2023 IEEE/ACM 11th International Conference on Formal Methods in Software Engineering (FormaliSE) | 979-8-3503-1263-8/23/$31.00 ©2023 IEEE | DOI: 10.1109/FormaliSE58978.2023.00018

Transparent Actor Model

Fatemeh Ghassemi*†, Marjan Sirjani‡ Ehsan Khamespanah*, Mahrokh Mirani† and Hossein Hojjat†*
*University of Tehran, Iran
Email: fghassemi, e.khamespanah, hojjat@ut.ac.ir
‡Mälardalen University, Sweden
Email: marjan.sirjani@mdh.se
†Khatam University, Iran
Email: m.mirani@khatam.ac.ir

***Abstract*—Several programming and formal modeling languages are designed based on actors. Each language has certain policies for message delivery between actors and for handling the messages in the buffers. These policies are implicit in the semantics of each language. One can infer interesting properties of actor languages related to communication and coordination based on different policies and their interactions. We define the "Transparent Actor" model where we make policies explicit as points of possible variations. We identify an abstract network entity and define the semantics of Transparent Actors in three parts: actors, network, and composition. We define a core actor language named BABEL as a basis to describe the semantics of Transparent Actors using structural operational semantics (SOS) rules with variation points. These parametric rules make the implicit policies clear and can be used as a template to define the semantics of different actor-based languages. We evaluate the applicability of the template by examining the semantics for actor-based languages Rebeca, Lingua Franca, ABS, AKKA, and Erlang. We implement BABEL in Maude as a proof of concept, then concretize the parametric rules to implement some of the above languages. We consider a few properties, check them via a set of designated litmus test cases using our Maude implementations, and discuss the policy interactions.**

***Index Terms*—Distributed systems, Actor languages, Design policy, Structural operational semantics**

I. INTRODUCTION

Actors are getting more attention as the number of distributed and networked systems increases. For example, in managing complex cyber-physical systems, connected vehicles, and highly-sensitive distributed data we need precise synchronization and coordination in a networked system. We introduce the "Transparent Actor" model to make communication and coordination among actors, and local scheduling of tasks within an actor, explicit and transparent.

Hewitt [1] introduced the actor model as an agent-based language in the 70s, and Agha [2] developed it as a concurrent object-based language. Actors have been used as a framework for theoretical understanding of concurrent and distributed computation, as the basis for designing many modeling and programming languages, and as a model for many practical implementations of concurrent systems. Actors are units of concurrency, with no shared variables, communicating via asynchronous message passing. Each actor has a mailbox with a unique address to store the received messages. An actor's behavior is defined using a set of "message handlers" or "methods" that specify how the actor reacts to each received message. Many modeling languages, like Timed Rebeca [3] and ABS [4], and many programming languages like Erlang [5], Scala [6], Go [7], and Lingua Franca [8] are designed based on actors. Languages are designed with different prime purposes. For example, Rebeca and ABS are designed for analysis and code generation [9], while Lingua Franca aims at providing timed deterministic programs, and Erlang is optimized for efficient execution.

Motivation: Different languages adopt different features and these subtle differences may make confusion in understanding the semantics. According to [10], the best way for understanding a language is to reduce it to its essence, figure out its features and determine how these features are composed. The selected feature set and composition reveal the design decisions. Karmani et al. [11] argue that understanding the implications of the various design decisions in building or using a particular actor-based framework is not always easy. The goal of Transparent Actors is to make the communication and coordination among actors, and local scheduling of tasks within an actor explicit and transparent; and we select the features of our focus according to this goal. Our main decision is to make the network explicit in our Transparent Actor model (and define an abstract network entity). We structure the semantics of Transparent Actors as "actor", "abstract network", and their "composition". For communication, we consider the operations of sending and receiving messages by actors and transferring messages from one actor to another by the abstract network. For task scheduling in an actor, we consider taking a message from the mailbox by actors, and for coordination, we consider the scheduling of actors. We distinguish the variation points in all the above operations. We believe these are the main features in designing actor languages that affect communication and coordination. There are several features of actor languages, e.g. dynamic actor creation, dynamic topology, and typical features in programming languages like typing, that we do not consider or discuss in our work. We can use these features to investigate different properties of actor languages. We do not claim that this is a canonical form for defining the features of actors, but we came up with these features and properties based on our years of experience with different actor languages and actor-based applications.

We define a basic actor language and use Structural Operational Semantics (SOS) to define its semantics as a labeled

979-8-3503-1263-8/23/$31.00 ©2023 IEEE
DOI 10.1109/FormaliSE58978.2023.00018

transition system. These rules are defined with variation points that allow them to be used as templates for defining the semantics of different actor languages. We use these templates to define the semantics of Rebeca and Lingua Franca. We also discuss how they can be extended to support the languages ABS, AKKA, and Erlang. We show the design decisions of each language at the variation points.

We organize the rules into three levels: actor, abstract network, and their compositions. At the actor level, we have two variation points: one for receiving and one for taking a message. For example, when the mailbox is full, the receiving policy in some languages is to drop new messages, and in others, to overwrite the old messages. The policy for taking a message from the buffer realises the scheduling of tasks within an actor, it may be FIFO (First-In First-Out), EDF (Earliest Deadline First), or based on priority or pattern matching. At the abstract network level, the transfer of messages can be based on different policies. For example, in Timed Rebeca and LF the messages are transferred respecting the order of their time tags, in Rebeca the order of sending (in one actor) determines the order of transfer, and in ABS it is nondeterministic. At the composition level, the semantics of the language decides which actor is scheduled next. We manage to capture different coordination and synchronization mechanisms of Rebeca, ABS, and LF at the composition level. Rebeca schedules actors in a nondeterministic way, ABS shares a thread among a group of concurrent objects, and the semantics of LF is close to synchronous languages. Generally, these policies are not explicitly stated in the definitions of languages and they can only be extracted from their formal semantics (if any). The variation points that we identify in this paper are the questions one usually asks to understand the semantics of an actor-based language. These questions may also be used for understanding many features of distributed systems in general. In summary, our contributions are as follows:

- We make the communication and coordination in actor languages more explicit and transparent, and divide the semantic rules into three parts of actor, abstract network, and their composition;
- We make the variation points (features) in designing actor languages explicit and transparent. We formalise these decisions as policies in the transition relations in the semantics;
- We express policies of several actor-based languages based on their design decisions;
- We discuss language properties as the consequence of choosing different policies (feature interaction).

Related Work: It is a common practice to design frameworks in order to compare various aspects of programming languages [10]. Henderson and Horn [12] give a comparison of four popular object-oriented languages of the time (Oberon, Modula, Sather and Self). The features of comparison in [12] include inheritance, dynamic dispatch, code reuse, information hiding, and performance. Features of context-oriented programming languages (COPL) are identified in [13]. Based on variations of these features, they compare eleven COPLs and assess the overhead of design decisions using a set of benchmarks. A framework for analyzing the properties of COPLs is proposed in [14]. The framework serves as an evaluation means based on a set of the COPL's concepts, the development process support, and pragmatics issues. To the best of our knowledge, we are the first in proposing a comparison framework for actor-based languages.

