Actors and Logical Analysis of Interactive Systems

Ian A. Mason [1,2]

School of Mathematics, Statistics, and Computer Science
University of New England
Armidale, NSW 2351, Australia

Carolyn L. Talcott [3]

Computer Science Laboratory
SRI International
Menlo Park, CA 94025, USA

Abstract

Formalization in a logical theory can contribute to the foundational understanding of interactive systems in two ways. One is to provide language and principles for specification of and reasoning about such systems. The other is to better understand the distinction between sequential (turing equivalent) computation and interactive computation using techniques and results from recursion theory and proof theory. In this paper we briefly review the notion of interaction semantics for actor systems, and report on work in progress to formalize this interaction model. In particular we have shown that the set theoretic models of the formal interaction theory have greater recursion theoretic complexity than analogous models of theories of sequential computation, using a well-known result from recursion theory.

Key words: Actors, interaction semantics, Feferman theories, recursion theory.

1 Introduction

An important challenge for foundations of interactive computation is to provide a basis for specification and reasoning about interactive systems, eventually leading to principled methods for design, implementation, and deployment of such systems. A foundation should identify primitives for specification that in combination with

[1] The authors wish to thank Michael Beeson for helpful discussions, and the anonymous reviewers for helpful criticisms. The work was partially supported by NSF grant CCR-023446.

[2] Email: iam@turing.une.edu.au

[3] Email: clt@cs.stanford.edu

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

appropriate logical constructs lead to specification languages and logics for reasoning about the behavior of specified systems, and means of checking (statically or dynamically) that a given system meets its specification. Another challenge is to better understand the distinction between interactive computation and computability in the sense of Turing machines or lambda calculus. Intuitively it seems clear that interactive computation is not equivalent to Turing computation. The question is how to make this intuition more precise.

Traditionally, notions of computability are strongly tied to complexity of fragments of first-order and other logics. We propose that one way to begin to understand the distinction between sequential (Turing equivalent) computation and interactive computation is to understand the power of the logics needed to formally represent models of interactive computation. To explore this idea in more depth, we examine a formalization, currently being developed, of the *interaction semantics* of actor systems. To be clear, we are not proposing a new model of interactive computation, but rather analyzing the expressive power of an existing model in comparison to sequential computation.

To set context, in Section 2 we briefly review some of our work on actor semantics that lead to this notion of interaction semantics. In Section 3 we briefly summarize our work on formal theories for sequential computation based on Feferman's theories for formalizing constructive mathematics, and then describe a Feferman style formal theory of interaction semantics. The theories of sequential computation formalize and reason about input/output relations and their properties. These theories have natural, term generated, recursively enumerable models that capture the intended semantics. For example, Feferman's theories such as IOC_Λ and IOC_λ [10] all have natural recursion theoretic models where the space of total functions $\mathbf{Nat} \to \mathbf{Nat}$ is interpreted as the total recursive functions, and the corresponding partial function space as the partial recursive functions. In contrast, theories of interactive computation must formalize the interactions a system may have with its environment, where nothing is known about the behavior of entities in the environment. In Section 4 we show that recursively enumerable models are not adequate to capture the intended interaction semantics of actor systems using a well-known result from recursion theory.

2 Actor semantics

The actor model [13,12,2,3] is a model of distributed computation based on the notion of independent computational agents, called actors, that interact solely via message passing. An actor can create other actors; send and receive messages; and modify its own local state. An actor can only affect the local state of other actors by sending them messages, and it can only send messages to its acquaintances—either actors whose names it was given upon creation, or names it received in a message or names of actors it created. Actor semantics admits only fair computations, which in the simplest case means reliable message delivery.

