. Second, for an evolving language like LF, experimentation with new features may be difficult to do since it may require changes to the runtime implementation or the mapping.

Sirjani et al. [33] proposed to use Timed Rebeca, a timed and reactive language based on actors, to perform model checking on LF programs. This is convenient since the semantics of LF and Rebeca are close. However, no automatic transformation, nor a way to fill the resulting semantic gap between the LF code and the Rebeca models are provided. Instead, our approach directly augments the original LF language implementation with an abstract operational model and tooling, avoiding drawbacks mentioned earlier.

There also exists related work in the field of embedded DSLs, where the DSL is defined as syntactic sugar for the host DSL and the execution/analysis relies on the execution capabilities of the host language [4], [5], [34], [35]. In these approaches, the debugger has to do extra work to make the results understandable to the user of the DSL. There also exist debuggers provided by language workbenches [36], [37]. However, these approaches are not supported anymore, are not based on any well-defined formalism, and provide no support for omniscient debugging or state space analysis.

VI. Conclusion & Future Work

We present a framework for analyzing and debugging LF programs and runtime implementations. We follow the GEMOC MoCCML approach, in which we augment the language artifacts in the existing compiler implementation with instrumentation that encodes the operational semantics of the language. As a result, we obtain access to advanced tools like an omniscient debugger, exhaustive simulation, and graphical representations like diagram animation. This functionality depends on the ability to interpret Groovy reaction code to support data dependent control. Even with a "black box" treatment of reaction code, however, our tooling proves useful (albeit to a more limited extent) when applied to *execution traces*, which can be produced by any target runtime.

Author version

All together, we believe the proposed tooling provides a handy way to understand LF programs. Only 1300 lines of code are written to implement the LF semantics and obtain an omniscient debugging environment of LF programs. An extra 550 lines of code are necessary to obtain tuning of the graphical animation and another 350 lines of code are necessary to obtain the trace to dot utility.

Our work does not yet support all the features of LF. Two important features are not supported yet: the link with physical time and handling multiple instantiations of the same reactor. During an LF execution, the logical time always lags behind physical time, and if the next available tag is early, the runtime waits for physical time to catch up [1]. During a debugging session using breakpoints, it is not clear how such a link between the logical time and physical time should be addressed. The issue regarding multiple instantiations stems from a limitation of the MoCCML compiler, and is to be addressed in near future.

References

[1] M. Lohstroh, C. Menard, S. Bateni, and E. A. Lee, "Toward a Lingua Franca for deterministic concurrent systems," *ACM Trans. Embed. Comput. Syst.*, vol. 20, no. 4, May 2021.

[2] M. Lohstroh, Í. Íncer Romeo, A. Goens, P. Derler, J. Castrillon, E. A. Lee, and A. Sangiovanni-Vincentelli, "Reactors: A deterministic model for composable reactive systems," in *8th International Workshop on Model-Based Design of Cyber Physical Systems (CyPhy'19)*, vol. LNCS 11971. Springer-Verlag, 2019, Conference Proceedings, in press.

[3] M. Lohstroh, "Reactors: A deterministic model of concurrent computation for reactive systems," Ph.D. dissertation, EECS Department, University of California, Berkeley, Dec 2020.

[4] D. Pavletic, S. A. Raza, M. Voelter, B. Kolb, and T. Kehrer, "Extensible debuggers for extensible languages," *GI/ACM WS on Software Reengineering*, 2013.

[5] A. Chiş, T. Gîrba, and O. Nierstrasz, "The moldable debugger: A framework for developing domain-specific debuggers," in *Software Language Engineering*, B. Combemale, D. J. Pearce, O. Barais, and J. J. Vinju, Eds. Cham: Springer International Publishing, 2014, pp. 102–121.

[6] E. Bousse, T. Degueule, D. Vojtisek, T. Mayerhofer, J. Deantoni, and B. Combemale, "Execution framework of the GEMOC studio (tool demo)," in *Proc. ACM SIGPLAN Int'l Conference on Software Language Engineering (SLE'16)*, 2016, pp. 84–89.

[7] C. Hewitt, P. B. Bishop, and R. Steiger, "A universal modular ACTOR formalism for artificial intelligence," in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, USA, August 20-23, 1973*, 1973, pp. 235–245.

[8] E. A. Lee, "The problem with threads," *Computer*, vol. 39, no. 5, pp. 33–42, 2006.

[9] A. Benveniste and G. Berry, "The synchronous approach to reactive and real-time systems," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1270–1282, 1991.

[10] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*. Pearson Education, 2008.

[11] B. Combemale, J. DeAntoni, M. V. Larsen, F. Mallet, O. Barais, B. Baudry, and R. B. France, "Reifying concurrency for executable metamodeling," in *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings*, ser. Lecture Notes in Computer Science, M. Erwig, R. F. Paige, and E. V. Wyk, Eds., vol. 8225. Springer, 2013, pp. 365–384.