II. Transparent Actors

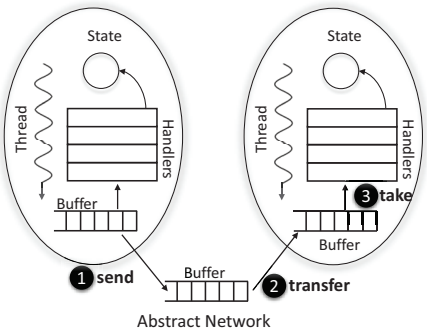
In Transparent Actors (see Fig 1) we make the transfer of messages among actors explicit and model that as an abstract network. In Transparent Actors we consider three phases for a point-to-point message communication as illustrated in Fig. 1a:

- *Sending*: The actor pushes its messages into the buffer of the abstract network.
- *Transferring*: The abstract network delivers messages to the receiver. The abstract network behaves like an actor in transferring a message, it takes a message from its buffer and then sends it to the receiver actor.
- *Taking*: The actor chooses a message from its buffer to handle.

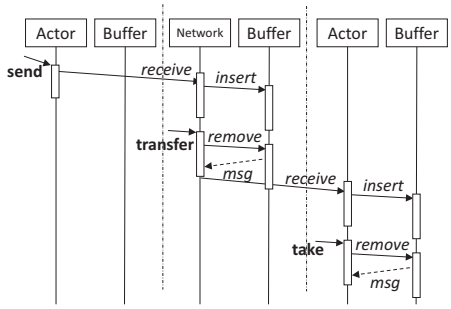
For these three phases, we define a set of operations with well-defined responsibilities for actors, the abstract network, and buffer entities. Policies are the variation points in these operations. Fig. 1b shows the relation between entities and their operations. The sending phase starts when an actor calls its $send$ operation. This operation pushes a message into the abstract network by calling (asynchronously) its $receive$ operation. The $receive$ operation stores the received messages into the network buffer by calling the operation $insert$. The transferring phase starts when the abstract network calls its $transfer$ operation to dispatch received messages to their recipients. The $transfer$ operation first selects a message from the network buffer, based on a policy that determines which messages should be dispatched. The message is removed from the network buffer by calling the $remove$ operation of the buffer. When a message is selected for delivery, it is delivered to its destination actor by calling the $receive$ operation of the actor which has a policy of its own. This policy indicates whether the message can be accepted or dropped, and in the case of acceptance how the buffer should be updated. This operation calls the $insert$ operation of the buffer. The taking phase starts when actors call the operation $take$ to handle the messages in their buffers. This operation selects a message based on a policy that determines which messages should be handled. When a message is selected, it is removed from the buffer by calling the $remove$ operation of the buffer.

III. Actor Language Framework

To concretely highlight the variation points, we introduce a simple language, called Babel: Basic Actor-based Language. We define its semantics by providing a set of template SOS rules that have a set of variation points at their assumptions. Fixing those variation points leads to the concrete semantics



(a) Three phases of point-to-point communication.



(b) Actors sending and receiving messages. Network transfer and buffer operations are transparent.

Fig. 1: Transparent Actors

of an actor-based language. These variation points show the decisions that should be made in designing actor-based languages. We distinguish these assumptions by ? preceding the name given to the assumptions.

We rely on a set of notations and assumptions to define the rules. Assume Var is the set of variable identifiers and $Value$ is the set of possible values for variable identifiers. We define a valuation function (environment) as $Env : Var \rightarrow Value$. Let $domain(e)$ denote the domain variables of the environment e and $e[x \mapsto y]$ denote updating e with the mapping x to y. We use the type $Buffer$ to address the actor mailbox and network buffer in the semantics. We consider a set of operations on this type: $insert$, $remove$, $contains$, and $size$. We use $\emptyset$ to denote an empty buffer. The items in the buffer are of type Msg. ID is the set of actor identifiers and each actor has a unique identifier. Messages are a tuple of elements that essentially include the method name and the endpoint. Let $Name$ be the set of method names. Note that the message tuple may also include the method parameters, the sending actor identifier, and the delivery time, depending on the target language. We use the dot notation to access elements of a message tuple. We assume that $m.rcv$ specifies the receiving actor which can be inferred from the endpoint in the message.

A. BABEL: *Basic Actor Language - The Syntax*

BABEL has the core features of actor-based languages. There is no shared variable, communication takes place through asynchronous message passing, and the computation is message-driven (messages trigger method activation). We consider non-preemptive execution of methods (like most actor-based languages); meaning that an actor cannot handle another message unless it completes its previous method execution. We do not include the features (like actor creation) that do not impact the variation points of our interest. The syntax of BABEL is given in Fig. 2 inspired by the Rebeca syntax.

A model is defined by a set of actor classes and a main block. Each actor class is declared by keyword actorclass. We define the state variables of a class by using statevars and its methods by method. The body of methods is specified by a list of statements. Execution of a send statement generates a message. We do not fix the syntax of the send statement deliberately. Generally speaking, v ! name expresses that a message corresponding to the method name is sent to an endpoint v. The variable v in some languages may directly refer to an actor or maybe an output port. The statement end defines the end of a method.

B. BABEL *Semantics*

We define the semantics of BABEL in terms of the labeled transition system $\langle S, \rightarrow, s_0 \rangle$, where S is the set of global states, $\rightarrow \subseteq S \times Act \times S$ is the set of transition relations, Act is the set of actions, and s_0 is the initial state. We provide a set of template rules to derive the transition relation $\rightarrow$ over the global states. The global states are defined as the composition of the local states of actors and the abstract network. The transition relation over the global state is defined based on the transition relations over the local states of actors and the abstract network. We provide the transition rules for local states of actor and abstract network levels. We organize these template rules into three levels: actor, network, and composition, depicted in Table I. The template rules may have a variation point.

1) Transition Relation Rules of the Actor Level: We define the transition relation of actors as $\longrightarrow\!\!\triangleright \subseteq LocalSt \times Act \times$

Model ::=	$\langle$Class$\rangle^+$ Main
Main ::=	main {ActorInstance$^+$}
ActorInstance ::=	C actor : ($\langle$const$\rangle^*$)
Class ::=	actorclass C {Vars Method$^+$}
Vars ::=	statevars $\langle$T v;$\rangle^*$
Method ::=	method name {$\langle$Stmt;$\rangle^*$end}
Stmt ::=	v = $expr$ | v!name | skip |
	if($expr$) {$\langle$Stmt;$\rangle^+$} else {$\langle$Stmt;$\rangle^+$} | end

Identifiers C, T, $name$, v, $const$, $expr$, and $actor$ resp. denote class, type, handler name, variable, constant, expressions, and actor name.
$\langle\ \rangle$ denotes meta parenthesis, $+$ and $*$ denote resp. one or more and zero or more repetitions.

Fig. 2: BABEL Syntax.

$LocalSt$ where $LocalSt \subseteq Env \times Buffer \times Stmt^*$ is the set of local states of actors, and $Act = \{\tau, end\} \cup \{m!, m?, m | m \in Msg\}$. The action τ denotes an internal action, end indicates completion of a method, m shows taking the message m from the buffer, $m?$ receipt of the message m, and $m!$ sending the message. The local state of an actor is (e, b, π) where $e \in Env$ is a valuation function with the domain of state variables, $b \in Buffer$ is the local buffer and $\pi \in Stmt^*$ is the remaining statements from a method that should be executed. An empty sequence, denoted by ϵ, shows that an actor is not executing any methods.