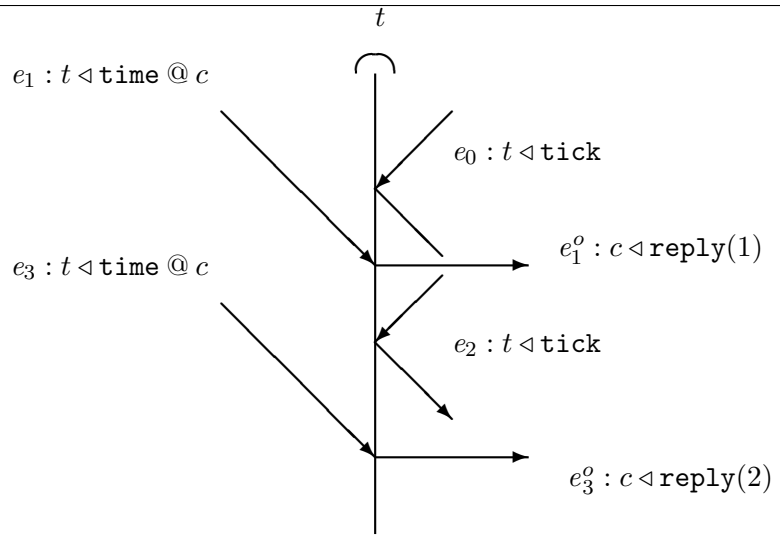

Fig. 1. A Ticker event diagram. The vertical line, t, is the local time-line for the *ticker* actor t. The diagram illustrates: the arrival order (incoming arrows) at t—$e_0 < e_1 < e_2 < e_3$; and the activation order (outgoing arrows)—$e_1 < e_1^o$, $e_3 < e_3^o$. Like Email, messages need not arrive in the order sent: both $e_1^o < e_3^o$ and $e_3^o < e_1^o$ are possible orders of arrival of the messages at c.

2.1 Traditional actor semantics

The central concepts of traditional actor semantics [5] are the partial order of events (an event being a message receipt), acquaintance laws (who can come to know whom), and fairness of computations. The essential properties are captured in the notion of event diagram [11,6] characterizing the possible computations of an actor system.

In the following, we will use the much overworked *Ticker* actor to illustrate concepts. A Ticker actor, t, has its own integral notion of time n, and responds to two types of messages: a `tick` request, and a `time` request. A Ticker processes requests as follows:

- upon receiving a `tick` message ($t \triangleleft \texttt{tick}$) a Ticker increments n, and sends itself a new `tick` message.
- upon receiving a `time` request from an actor c ($t \triangleleft \texttt{time} \mathbin{@} c$) a Ticker sends c a message $\texttt{reply}(n)$ where n is its current notion of time.

Figure 1 shows the event diagram for a possible Ticker computation, where the Ticker t interacts with some external actor c.
What might we say and/or prove about a Ticker?

- Every `time` request $t \triangleleft \texttt{time} \mathbin{@} c$ gets a reply $c \triangleleft \texttt{reply}(n)$ for some number n.
- If there is always another `time` request, then for any n there is a reply $c \triangleleft \texttt{reply}(n')$ with $n < n'$ among the messages sent.

Not much else can be said without imposing some additional causality constraints on the events.

2.2 Actor Theories, Components and Interaction Semantics

Although concerned with who could receive messages and when, the early work on actor semantics did not provide a notion of interface, or a mechanism for abstracting a group of actors as an interacting component. A theory of program equivalence for actors was developed in [4] using a lambda calculus based programming language. To define the semantics, a notion of actor system *interface* was introduced. An interface consists of two finite sets of actor names: *receptionists* (system actors whose names are known externally, and thus can receive messages from the environment), and *externals* (environment actors whose names are known by some system actor, and thus can be sent messages from a system actor). An actor system configuration is then a collection of actors and messages encapsulated by an interface. The operational semantics for the language is defined by a transition system on actor system configurations. A transition is either a computation step by a system actor processing a message, or input of a message to a receptionist from the environment, or output of a message to an external actor. A notion of observational equivalence was defined in the usual way as indistinguishability in all contexts [7]. In this setting, a context is a closed configuration with a hole to be filled by the program. The observation to be made is whether or not a specific message is emitted by the context. Equivalence of configurations is defined similarly. A number of techniques were developed to prove equational laws for programs and equivalence of configurations. These techniques allowed one to focus on messages sent and received by a configuration under consideration, to treat computations (transition sequences) modulo an equivalence relation corresponding to equating different linearizations of the same event partial order, and to collapse multiple steps of an actor into one.