[12] J. Deantoni, P. I. Diallo, J. Champeau, B. Combemale, and C. Teodorov, "Operational semantics of the model of concurrency and communication language," INRIA, Research Report RR-8584, Sep. 2014.

[13] J. Deantoni, P. Issa Diallo, C. Teodorov, J. Champeau, and B. Combemale, "Towards a meta-language for the concurrency concern in DSLs," in *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, Grenoble, France, Mar. 2015.

[14] D. E. Knuth, "The genesis of attribute grammars," in *Attribute Grammars and their Applications*, P. Deransart and M. Jourdan, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1990, pp. 1–12.

[15] J. Deantoni and F. Mallet, "ECL: the event constraint language, an extension of OCL with events," INRIA, Tech. Rep. RR-8031, Jul. 2012.

[16] J. Deantoni, C. André, and R. Gascon, "CCSL denotational semantics," Inria, Research Report RR-8628, Nov. 2014.

[17] J. Deantoni, "Towards formal system modeling: Making explicit and formal the concurrent and timed operational semantics to better understand heterogeneous models," Habilitation à diriger des recherches, Université Côte d'Azur, CNRS, I3S, France, Jul. 2019.

[18] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[19] C. Fidge, "Logical time in distributed computing systems," *Computer*, vol. 24, no. 8, pp. 28–33, 1991.

[20] R. Alur, T. A. Henzinger, O. Kupferman, and M. Y. Vardi, "Alternating refinement relations," in *International Conference on Concurrency Theory*. Springer, 1998, pp. 163–178.

[21] J. DeAntoni and F. Mallet, "TimeSquare: Treat your models with logical time," in *50th Intl. Conf. on Objects, Models, Components, Patterns*, ser. LNCS. Springer, 2012, no. 7304, pp. 34–41.

[22] F. Latombe, X. Crégut, B. Combemale, J. Deantoni, and M. Pantel, "Weaving concurrency in executable domain-specific modeling languages," in *Proc ACM SIGPLAN Int'l Conf Software Language Engineering (SLE'15)*, 2015, pp. 125–136.

[23] R. Gascon, J. Deantoni, and J.-F. Le Tallec, "Priority in logical time partial orders with synchronous relations," in *IEEE RIVF 2019 - Research, Innovation and Vision for the Future*, Danang, Vietnam, Mar. 2019.

[24] B. Lewis, "Debugging backwards in time," *arXiv preprint cs/0310016*, 2003.

[25] H. Garavel, F. Lang, R. Mateescu, and W. Serwe, "CADP 2011: a toolbox for the construction and analysis of distributed processes," *International Journal on Software Tools for Technology Transfer*, vol. 15, no. 2, pp. 89–107, 2013.

[26] E. A. Lee and H. Zheng, "Operational semantics of hybrid systems," in *International Workshop on Hybrid Systems: Computation and Control*. Springer, 2005, pp. 25–53.

[27] S. Edwards and J. Hui, "The sparse synchronous model," in *2020 Forum for Specification and Design Languages, FDL 2020, Kiel, Germany, September 15-17, 2020*. IEEE, 2020, pp. 1–8.

[28] R. Mateescu and D. Thivolle, "A model checking language for concurrent value-passing systems," in *International Symposium on Formal Methods*. Springer, 2008, pp. 148–164.

[29] F. Boulanger, C. Jacquet, C. Hardebolle, and I. Prodan, "TESL: a language for reconciling heterogeneous execution traces," in *Formal Methods and Models for Codesign (MEMOCODE), 2014 Twelfth ACM/IEEE International Conference on*, Lausanne, Switzerland, Oct 2014, pp. 114–123.

[30] R. Alur, "Timed automata," in *Computer Aided Verification*, N. Halbwachs and D. Peled, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 8–22.

[31] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. De Simone, "The synchronous languages 12 years later," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 64–83, 2003.

[32] J. E. Rivera, F. Durán, and A. Vallecillo, "Formal specification and analysis of domain specific models using Maude," *SIMULATION*, vol. 85, no. 11-12, pp. 778–792, 2009.

[33] M. Sirjani, E. A. Lee, and E. Khamespanah, "Verification of cyberphysical systems," *Mathematics*, vol. 8, no. 7, 2020.

[34] X. Li and M. Flatt, "Debugging with domain-specific events via macros," in *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, 2017, pp. 91–102.

[35] M. Voelter and K. Solomatov, "Language modularization and composition with projectional language workbenches illustrated with mps," *Software Language Engineering, SLE*, vol. 16, no. 3, 2010.

[36] R. T. Lindeman, L. C. Kats, and E. Visser, "Declaratively defining domain-specific language debuggers," *ACM SIGPLAN Notices*, vol. 47, no. 3, pp. 127–136, 2011.

[37] P. Klint, "A meta-environment for generating programming environments," *ACM Trans. Softw. Eng. Methodol.*, vol. 2, no. 2, p. 176–201, Apr. 1993.

Author Version