Each actor has a thread that invokes the operation $take$ to process a message from the buffer. The abstract network calls the operation $receive$ to deliver a message to the actor. We define the behavior of an actor by defining the semantics of these two operations, namely $receive$ and $take$, and how the actor internally evolves, shown in Table I. The rule RECEIVE specifies the semantics of the operation $receive$. This rule explains that an actor can receive message $m \in Msg$ if it is admissible by its receive policy based on the status of the actor buffer.

The predicate $?rcvPolicy(b, m)$ determines whether the receiving actor can receive m with respect to the receive policy and the current state of the buffer b. Different policies can be used for inserting a message into the buffer. When there is a buffer overflow, one policy is to overwrite the oldest message in the buffer or to purge the new arriving message. In timed models, we may have different policies for handling timestamped messages, e.g., for handling duplicate messages (the same messages sent by a sender at different timestamps) or old (stale) messages (messages with timestamps too old to be valuable). We can also enforce the visibility of messages and generally, privacy rules like who can receive a message from whom in the receive policy.

According to the rule TAKE (semantics of the operation $take$) an actor can take a message from its buffer when it is not busy handling any other messages, i.e., when it has no remaining statement to execute (ϵ). The actor handles the taken message by executing the statements of its corresponding method $Method(m.name)$. The rule has two assumptions. The predicate $contains(b, m)$ expresses that the actor can only take messages from its buffer. The predicate $?takePolicy(e, b, m)$ determines whether the message m can be taken with respect to the take policy, the actor buffer b, and its local variables in e. For example, taking the message with the highest priority, or the message with the smallest timestamp. If the policy is the smallest timestamp and more than one message has the same timestamp, there should also be a policy to determine whether we take a message nondeterministically, or, if there is a pre-defined priority. Another example is languages such as Erlang where pattern matching is used for the incoming messages. As a result of the $take$ operation, the content of b is updated by calling $remove(b, m)$.

We separate the semantics of method statements and the actor-level semantics to simplify the rules at the Actor-level semantics. The semantics of the statements of a method is shown in Table II. Rule STMTEXEC connects the rules in Table II to the rules in the Actor-level semantics rules. The rule indicates that the execution of a statement does not have any effect on the buffer.

TABLE I: Semantics of BABEL

Actor	(STMTEXEC) $\dfrac{e, \pi \overset{\alpha}{\hookrightarrow} e', \pi'}{(e, b, \pi) \overset{\alpha}{\longrightarrow\!\triangleright} (e', b, \pi')}$
	(TAKE) $\dfrac{contains(b, m)?takePolicy(e, b, m)}{(e, b, \epsilon) \overset{m}{\longrightarrow\!\triangleright} (e, remove(b, m), Method(m.name))}$
	(RECEIVE) $\dfrac{?rcvPolicy(b, m)}{(e, b, \pi) \overset{m?}{\longrightarrow\!\triangleright} (e, insert(b, m), \pi)}$
Network	(RECEIVE) $(e, b) \overset{m?}{\rightsquigarrow} (e, insert(b, tag(e, m)))$
	(TRANSFER) $\dfrac{contains(b, m)?transferPolicy(e, b, m)}{(e, b) \overset{m!}{\rightsquigarrow} (e, remove(b, m))}$
Composition	(ACTORPROG I) $\dfrac{s(x) \overset{m}{\longrightarrow\!\triangleright} (e', b', \pi') \;\; ?schedPolicy(e, s, x)}{(e, s, n) \overset{m}{\rightarrow} (start(e, x), s[x \mapsto (e', b', \pi')], n)}$
	(ACTORPROG II) $\dfrac{s(x) \overset{\tau}{\longrightarrow\!\triangleright} (e', b', \pi')}{(e, s, n) \overset{\tau}{\rightarrow} (e, s[x \mapsto (e', b', \pi')], n)}$
	(ACTORPROG III) $\dfrac{s(x) \overset{end}{\longrightarrow\!\triangleright} (e', b', \pi')}{(e, s, n) \overset{\tau}{\rightarrow} (finish(e, x), s[x \mapsto (e', b', \epsilon)], n)}$
	(COMM I) $\dfrac{s(x) \overset{m!}{\longrightarrow\!\triangleright} (e', b', \pi') n \overset{m?}{\rightsquigarrow} (e'', b'')}{(e, s, n) \overset{\tau}{\rightarrow} (e, s[x \mapsto (e', b', \pi')], (e'', b''))}$
	(COMM II) $\dfrac{n \overset{m!}{\rightsquigarrow} (e'', b'') m.rcv = ys(y) \overset{m?}{\longrightarrow\!\triangleright} (e', b', \pi')}{(e, s, n) \overset{\tau}{\rightarrow} (e, s[y \mapsto (e', b', \pi')], (e'', b''))}$

TABLE II: Semantics of BABEL statements. $\alpha \in \{\tau, end, m!\}$ where $m \in Msg$.

(COND 1) $\dfrac{eval(expr, e)}{e, if\ (expr)\ \pi_1\ else\ \pi_2 \overset{\tau}{\hookrightarrow} e, \pi_1}$

(COND 2) $\dfrac{\neg eval(expr, e)}{e, if\ (expr)\ \pi_1\ else\ \pi_2 \overset{\tau}{\hookrightarrow} e, \pi_2}$

(SEQ 1) $\dfrac{e, \pi_1 \overset{\alpha}{\hookrightarrow} e', \pi'}{e, \pi_1; \pi_2 \overset{\alpha}{\hookrightarrow} e', \pi'; \pi_2}$ (SEQ 2) $e, \top; \pi \overset{\tau}{\hookrightarrow} e, \pi$

(SKIP) $e, skip \overset{\tau}{\hookrightarrow} e, \top$ (END) $e, end \overset{end}{\hookrightarrow} e, \top$

(ASSIGN) $e, v = expr \overset{\tau}{\hookrightarrow} e[v \mapsto eval(expr, e)], \top$

(SEND) $e, v!n \overset{GenerateMsg(e,v,n)!}{\hookrightarrow} e, \top$

The relation $\hookrightarrow$ expresses the semantics of statements as given in Table II. The function $eval(expr, e)$ evaluates the given $expr$ based on the actor environment e. We denote the successful termination of a single statement by $\top$. The function $GenerateMsg(e, v, n)$ generates a message of Msg, based on the actor environment e, the variable v, and the method name n. This function depends on the structure of the message in each language. For example, it may generate a message with the tuples (i, v, n) in Rebeca where i is the identifier of the sending actor retrieved from e.

2) Transition Relation Rules of the Abstract Network Level: We define the transition relation of the abstract network by $\rightsquigarrow \subseteq NetLocal \times Act \times NetLocal$ where $NetLocal \subseteq Env \times Buffer$ is the set of network local states and $Act =$

$\{m!, m? \mid m \in Msg\}$. For brevity, we use "network" instead of "abstract network". We represent the local state, $NetLocal$, by a pair of (e, b) where $e \in Env$ is the valuation of network local variables and $b \in Buffer$ is its local buffer. The network provides two operations $receive$ and $transfer$. An actor calls the operation $receive$ to deliver its message for transmission. The network calls its $transfer$ operation to dispatch the buffer messages to their recipients. The network-level transition relation rules define the semantics of the network operations, shown in Table I. The rule RECEIVE specifies the semantics of operation $receive$. The rule explains that the network can always receive a message. The network may add some information, by calling tag, to the received messages before their insertion into the buffer. For example, the network may add the delivery time of the message based on the communication delay between two actors.