Generalization of these reasoning techniques lead to two new ideas presented in [25,28]. One is the notion of *interaction semantics* of an actor system configuration as the set of possible interaction paths (sequences of input/output interactions) that could result from interaction of the system with an arbitrary environment. The other is the notion of actor theory as a rewriting logic based specification of the behavior of actors. Rewriting logic provides an operational semantics of actor system configurations in the form of derivations viewed as computations. These derivations satisfy the equations used in [4], and contain sufficient information to also derive the interaction semantics.

In [27] an actor component algebra is defined to study the compositionality of the different forms of specification and semantics of actor systems. The algebra makes minimal assumptions about what is being composed, limiting the operations to interface restriction, parallel composition, renaming, and an identity component. In particular the ability to prefix an action to a component behavior is not assumed. Using morphisms between the different syntactic and semantic structures, compositionality of computational and interaction semantics was shown, thus justifying thinking of interaction semantics as a denotational semantics (without need for CPOs and limits!).

Two observations about interaction paths are in order. Firstly, while event diagrams give a true concurrency model, they talk about internal events. Interaction paths correspond to what can be observed from the outside, from an arbitrary point of view. Thus all orderings of independent events will be possible, but in general it will not be possible to infer causality beyond what is implied by local observations. Secondly, although only fair computations are used to define the set of interaction paths for a given actor configuration, there are no explicit fairness constraints on what interaction sequences can form an interaction path. The only constraints are that actor acquaintance laws are obeyed. For example, a message cannot be input to an actor that is not a receptionist, and an external actor is only known if it was known initially or introduced in an incoming message.

2.3 Specifying interaction paths

Actor theories are one way to specify the possible interactions of a component. Some advantages of such specifications are that they are executable and composible. On the other hand an actor theory specifies how a system works, not what it should do. For example, one might want to express that certain requests (incoming messages) are alway answered with a reply meeting given constraints, or the results of processing particular messages in a specific order (thus giving a stronger guarantee for a requestor that always waits for a reply before sending the next request). We have explored two alternative methods for specifying the interaction semantics of a component; specification diagrams (SD) [22,23,29], and mathematical specifications.

SD is a language with both textual and graphical representations. SDs can express interaction patterns of sequencing, choice and concurrency, (similar to regular expressions over interactions). SD can also express requirements on the environment—messages that must/must not be sent, and internal states that should or should not be reached. A restricted subset of SDs correspond to executable specifications in the spirit of actor theories. In general a SD may be partial (specifying behavior only for inputs of interest) and may not be realizable, for example requiring behavior to depend on the future, not just the past! Partial specifications are appealing as they allow one to concentrate on interactions with the intended environment. However, compositionality is not guaranteed, since a component may meet the specified constraints but exhibit behavior that leads other partially specified components to go wrong when subjected to situations not provided for. A detailed comparison of SDs and interaction semantics with other formalisms for concurrent, interactive computation can be found in [23].

Mathematical specifications specify a set of interaction paths by mathematical formulas with variables ranging over paths, messages, and other relevant entities. They are informal but rigorous and written in a stylized way. Notation and principles have been developed for using event diagram concepts to constrain interaction paths to those compatible with a set of event diagrams. That is, we can specify a system by saying it is indistinguishable from one whose event diagram semantics

obeys the event diagram constraints. This is analogous to specifying a system by requiring it to be 'equivalent' to some simply defined system.

In the next section we describe a formal theory of interaction paths in which mathematical specifications can be represented as logical formulae. This is illustrated by $TickerMS$, a mathematical specification of a Ticker.

3 Variable Type Theories for Actors

When developing a formal theory the first question to ask is: *What do we want to represent?* Here we focus on formalization of semantic notions and their properties, a meta-logic, as a stepping stone to a logic of interaction. Thus we need to represent: actor system descriptions (behaviors, interfaces, configurations); operational semantics (transitions and fair computations); interaction semantics; and the satisfaction relation between a system description and a property of interaction paths

$$iC \models \Phi \;\Leftrightarrow\; [\![iC]\!] \subseteq [\![\Phi]\!].$$

Here iC is an actor configuration, with an explicit interface, and Φ is a sentence in our formal theory. The semantics $[\![iC]\!]$, or meaning, of the configuration iC is a set of interaction paths, each interaction path distilling the observable interactions from the actions that take place in a particular computation path. Analogously the semantics $[\![\Phi]\!]$, or meaning, of the formula Φ is the set of those interaction paths that satisfy it.

We continue our approach of using logical theories developed by Feferman to formalize constructive mathematics. These are 2-sorted classical theories called variable type theories in which both functions and data are objects of discourse in a first order setting, as are collections of such things (called classifications). Classifications provide a balance between expressive power and complexity, allowing one to represent inductively and co-inductively defined sets, computable function spaces and other sets of interest, all in a first-order setting.

In the spirit of Landin [16], the languages we have studied consist of lambda expressions augmented with operations for computational primitives of interest: control abstractions, memory allocation and access, actor creation and messaging, and so on. The sequential languages all have a transition system semantics with strong uniformity properties [26] and are called Landinesque languages. Earlier work on formal theories for sequential languages includes IOCC, VTLoE, and FLL. IOCC [24] is an adaptation of Feferman's IOC_λ [10] that formalizes continuations and control primitives such as Scheme's `call-cc`, providing axioms for reasoning about computations with continuations. VTLoE [15] was developed to reason about functional programs with effects, such as ML, Scheme, or Lisp. VTLoE was generalized to a logic, FLL, (Feferman-Landin Logic) [18] for reasoning about Landinesque languages. Adapting constructions of [8,9], it was shown in [24] that IOCC has term-based, recursively enumerable models.

3.1 Formalizing Interaction Semantics

The formalization uses Feferman's IOC_λ as a starting point. This formal system provides quantification over individuals and classifications. Individuals include lambda terms, and numbers. Classifications (briefly classes) are collections of individuals defined by comprehension $K = \{x \mid \psi(x)\}$. Constants, operations, axioms and rules for actor specific entities are then added, including: the Actor Communication Basis (ACB) that provides a basis for both the semantic and behavioral descriptions of actor configurations; interfaces; *interaction paths*, the elements of interaction semantics; configurations, specified by describing their constituent actor's behaviors; and computation paths, corresponding to single executions of a configuration.

Notationwise, $\mathbf{P}_\omega(X)$ is the set of finite subsets of X, while $\mathbf{M}_\omega(X)$ is the set of finite multisets from X, and $\emptyset$ is the empty set. Following logical tradition, the axioms are presented informally, with the understanding that it is not problematic to fill in details needed to be completely formal.

Actor Communication Basis.

The actor communication basis, ACB, is a direct translation of the rewriting logic formalization of actor theory [28]. It defines the basic language needed to talk about actor interactions. The basic sorts are actor names, $a \in \mathbf{A}$, message contents, $M \in \mathbf{Msg}$, and message packets, $a \triangleleft M \in \mathbf{MP} \cong \mathbf{A} \times \mathbf{Msg}$. The components of a message packet are call the target (an actor name) and the message contents which can be extracted by the two operations $target : \mathbf{MP} \to \mathbf{A}$, and $message : \mathbf{MP} \to \mathbf{Msg}$. We let mp range over $\mathbf{MP}$.

Interfaces.

Interfaces are used to encapsulate configurations, and interaction sequences. They consist of two finite sets of actor names, $(\rho, \chi) \in \mathbf{Iface}$, the *receptionists*, ρ, and the *externals*, χ. In other words $\mathbf{Iface} \cong \mathbf{P}_\omega(\mathbf{A}) \times \mathbf{P}_\omega(\mathbf{A})$. Recall that the receptionists are those internal actors (internal to a configuration) known externally, while the externals, are those external actors (external to a configuration) known internally.