The rule TRANSFER defines the semantics of the $transfer$ operation. The rule has two assumptions $contains(b, m)$ and $?transferPolicy(e, b, m)$. The predicate $?transferPolicy(e, b, m)$ determines whether the message m can be selected from the buffer b to dispatch with respect to the transfer policy and the network variables in e. By specifying this assumption, we explicitly identify the policy on selecting messages for transmission. For instance, this policy may dispatch messages based on their arrival or timestamps. For messages with the same timestamp, the policy may consider a priority among messages or transfer them nondeterministically.

3) Transition Relation Rules of the Composition: We define the transition relation for the composition level by $\rightarrow \subseteq GlobalSt \times Act \times GlobalSt$ where $GlobalSt$ is the set of global states and $Act = \{\tau\} \cup \{m \mid m \in Msg\}$. The global state is defined by the values of global variables, and the local states of actors and the network. Note that there are no shared variables among the actors. Global variables refer to e.g. the variables required for scheduling actors in ABS or the global time in LF and Timed Rebeca. A global state is (e, s, n) where $e \in Env$ is the valuation function for the global variables, $s : ID \rightarrow LocalSt$ is a mapping from actor identifiers to their local states, $n \in NetLocal$ is the network local state.

The rules at this level define the semantics of (1) communication between actors and the network, (2) the progress of actors, given in Table I. An actor locally evolves when either it takes a message from its buffer or it executes the statements of a method. Executing a send statement results in communication from the sending actor to the network, and then from the network to the receiving actor.

ACTORPROG I rule: An actor x among those that can take a message, is selected. The assumption $?schedPolicy(e, s, x)$ shows the scheduling policy on selecting x for execution. This policy may select an actor based on the priority of actors given by e and the local states of actors. The function $start(e, x)$ defines how the environment should be updated based on the scheduling policy upon starting the execution of an actor method.

ACTORPROG II rule: An actor x executes a statement (except send and the end of a method) and locally updates its state. As a consequence, the global state changes accordingly.

ACTORPROG III RULE: An actor x terminates the execution of a method by executing end. The function $finish(e, x)$ defines how the environment should be updated based on the scheduling policy upon the termination of a method.

COMM I RULE: This rule shows the communication of an actor x with the network. The assumption $s(x) \xrightarrow{m!} (e', b', \pi')$ indicates that the actor with the identifier x has the message m for delivery (by executing a send statement). The assumption $n \overset{m?}{\rightsquigarrow} (e'', b'')$ expresses that the network is ready to receive m.

COMM II RULE: This rule shows the communication of the network with an actor x. When the network has a message ready to deliver, i.e., $n \overset{m!}{\rightsquigarrow} (e'', b'')$, and the intended actor to receive m is ready to receive, i.e., $s(y) \xrightarrow{m?} (e', b', \pi')$, the message is transferred.

We have summarized the transition relation rules for each level with their variation points in Table III. To define the semantic rules of a language, we mainly instantiate those rules by specifying their variation points. The rules with no variation point are inherited from the generic framework with no modification. This table makes the decision points in designing actor languages explicit and transparent. Note that for some languages like LF and Timed Rebeca, the rules given in Table I are extended to address real-time behavior.

IV. CORE REBECA AND LINGUA FRANCA

We show the semantics of two actor-based modeling languages using the proposed frameworks. The first language, Core Rebeca, does not support timing. Lingua Franca supports real-time features and we will show how they are supported by the framework. Note that in most timed languages, the progress of time is modeled by computation time and/or communication time.

To derive the semantics of each language, we instantiated the variation points of the template rules of the framework based on our comprehension of their given semantics. We iteratively moved between the framework and the semantics of each language to revise the framework many times to remove ambiguities and inconsistencies.

A. Core Rebeca

Rebeca [15], [16] is an actor-based language for modeling and verification of concurrent and distributed systems. Rebeca has a Java-like syntax and is supported by the model checking tool Afra [17]. A Rebeca model consists of a set of class declarations and a main block. Actors, called rebecs, are instantiated from the defined *reactive classes* in the main block in the model. Each class has three parts: *state variables*, *known rebecs*, and *message servers* (methods).

Example 1: The Rebeca model given in Fig. 3 shows a system composed of a monitor and an alarm. The monitor is informed of the temperature of its environment by receiving a check message. If the received value is above the specified

TABLE III: Variation points of semantics rules at each semantic level. Semantic rules with no variation points are not included.

Level	Rule		
	Name	Assumption	Intuition of assumption
Actor	RECEIVE	?*rcvPolicy*	When a message can be inserted into the buffer. Can we insert when the buffer is full or we should drop or rewrite?
	TAKE	?*takePolicy*	The policy by which a message is selected, based on e.g., arrival order (FIFO), deadline (EDF), timestamp, priority, the sender, pattern matching or nondeterministic.
Network	TRANSFER	?*transferPolicy*	The policy for transferring a message, e.g., after a network delay or based on the priority of the receiver, or the network communication protocol e.g., token ring, Time Division Multiple Access.
Composition	ACTORPROG I	?*schedPolicy*	The policy for selecting an actor to execute, based on e.g., the priority of actors or nondeterministically among a group of actors.

```
1  reactiveclass Monitor(5) {
2    knownrebecs { Alarm a; }
3    statevars { int max; }
4    Monitor(int x) {
5      max = x;
6    }
7    msgsrv check(int temp) {
8      if (temp > max) {
9        a.notify () ;
10     }
11   }
12 }
13 reactiveclass Alarm(5) {
14   knownrebecs { }
15   statevars { }
17   msgsrv notify () {
18     //Play the alarm
19   }
20 }
21 main {
22   Monitor m (a):(25) ;
23   Alarm a():() ;
24 }
```

Fig. 3: A Rebeca model of a monitor and an alarm.

TABLE IV: Semantic rules of Core Rebeca. $\beta = \{\tau, end\} \cup \{m \mid m \in Msg\}$.

Actor	(RECEIVE) $(e,b,\pi) \xrightarrow{m?} (e, insert(b,m), \pi)$ (TAKE) $\dfrac{contains(b,m) m = head(b)}{(e,b,\epsilon) \xrightarrow{m} (e, remove(b,m), Method(m.name))}$
Network	(RECEIVE) $(e,b) \overset{m?}{\rightsquigarrow} (e, insert(b,m))$ (TRANSFER) $\dfrac{contains(b,m) \exists x \in ID \cdot m = head(getBuff(b,x))}{(e,b) \overset{m!}{\rightsquigarrow} (e, remove(b,m))}$
Compos.	(ACTORPROG) $\dfrac{s(x) \xrightarrow{\beta} (e',b',\pi')}{(e,s,n) \xrightarrow{\beta} (e, s[x \mapsto (e',b',\pi')], n)}$

value max, it sends a notify message to its known rebec of class Alarm (line 9). In the main block, the known rebecs and actual values of parameters of the constructors are passed via two pairs of parenthesis (lines 22-23). So the known rebec a of monitor m is initialized by the first pair of parenthesis while max is initialized by passing 25 to the constructor via the second pair of parenthesis (line 22). The numeric arguments to the reactive class declarations are the buffer lengths.

For simplicity, we assume that messages have no parameters. The ID set includes the name of instantiated rebecs, and the $Name$ set includes the names of all message servers. The set of statements in the body of message servers is the same as BABEL. The variable in the send statement should be a variable declared within the *knownrebecs* scope.

Structure of Messages and Buffers The messages consist of three parts: the sender identifier, the message name, and the receiver identifier, i.e., $Msg = ID \times Name \times ID$.