Interaction paths.

Interaction paths are formalized by introducing a class constant $\mathbf{ISeq}$ of interaction sequences. $\mathbf{ISeq}$ is subject to axiomatic constraints, but is not defined by comprehension. Thus interaction sequences are not necessarily λ definable. In fact we will see below that they cannot all be λ definable. Mathematically, an interaction path, $ip \in \mathbf{IP}$, is a sequence of interactions annotated by an initial interface. An interaction, $io \in \mathbf{IO}$, is either the input or the output of a message packet: $\mathbf{IO} \cong \mathtt{in}(\mathbf{MP}) \cup \mathtt{out}(\mathbf{MP})$. Sequences of interactions, $\vartheta \in \mathbf{ISeq}$, are just functions from natural numbers into interactions, enriched with a silent tau transition: $\mathbf{ISeq} \cong (\mathbf{Nat} \to \mathbf{IO} \cup \{\tau\})$. Interaction paths are constructed from interfaces and

interaction sequences via $(\_)\_ : \mathbf{Iface} \times \mathbf{ISeq} \to \mathbf{IP}$. We write $\mathbf{IP}(\rho, \chi)$ for the set of interaction paths of the form $(\rho, \chi)\vartheta$.

Actors and Configurations.

An actor has the form $a : B$ where $a \in \mathbf{A}$ is the actor's name and B is the actors behavior. A behavior is a lambda term of the form $\lambda(a, M, \nu)e$, that takes a message packet, decomposed into target a, and contents M, and a function $\nu : \mathbf{Nat} \to \mathbf{A}$. The function ν is to be used to generate fresh names for any actors created: $\nu(0), \nu(1), \ldots$ are guaranteed to be fresh names. The body e must evaluate to a new behavior lambda together with a configuration consisting of any created actors and messages to be sent. A *configuration*, $C \in \mathbf{Conf}$, is a multiset of actors and messages. An interfaced configuration, $C \in \mathbf{Conf}$, is a configuration encapsulated by an interface. The operator $(\_)\_$ is overloaded to map an interface and a configuration to an interfaced configuration.

Computations.

Computations, like interaction paths are formalized by introducing a class constant $\mathbf{CP}$ of computation paths that is constrained by axioms but not defined by comprehension. To define computations, individual transitions are axiomatized as triples of the form $iC \stackrel{l}{\Longrightarrow} iC'$. Then a computation path is formalized as a sequence of transitions meeting certain semantic constraints. We write $\mathbf{CP}(iC)$ for the set of computation paths starting from the interfaced configuration iC.

To spare the reader more formalities we illustrate the formalization of the actor behaviors and operational semantics by some simple examples. that will be useful later on. The simplest actor behaviour is perhaps the sink. A sink simply accepts any message sent it, doing absolutely nothing in response.

We represent the behavior of the sink by the term $Sink$. Where

$$Sink = \lambda(a, M, \nu)(Sink, \emptyset)$$

Thus after accepting a message, with contents M, the actor's behavior remains unchanged, and no actors or messages are created in response.

Only slightly more challenging is the term describing the Ticker behavior. $Ticker(n)$ is a ticker actor behavior with local time n where

$$\begin{aligned} Ticker = \lambda n.\lambda(a, M, \nu)& \\ &\texttt{if}(M = \texttt{tick}) \\ &\quad \texttt{then}(Ticker(n+1), a \triangleleft \texttt{tick}) \\ &\quad \texttt{else if}(M = \texttt{time} \ @ \ c) \\ &\qquad \texttt{then}\ (Ticker(n), c \triangleleft \texttt{reply}(n)) \\ &\qquad \texttt{else}\ (Ticker(n), \emptyset) \end{aligned}$$

In the cases of the Sink and the Ticker behaviors, no new actors are created, so the argument ν is not used in the body. To illustrate actor creation we define a

TickerFactory behavior, that takes a message $\texttt{new}(c)$, requesting creation of a new ticker, where the name of the new ticker is to be sent to actor c.

$$\begin{aligned} \mathit{TickerFactory} = \lambda(a, M, \nu) \\ \quad \texttt{if}(M = \texttt{new}(c)) \\ \quad\quad \texttt{then}\ \texttt{let}\{t := \nu(0)\} \\ \quad\quad\quad\quad (\mathit{TickerFactory}, t : \mathit{Ticker}(0), c \triangleleft \texttt{reply}(t)) \\ \quad\quad \texttt{else}\ (\mathit{TickerFactory}, \emptyset) \end{aligned}$$

We can further elaborate on the Ticker example, by providing computation and interaction paths corresponding to the Ticker event diagram of Figure 1.

- The following is a computation of an interfaced configuration consisting of a ticker actor $(t : \mathit{Ticker}(0))$ and a message $t \triangleleft \texttt{tick}$. Input/output transitions add/remove messages from the internal configuration. A delivery transition $d(mp)$ applies the behavior of the target actor to the message packet, and a name generating function, to obtain the new behavior of that actor and any additions to the configuration. The resulting expression is evaluated by sequential steps (not shown), interleaved with other transitions, thus a non-terminating behavior does not bring the whole system to a halt.

$$\begin{aligned} & (t, \emptyset)(t : \mathit{Ticker}(0),\ t \triangleleft \texttt{tick}) \\ & \quad \overset{\texttt{in}(t \triangleleft \texttt{time} @ c)}{\Longrightarrow} (t, c)(t : \mathit{Ticker}(0),\ t \triangleleft \texttt{tick},\ t \triangleleft \texttt{time} \,@\, c) \\ & \quad \overset{d(t \triangleleft \texttt{tick})}{\Longrightarrow} (t, c)(t : \mathit{Ticker}(1),\ t \triangleleft \texttt{tick},\ t \triangleleft \texttt{time} \,@\, c) \\ & \quad \overset{d(t \triangleleft \texttt{time} @ c)}{\Longrightarrow} (t, c)(t : \mathit{Ticker}(1),\ t \triangleleft \texttt{tick},\ c \triangleleft \texttt{reply}(1)) \\ & \quad \overset{d(t \triangleleft \texttt{tick})}{\Longrightarrow} (t, c)(t : \mathit{Ticker}(2),\ t \triangleleft \texttt{tick},\ c \triangleleft \texttt{reply}(1)) \\ & \quad \overset{\texttt{in}(t \triangleleft \texttt{time} @ c)}{\Longrightarrow} (t, c)(t : \mathit{Ticker}(2),\ t \triangleleft \texttt{tick},\ t \triangleleft \texttt{time} \,@\, c,\ c \triangleleft \texttt{reply}(1)) \\ & \quad \overset{\texttt{out}(c \triangleleft \texttt{reply}(1))}{\Longrightarrow} (t, c)(t : \mathit{Ticker}(2),\ t \triangleleft \texttt{tick},\ t \triangleleft \texttt{time} \,@\, c) \\ & \quad \ldots \end{aligned}$$

- The corresponding interaction path is

$$(t, \emptyset)((0, \texttt{in}(t \triangleleft \texttt{time} \,@\, c)), (4, \texttt{out}(c \triangleleft \texttt{reply}(1))), , \ldots)$$

  where an interaction sequence, ϑ is represented as sets of *time stamped* interactions (n, io) such that $\vartheta(n) = io$, omitting silent interactions ($io = \tau$).

3.2 Specification

We now have enough of the formalization to illustrate mathematical specifications and discuss the structure of models of the theory. We first make precise our notion

of satisfaction. As discussed above, an actor system specification is a predicate on interaction paths. An actor system satisfies a specification just if the predicate holds for each of its interaction paths.

Definition of Satisfaction.