The buffers of actors are unbounded buffers with the FIFO policy. However in practice, the buffers are bounded in the model checking tool Afra, specified by the arguments of reactive class declarations in Fig. 3. The function $head(b)$ specifies the message that was inserted at first. In Rebeca, the order of delivery of messages matches the order of their sending. So, the network buffer is a set of unbound buffers, one for each actor. We assume that the network buffer has the function $getBuff(b,x)$ which returns a buffer of the pending messages in the network buffer b with the actor x as their receiver. The order of sending is preserved in this buffer. So, the function $head(getBuff(b,x))$ returns the first pending message sent to x.

Semantic Rules The semantic rules derived from the generic framework are given in Table IV. We only include the rules with variation points. The variation points are fixed according to the Rebeca semantics. The local state of an actor is defined by the triple (e,b,π) where $domain(e)$ contains state variables, known rebecs names, and $self$ ($e(self)$ is the name of rebec). Actors can send messages to themselves by using the reserved word $self$. The function $GenerateMsg$ in the rule SEND of Table II generates a triple in the form of $(e(self), n, v)$ for the statement v!n, where n refers to a message server name. As the actor buffers are unbounded, the rule RECEIVE has no limitation and its $?rcvPolicy$ is set to true. The actor buffer take policy chooses the message first inserted into the buffer, denoted by $head(b)$ as shown in the rule TAKE.

At the network level, the environment is empty. In the rule RECEIVE, the network does not change the received message, i.e., $tag(e,m) = m$. In the rule TRANSFER, the transfer policy of messages is FIFO for each receiver who has at least one pending message in its buffer. The selection among those receivers is nondeterministic. At the composition level, the environment is empty. The scheduling policy for executing actors is nondeterministic, $?schedPolicy(e,s,x) = true$ in the rule ACTORPROG I. There is no update upon execution and termination of a method in rules ACTORPROG I/III, so $start(e,x) = e$, and $finish(e,x) = e$. As a result, the assumptions and conclusions of the three rules for actor progress in Section III-B3 are the same (except for the labels of the transitions). We represent them by one rule ACTORPROG by using the label β in Table IV.

B. Lingua Franca

The Lingua Franca (LF) language is a language for modeling distributed real-time systems [18]. An LF model is a collection of reactors and how they are connected together. Reactors are deterministic actors including methods that are

```
1  reactor Monitor(max:int(10))
       {
2    output notify: bool(false);
3    logical action temp;
4    reaction(temp)->notify {=
5      if(temp > max)
6        SET(notify, true);
7    =}
8  }
9  reactor Alarm {
10   input notify:bool;
11   reaction(notify) {=
12     //Play the alarm
13   =}
14 }
15 main reactor System {
16   m = new Monitor(25);
17   a = new Alarm();
18   m.notify -> a.notify after
           1 sec;
19 }
```

Fig. 4: A model of a monitor and an alarm specified in LF

called reactions. Reactions are invoked in response to triggered events, based on discrete event semantics of Ptolemy [19]. The parameters of a reaction are the input ports or clock variables. Messages (events) are timestamped, and their corresponding reactions are invoked based on the order of their timestamps. Note that events with identical timestamps (logically simultaneous) are handled in a deterministic order based on the order of the definition of their reactions in the reactor. Reactors have state variables, input/output ports, and actions, i.e. variables that their values are assigned externally. The composition of reactors is defined in the "main" reactor of the model by binding of input and output ports. An output port may be connected to multiple input ports, but an input port can only be connected to one output port [8].

Example 2: An LF implementation of the Rebeca model of Fig. 3 is presented in Fig. 4. As shown in line 18 of Fig. 4, the communication delay of 1 unit of logical time is associated with the communication link between the alarm and monitor. The notify input port of Alarm is defined in line 10 and upon any change to the value of this variable, its corresponding reaction (lines 11 to 13) is executed. In Monitor, the value of temp is set from the outside of the model, as it is defined as a logical action variable. Checking the value of the received temperature and notifying the alarm is presented in lines 5 and 6. Note that setting a value to an output variable takes place using the SET method. The composition of reactors is presented in the main reactor, which contains instances of each of the reactors and connects their ports. The bodies of reactions are written in some target languages; in our example, they are written in C. The target code in the figure is delimited by $\{=$ and $=\}$.

The ID set includes the name of instantiated reactors, and $Name$ set includes the input ports, output ports, and logical actions. The set of statements in the body of reactions is the same as BABEL except for the send statement. In LF, a message is sent using $\mathsf{set}(\mathsf{v}, \mathsf{val})$ statement by which the value of the output port v is set to the value val.

Structure of Messages and Buffers Messages in LF are tuples of four parts: the receiving (or sending) reactor identifier, the port name, the sent value, and the message delivery time, i.e., $Msg = ID \times Var \times Value \times (\mathbb{N} \times \mathbb{N})$. Time in LF is a pair of type $\mathbb{N} \times \mathbb{N}$. For a given time pair (st, ms), st, and ms are called the system time and micro-step, respectively. We use dot notation $m.prt$, $m.val$, $m.time$ to access the corresponding receiving port identifier, the sent value, and the delivery time of a given message m.

The buffers of reactors are bounded buffers with a priority policy based on the order of reactions in the code. The size of this buffer is the same as the number of input ports. The function $head(b)$ specifies the message that has the highest priority. The buffer of the network in LF also follows a priority policy based on the time tags of messages. The function $head(b)$ nondeterministically specifies a message among the ones with the smallest time tag.

Semantic Rules The semantic rules derived from the generic framework are given in Table V. We only include the rules with variation points, and the variation points are determined based on the LF semantics. In LF, a reactor has a fixed number of input ports and in each time step only one value can be set for an input port. This way, overflow does not happen and $?rcvPolicy$ in RECEIVE is set to true. In the TAKE rule, a reactor chooses an element from the head of its buffer. The local state of a reactor is defined by the triple (e, b, π) where $domain(e)$ contains $this$ and state variables of the reactor, where $e(this)$ is the name of the reactor. The function $GenerateMsg$ in the rule SEND of Table II generates a tuple in the form of $(e(this), p, v, t)$ for the statement $\mathsf{set}(\mathsf{p}, \mathsf{v})$, where p refers to an output port name, v refers to a value which is set for the output port, and t is the transmission time, initially set to $(0, 0)$.

At the network level, the environment includes variables now and *map*, and $e(now)$ shows the time of the network. The variable map is in the form of $ID \times Name \rightarrow ID \times Name \times \mathbb{N}$ and shows the topology of the LF model, i.e. bindings among input/output ports together with their communication delays. The map function maps its given sender reactor id and output port id to a tuple that contains the receiver reactor id, the input port id, and its communication delay time. The RECEIVE rule receives a message and updates the elements of the message by using tag function before its insertion into the network buffer. This function replaces the sending id and output port with the receiver reactor id and input port based on the network configuration (map) of the model. It also updates the time tag; if the message is sent via a connection link with a communication delay of zero, the micro-step part of the time tag of the message has to be set to the next micro-step. If the connection link has a communication delay d, the transmission time is set to $(now + d, 0)$. Function $tag(e, m)$ is defined as:

$$\begin{cases} (a', p', v, (t.st, t.ms + 1)), & map(a, p) = (a', p', 0) \\ (a', p', v, (t.st + t', 0)), & map(a, p) = (a', p', t') \wedge t' \neq 0 \end{cases}$$

where a, p and a', p' are $m.rcv$ and $m.ptr$ before and after the mapping, respectively, $v = m.val$, $map = e(map)$, and $t = e(now)$. In the rule TRANSFER, the network policy for dispatching messages is based on their time tags. The transfer policy delivers messages that their transmission time $(m.time)$ is the current time.