For Φ a predicate on $\mathbf{IP}$ and $iC \in \mathbf{IConf}$

$$\begin{aligned} iC \models \Phi &\Leftrightarrow (\forall p \in \mathbf{CP}(iC))\Phi(cp2ip(p)) \\ &\Leftrightarrow cp2ip(\mathbf{CP}(iC)) \subseteq \{ip \in \mathbf{IP} \mid \Phi(ip)\} \end{aligned}$$

To illustrate how properties might be formalized in this theory we develop notation for expressing properties using event diagram notions, and show how this can be used to specify Ticker interactions.

Event Diagram Notation.

As above, we represent events as *time stamped* interactions (n, io). The input events ($InE(ip)$) and output events ($OutE(ip)$) of an interaction path $ip = (\rho, \chi)\vartheta$, are then defined by

$$\begin{aligned} InE(ip) &= \{(n, \mathtt{in}(mp)) \mid n \in \mathbf{Nat} \wedge mp \in \mathbf{MP} \wedge \vartheta(n) = \mathtt{in}(mp)\} \\ OutE(ip) &= \{(n, \mathtt{out}(mp)) \mid n \in \mathbf{Nat} \wedge mp \in \mathbf{MP} \wedge \vartheta(n) = \mathtt{out}(mp)\} \end{aligned}$$

An event diagram is given by two ordering relations: the arrival order—that determines for each actor, the order in which messages are received; and the activation order— the causal relation between message sending (as a result of a receive) and receipt by the target. For input events D (for delivered) and actor name a an arrival order, $\xrightarrow{ao} \in Arro(D, a)$, is a total order on the events in D with target a. The arrival order is a postulated order in which the messages are delivered to a during the computation, and may be different from the order in which they are input to the system. An activation order for D and output events O, $<_{\mathrm{ao}} \in Acto(D)$, is a binary relation between events in D and events in O. This models the causal relation between sending and delivery as a relation between the input of the delivered message resulting in the send and the output of message which must happen before it can be delivered. A theorem of Clinger [6] states that for any event diagram there is at least one total ordering of the events compatible with the combined partial order (the transitive closure of the arrival and activation orders). We call this a global time. The formula

$$\mathbf{GT}(ip, \xrightarrow{ao}, <_{\mathrm{ao}})$$

expresses the property that an interaction path ip (specifically, its interaction sequence) is a global time for the event diagram with arrival order $\xrightarrow{ao}$ and activation order $<_{\mathrm{ao}}$.

We now use these ordering and global time notations to formalize the properties of the ticker discussed in Section 2.1.

$TickerMS(a)(ip) \Leftrightarrow$

1. $ip \in \mathbf{IP}(a, \emptyset) \wedge$
2. letting $D = \{e \in InE(ip) \mid msg(e) \in \texttt{time} \mathbin{@} Cust(ip)\}$
3. $(\exists \xrightarrow{ao} \in Arro(D, a) \wedge <_{\mathrm{ao}} \in Bij(D)(OutE(ip)))$

s.t.

4. $\mathbf{GT}(ip, \xrightarrow{ao}, <_{\mathrm{ao}}) \wedge$
   $(\forall e \in D, o \in OutE(ip))(e <_{\mathrm{ao}} o \Rightarrow$
5. $\quad (\exists n \in Nat)(mp(o) = msgCust(e) \triangleleft \texttt{reply}(n)) \wedge$
6. $\quad\quad (\forall e' \in D, o' \in OutE(ip))(e' <_{\mathrm{ao}} o' \wedge e \xrightarrow{ao} e'$
   $\quad\quad\quad \Rightarrow msgArgs(o) \leq msgArgs(o'))) \wedge$
7. $|D| = \omega \Rightarrow (\forall n \in Nat)(\exists o \in OutE(ip)) msgArgs(o) \geq n$