In actor-based languages with the concept of time like Timed Rebeca [3], realtime ABS [20], and realtime Creol [21], the global time evolves in agreement with actors and

TABLE V: The semantic rules of LF. $\beta = \{\tau, end\}$.

Actor	(RECEIVE) $(e,b,\pi) \xrightarrow{m?} (e, insert(b,m), \pi)$ (TAKE) $\dfrac{contains(b,m) m = head(b)}{(e,b,\epsilon) \xrightarrow{m} (e, remove(b,m), Method(m.ptr))}$ (TIMESYNC) $\dfrac{b = \emptyset}{(e,b,\epsilon) \xrightarrow{t} (e[now \mapsto t], b, \epsilon)}$
Network	(TRANSFER) $\dfrac{contains(b,m) m.time = e(now)}{(e,b) \overset{m!}{\rightsquigarrow} (e, remove(b,m))}$ (TIMESYNC) $\dfrac{head(b).time \neq e(now)}{(e,b) \overset{t}{\rightsquigarrow} (e[now \mapsto t], b)}$
Composition	(ACTORPROG I) $\dfrac{s(x) \xrightarrow{m} (e', b', \pi') \forall y \in ID \cdot e(pty(y)) > e(pty(x)) \Rightarrow (s(y) = (e'', \emptyset, \epsilon)) \nexists m' \in Msg \cdot n \overset{m'!}{\rightsquigarrow} n'}{(e,s,n) \xrightarrow{m} (e, s[x \mapsto (e', b', \pi')], n)}$ (ACTORPROG) $\dfrac{s(x) \xrightarrow{\beta} (e', b', \pi')}{(e,s,n) \xrightarrow{\beta} (e, s[x \mapsto (e', b', \pi')], n)}$

TABLE VI: Template rules for addressing the timed actor-based languages

Actor	(TIMESYNC) $\dfrac{?synCondition(e,b,t)}{(e,b,\pi) \xrightarrow{t} (e[now \mapsto t], b, \pi)}$
Network	(TIMESYNC) $\dfrac{?synCondition(e,b,t)}{(e,b) \overset{t}{\rightsquigarrow} (e[now \mapsto t], b)}$
Comp.	(TIMEPROG) $\dfrac{n \overset{t}{\rightsquigarrow} n' \forall x \in ID \cdot s(x) \xrightarrow{t} s'(x)}{(e,s,n) \xrightarrow{t} (e[now \mapsto t], s', n')}$

the network. We extend the framework to address real-time features by introducing two rules of Table VI. Both actors and network have a local variable now in their environment which shows their local times. The rule TIMESYNC at the actor and network levels makes the progress of time explicit by updating now. The rule has an assumption, denoted by $?synCondition(e,b,t)$, for identifying conditions to indicate when time with the amount of $t \in \mathbb{N}$ can progress. The variable now in the environment of the composition level shows the global time. The rule TIMEPROG at the composition level advances the global time in agreement with actors and the network. Generally speaking, actors and the network agree that time advances when they cannot progress. In LF, actors cannot progress when they have no message to handle. So, the assumption $SynCondition$ is $b = \emptyset$ in the rule TIMESYNC at the actor level. The network cannot progress when it has no message to deliver at the current time, So, the assumption $SynCondition$ is $head(b).time \neq e(now)$ in the rule TIMESYNC at the network level.

At the composition level, the environment includes the mapping $pty : ID \to \mathbb{N}$ which assigns a priority to reactors. This priority is obtained by a topological sort of the reactors in the model. The scheduling policy for executing reactors is based on their priority and the state of the network. A reactor can be executed if the network has no message to dispatch ($\nexists m' \in Msg \cdot n \overset{m'!}{\rightsquigarrow} n'$) and there is no reactor with a higher priority with an unhandled message, i.e., $\forall y \in ID \cdot e(pty(y)) > e(pty(x)) \Rightarrow s(y) = (e'', \emptyset, \epsilon)$. There is no update upon execution and termination of a reactor message handler in rules ACTORPROG I/III, so $start(e,x) = e$, and $finish(e,x) = e$. As a result, the assumptions and conclusions of ACTORPROG II and ACTORPROG III become the same except for the labels of the transitions. We represent them by one rule ACTORPROG by using the label β in Table V.

V. BRIEF OVERVIEW OF FEW OTHER LANGUAGES

Here we briefly show how we can use the framework by adding or changing rules to capture the semantics of (a subset of) Erlang, Akka, and ABS actor languages.

Erlang [5] is designed for building massively scalable software systems. It supports user-defined pattern matching for taking messages from a mailbox via receive blocks: patterns in each receive block are sequentially matched against the messages and their time order in the mailbox. This user-defined pattern is addressed by defining $?takePolicy$ in our framework that identifies which messages can be handled based on the state of the actor. There is a possibility for a message to be lost in Erlang. We can model message loss by splitting the RECEIVE rule at the network level into two rules as shown in Table VII. The new extension ignores the received message m by not inserting m into the buffer (RECEIVE II). Actors with the same priority are executed using round-robin scheduling. At the composition level, we need to have a set of buffers. Each buffer stores the actor identifiers with the same priority level. We need to adjust the scheduling policy $?schedPolicy$ to result in selecting the buffer with the highest priority. The scheduling policy should also consider a round-robin scheduler for each buffer as well. The other semantic rules of Erlang, as far as we studied, are similar to Core Rebeca.

Akka is an actor framework for the Java and Scala languages [22]. From what we understood, Akka provides three different message delivery policies which are exactly-once, at-most-once, and at-least-once. Exactly-once delivery means the message can neither be lost nor duplicated. The semantic rule of Akka for exactly-once policy is the same as Core Rebeca. In the at-most-once policy there is no guarantee for message delivery. The RECEIVE II rule in Table VII can be used to express the possibility of missing a message for this policy. Finally, for the case of at-least-once policy, for each message potentially multiple attempts are made to deliver it. We split the rule TRANSFER at the network level to make the possibility

TABLE VII: Extensions of the network level rules to support message loss and multiple deliveries.

$$\text{(RECEIVE I)}\ (e,b) \overset{m?}{\rightsquigarrow} (e, insert(b,m))$$

$$\text{(RECEIVE II)}\ (e,b) \overset{m?}{\rightsquigarrow} (e,b)$$

$$\text{(TRANSFER I)}\ \frac{contains(b,m)}{(e,b) \overset{m!}{\rightsquigarrow} (e, remove(b,m))}$$

$$\text{(TRANSFER II)}\ \frac{contains(b,m)}{(e,b) \overset{m!}{\rightsquigarrow} (e,b)}$$

of multiple deliveries by not removing a message from the buffer.

The ABS modeling language [4] allows to compose actors into concurrent object groups (cogs), and cogs are units of concurrency. Conceptually, each cog has a dedicated execution thread that is shared among its actors. An idle actor can take one of its received messages if the thread which is associated with its corresponding cog has not been assigned to another actor of that cog. We can address this concept by adding a mapping $cog : ID \rightarrow Bool$ to the environment of the composition level. For an actor x, the true value of $cog(x)$ indicates that the thread of the cog of x is assigned to an actor. An actor x can take a message if its thread is not assigned (i.e., $\neg cog(x)$). Upon starting and termination of an actor method, $cog(x)$ is updated accordingly (i.e., *start* and *finish*).

As we discussed, Transparent Actor highlights the design decisions (features) of languages by explicating the policies chosen at the three levels of our framework. These policies show the design decisions at variations points. We compare different actor languages in Table VIII based on these policies. Each column characterizes a language by a feature vector.

VI. Experimental Comparison of Languages

We implemented Babel in the Maude language [23] as the proof of concept. We modeled the semantics of ABS, Rebeca, Timed Rebeca, and LF as an extension of Babel. The models are accessible through this link: https://github.com/mirgit/Babel. We employ the Maude implementations of these languages to empirically compare them based on some properties related to actor languages. Using our Babel implementation, one can derive a naive prototype of its language based on its selected features and then investigate its properties.

Each model of a language consists of three files; `basics`, `buffer`, and `semantics`. The file `basics` contains the basic structures like variable environments. Buffers and policies are exclusively defined for each targeted language. The `semantics` file imports `buffer`. The main idiosyncrasies of a language are demonstrated in the file `buffer`. This file should provide two modules. The first module `Buffers` defines the message structure, as it is not fixed in Babel, and two buffer structures, `bufferNet` and `bufferActor`. These buffers need to support insert and remove functions to be used by semantic rules. The second module `Policies` defines equations that are used to fill variation points in the rules. For example, take policy, in Rebeca, is `head(buffer)` since a rebec's buffer is modeled as a queue. The same policy in ABS is taking an arbitrary message from the buffer.

Finally, all the semantic rules are defined in the file `semantics`. This file also can vary slightly across different languages in order to add extra rules like time progress or particular statements of a language e.g. `set(p,v)` for Lingua Franca which sets the value of an output port `p` to `v`. Transition relations (with assumptions) are modeled by (conditional) rules. For example, the Take rule in Rebeca is modeled as a conditional rule:

```
conditional rule [TAKE] :
  < e,b,ϵ> => <e[sender−>sender(m)], remove(b,m),
                                handler(m.name)>
  if m := takePolicy(e,b) ∧ not(empty(b)).
```

This rule means that a transition is possible when an actor has an empty statement list ϵ; as a result, a message `m` is removed from its buffer b and `handler(m.name)` is added to its statements if `m` is the message that `takePolicy` returns. We remark that instead of modeling the predicate $takePolicy(e,b,m)$, we have modeled it as a function that returns the message that can be taken with respect to b and e.

We considered three properties to compare the modeled languages. For each property, we conducted litmus tests to illustrate the answer for the languages, inspired by [24]:

P1: Are two messages that are sent from one actor to another handled in the same order as their sending order? We call this the *in-order* property.

Litmus test: Actor A sends two consecutive messages m_1 and m_2 to actor B. We check if B handles m_1 before m_2. This property depends on the actor's take policy and the network's transfer policy. Changing any of these policies may affect the order of handling of messages in B.

We specified two types of actors A and B, such that A sends m_1 and then m_2 to B in its constructor. We use the search command of Maude to inspect the state space of the model for a state in which actor B handles m_2 before m_1. Table IX shows the search results. Maude found at least one state for Timed Rebeca and ABS in which the order of messages is reversed. In contrast, the result of the search for Rebeca is "No Solution", denoted by "Yes" in Table IX. This is expected since the buffers of actors and the network in Rebeca are designed as a queue and a set of queues, respectively. For LF, we experimented this test with different priorities for m_1 and m_2 ports. Only in cases where in B the input port priority for m_1 is higher than m_2, it handles m_1 first. As Timed Rebeca and LF are timed languages, we considered the same communication delay for both messages in their experiments. Due to nondeterminism in the transfer policy for messages with equal timestamps in Timed Rebeca, both message delivery orders are possible (and then the FIFO take policy does not change them). However, the scheduling policy of LF forces the messages to be delivered first to B. Depending on the port priorities, the take policy results in only one handling order.

TABLE VIII: Variation points and selected policies across different languages

	Rebeca	**ABS**	**LF**	**Timed Rebeca**
Actor Buffer	Unbounded FIFO	Multiset	Priority Queue	Unbounded Bag
Actor's receive policy	Always enabled, no restriction	Always enabled, no restriction	Always enabled, no restriction	Always enabled, no restriction
Actor's take policy	Takes the head of its message queue	Takes from the multiset nondeterministically	Takes the message with the highest priority	Takes the messages with the least time tag, if more than one, then choose nondeterministically
Network Buffer	Set of Unbounded FIFOs	Set of Multisets	Priority Queue	Unbounded Bag
Network's transfer policy	Dispatch the head of the queue of any actor that has a message, nondeterministically	Dispatches the messages of any actor, nondeterministically	Dispatches the message with the least time tag and highest priority	Dispatches messages with the least time tag, if more than one, then choose nondeterministically
Composition scheduling policy	Choose an enabled actor for execution, nondeterministically	Choose an enabled actor for execution, if the thread corresponding to its group is free	Choose an enabled actor for execution, if no actor with higher priority nor network can progress	Choose an enabled actor for execution, nondeterministically
Time synchronization in timed models	-	-	Time can progress if neither network nor actor can progress	Time can progress if neither network nor actor can progress, or when a suspended actor (because of a delay) can resume

P2: Is the scheduling of actors fair? We call this the *actor scheduling fairness* property. Our notion of fairness is absolute (a.k.a. impartiality), which means every process should be executed infinitely often [25].
Litmus test: Actor A sends a message to actor B and B sends a response to A and the actors repeat this message exchange infinitely. Actor A has a similar loop with another actor called C. The desired behavior is that both loops proceed and B or C never get starved. This property depends only on one policy: the scheduling of actors (note that in all the languages that we consider no message is lost, i.e., the network transfer policy is not lossy). We created A, B, and C, and their methods in our Maude model. We used LTL model-checking provided by Maude to see if one actor, e.g. C, gets starved. Since the properties in Maude are state-based, we made a small change in the PROGRESS rules of Table I to set the flag `prog` when they make a progress. We then checked fairness by the command:

```
reduce modelCheck( initialState ,
                   []<> progb ∧ []<> progc).
```

This means starting from `initialState`, always eventually `progb` and `progc` become `true`. The `progb` and `progc` flags denote setting the `prog` flag in the actors B and C. The result is false for all of our considered languages, and a counter-example is returned in which only one of the loops $\{A, B\}$ or $\{A, C\}$ is executed. Up to this point, we assumed no delay in our examples. If we add a computation or communication delay in timed languages, the model-checker succeeds. That is because, in both Timed Rebeca and LF, time progresses after all actors have executed their current statements, and there is no message left to handle. In our experiments, this property depends on both scheduling and time synchronization policies.

	Rebeca	ABS	Timed Rebeca	Lingua Franca
in-order	Yes	No	No	depends on port priorities
actor scheduling fairness	No	No	No, with delay, Yes	No, with delay, Yes
message interleaving	6	6	6	1

TABLE IX: The result of experiments

P3: If a group of actors sends a message to an actor, what permutations of messages are possible in the destination actor? We call this the *message interleaving* property.
Litmus test: Consider three actors, A, B, and C. Each actor sends a message to actor D. The actor D stores the order of the messages in the variable `ord`. For example, if D receives messages from B, A and C, the result of `ord` will be `[B,A,C]`. There can be 6 different orders of actors in `ord`. We are interested to know how many of those orders are actually possible in each language. By using the search command of Maude, we check the number of possible final states of the program. As shown in Table IX, all of the tested languages except LF let messages shuffle in any order and give all possible combinations of messages. In contrast, LF always gives one unique result independent from sending orders due to its deterministic execution, which stems from predefined priorities for ports.
Similar to P1, this property focuses on the order of handling messages. As multiple sending actors are involved, besides the take and transfer policies, the investigated property also depends on the scheduling policy. In these experiments, we considered equal communication delay for all messages. The nondeterminism in scheduling and transfer policies in Rebeca and ABS results in any orders for message delivery. In timed languages, first, all messages are delivered to the network and after time progresses, messages are delivered. The nondeterminism in the transfer policy of Timed Rebeca for

messages with equal timestamps makes any order of delivery possible (which is not changed by the take policy). However, the scheduling policy of LF enforces D to handle its messages only when all are delivered to its buffer. So, the priority of ports defines the resulting unique permutation. When the communication delays differ, the order of handling is also affected by the delays.

VII. Conclusion

We show how we can use the Transparent Actor model to highlight the features and design decisions of different actor-based languages. We defined BABEL and its formal semantics to build a framework for designing actor-based languages. This not only helps when designing a new actor-based language but also helps when building asynchronous event-based distributed software systems. While writing the formal semantics of BABEL as SOS rules we needed to revise the framework many times to remove ambiguities and inconsistencies. Using our framework, one can infer specific properties of a language based on its selected features. We believe that our framework helps the formal methods community in understanding crucial features of actor-based languages related to communication and coordination, and our intuitive discussions and experiments help the language and software designers to better grasp the decision points and feature interactions.

Acknowledgments: The work of Marjan Sirjani is supported by SSF Serendipity project, KKS DPAC Project (Dependable Platforms for Autonomous Systems and Control), and KKS SACSys Synergy project (Safe and Secure Adaptive Collaborative Systems). The authors would like to thank the anonymous reviewers for their helpful comments on this manuscript.

References

[1] C. Hewitt, "Viewing control structures as patterns of passing messages," *Artificial Intelligence*, vol. 8, no. 3, pp. 323–364, 1977. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0004370277900339

[2] G. A. Agha, *ACTORS - a model of concurrent computation in distributed systems*, ser. MIT Press series in artificial intelligence. MIT Press, 1990.

[3] M. Sirjani and E. Khamespanah, "On time actors," in *Theory and Practice of Formal Methods - Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*, vol. 9660, 2016, pp. 373–392. [Online]. Available: https://doi.org/10.1007/978-3-319-30734-3_25

[4] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen, "ABS: A core language for abstract behavioral specification," in *Proceedings of the 9th International Symposium on Formal Methods for Components and Objects*, vol. 6957, 2010, pp. 142–164.

[5] J. Armstrong, "A history of Erlang," in *Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming Languages*. ACM, 2007, pp. 1–26. [Online]. Available: https://doi.org/10.1145/1238844.1238850

[6] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger, "The scala language specification," 2004.

[7] "Go programming language homepage," https://golang.org, accessed: 2022-08-01.

[8] M. Lohstroh, Í. Í. Romeo, A. Goens, P. Derler, J. Castrillón, E. A. Lee, and A. L. Sangiovanni-Vincentelli, "Reactors: A deterministic model for composable reactive systems," in *Proceedings of the 9th International Workshop on Cyber Physical Systems. Model-Based Design*, vol. 11971. Springer, 2019, pp. 59–85.

[9] F. S. de Boer, V. Serbanescu, R. Hähnle, L. Henrio, J. Rochas, C. C. Din, E. B. Johnsen, M. Sirjani, E. Khamespanah, K. Fernandez-Reyes, and A. M. Yang, "A survey of active object languages," *ACM Computing Surveys*, vol. 50, no. 5, pp. 76:1–76:39, 2017. [Online]. Available: https://doi.org/10.1145/3122848

[10] S. Krishnamurthi, "Programming languages: Application and interpretation," https://cs.brown.edu/~sk/Publications/Books/ProgLangs/, 2007.

[11] R. K. Karmani, A. Shali, and G. Agha, "Actor frameworks for the JVM platform: a comparative analysis," in *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, 2009, pp. 11–20. [Online]. Available: https://doi.org/10.1145/1596655.1596658

[12] R. Henderson and B. Zorn, "A comparison of object-oriented programming in four modern languages," *Software: Practice and Experience*, vol. 24, no. 11, p. 1077–1095, 1994. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380241106

[13] M. Appeltauer, R. Hirschfeld, M. Haupt, J. Lincke, and M. Perscheid, "A comparison of context-oriented programming languages," in *International Workshop on Context-Oriented Programming*. ACM, 2009, pp. 6:1–6:6. [Online]. Available: https://doi.org/10.1145/1562112.1562118

[14] A. Elyasaf and A. Sturm, "Towards a framework for analyzing context-oriented programming languages," in *Proceedings of the 13th ACM International Workshop on Context-Oriented Programming and Advanced Modularity*, 2021, pp. 16–23. [Online]. Available: https://doi.org/10.1145/3464970.3468414

[15] M. Sirjani, A. Movaghar, A. Shali, and F. S. de Boer, "Modeling and verification of reactive systems using rebeca," *Fundamenta Informaticae*, vol. 63, no. 4, pp. 385–410, 2004.

[16] M. Sirjani, "Rebeca: Theory, applications, and tools," in *Proceedings of the 5th International Symposium on Formal Methods for Components and Objects,*, vol. 4709, 2006, pp. 102–126. [Online]. Available: https://doi.org/10.1007/978-3-540-74792-5_5

[17] "Afra homepage," http://rebeca-lang.org/alltools/Afra, accessed: 2022-08-01.

[18] M. Lohstroh, M. Schoeberl, A. Goens, A. Wasicek, C. Gill, M. Sirjani, and E. A. Lee, "Actors revisited for time-critical systems," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, p. 152.

[19] C. Ptolemaeus, *System Design, Modeling, and Simulation: Using Ptolemy II*. Ptolemy. org Berkeley, CA, USA, 2014.

[20] E. B. Johnsen, R. Schlatte, and S. L. T. Tarifa, "Integrating deployment architectures and resource consumption in timed object-oriented models," *Journal of Logical and Algebraic Methods in Programming*, vol. 84, no. 1, pp. 67–91, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352220814000479

[21] F. S. de Boer, M. M. Jaghoori, and E. B. Johnsen, "Dating concurrent objects: Real-time modeling and schedulability analysis," in *Proceedings of the 21st International Conference on Concurrency Theory*, vol. 6269, 2010, pp. 1–18.

[22] "Akka homepage," https://akka.io, accessed: 2022-08-01.

[23] "The Maude system," http://maude.cs.illinois.edu, accessed: 2022-08-01.

[24] S. Mador-Haim, R. Alur, and M. M. K. Martin, "Litmus tests for comparing memory consistency models: How long do they need to be?" in *Proceedings of the 48th Design Automation Conference*, 2011, p. 504–509. [Online]. Available: https://doi.org/10.1145/2024724.2024842

[25] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Transactions on Programming Languages and Systems*, vol. 8, no. 2, p. 244–263, 1986. [Online]. Available: https://doi.org/10.1145/5397.5399