Line 1 says that the interaction path has the right interface. Line 2 defines the set of input events D. The function msg extracts the message content of an event: $msg((n, \texttt{in}(a \triangleleft M))) = M$, and the function $Cust(ip)$ is the set of names actors that are not receptionists at any stage of ip. In the case of the Ticker, this will be any name other than a. Line 3 postulates the arrival and activation orderings of an underlying event diagram, requiring that $<_{\mathrm{ao}}$ be a bijection between D and $OutE(ip)$. The restriction of D to requests from external customers, $Cust(ip)$, is necessary, because replies to requests with customer a will not appear as output, since they are not addressed to an external actor, and thus violate the bijection requirement. Line 4 requires that ip be a global time for the postulated event diagram. Line 5 expresses the requirement that the reply contain a number and be addressed to the request customer, and line 6 says that the sequence of numbers in replies to time requests is non-decreasing. (The function $msgCust$ extracts the customer, the name following @: $msgCust((n, \texttt{in}(a \triangleleft M \mathbin{@} c))) = c$, and $msgArgs$ extracts the value from a reply message: $msgArgs((n, \texttt{out}(c \triangleleft \texttt{reply}(n)))) = n$.) Line 7 says that if there are infinitely many requests then there will be a reply with time bigger than any given number.

Claim:

$(t, \emptyset)\, Ticker(0) \models TickerMS(t)$

Proof Sketch:

We must show that if $ip \in [\![(t, \emptyset)\, Ticker(0)]\!]$ then $TickerMS(t)(ip)$. (Recall that $[\![\_]\!]$ is the denotation function mapping a configuration to its set of interaction paths.) By (the omitted) definition, ip is the interaction path derived from some Ticker computation π, a sequence of delivery, input, and output transitions where the result of a delivery transition is given by the Ticker behavior lambda. Thus we may take D to be the `time` request inputs, with customer other than t, $\xrightarrow{ao}$ to be the order of their delivery in π, and $<_{\mathrm{ao}}$ to be such that $(n, \texttt{in}(t \triangleleft \texttt{time} \mathbin{@}$

$c)) <_{\text{ao}} (n', \texttt{out}(mp))$ where mp is generated by the transition delivering the input message. Clearly $\mathbf{GT}(ip, \xrightarrow{ao}, <_{\text{ao}})$ and by definition of Ticker behavior and fairness, $<_{\text{ao}}$ is a bijection. Furthermore, if $e \xrightarrow{ao} e'$, then the counter held by the ticker at e' is at least as large as that at e (transitions do not decrement the counter). There is always a `tick` message present, and no matter how many requests arrive, fairness insures that each `tick` eventually gets delivered. Thus with an unbounded number of requests, there will always eventually be a reply bigger than any given number.

One of the motivations for defining the Ticker actor originally was as a simple example of unbounded non-determinism. Not only is the environment unpredictable, but also the system can have an unbounded number of possible behaviors. The unbounded non-determinism of a Ticker can be formalized as follows.

$$(\forall n, k \in \mathbf{Nat})(\exists (t, \emptyset)\vartheta \in \mathbf{IP}, j, n' \in \mathbf{Nat})$$
$$j \leq k \wedge n \leq n' \wedge \mathit{TickerMSpec}(t)((t, \emptyset)\vartheta) \wedge \mathit{msg}(\vartheta(j)) = \texttt{reply}(n')$$

This says that no matter how few requests there are, or how big n is, it is always possible that the reply to a request is larger than n. It is fairly easy to this see that property is a consequence of $\mathit{TickerMSpec}$.

4 Models of the Formal Theory

Without further axioms, some models of our formal theory will correspond to *small models*, where computation/interaction paths are lambda definable sequences as for the theories of sequential computation. The intended models of interest (*large models*) include paths with input/output interaction sequences that are not defined by computable functions. The question is, how does this affect the adequacy of our formal theory. For example we might ask

> Is there an actor system iC and property Φ such that in small models $iC \models \Phi$ and in some large model $iC \not\models \Phi$ or conversely?

To answer this question (positively) we describe a configuration $(\emptyset, \{o\})C$ with one external observer actor o such that in every small (i.e. recursively enumerable) model, the property Φ is satisfied, where Φ says that every interaction path for $(\emptyset, \{o\})C$ contains at least one $\texttt{out}(o \triangleleft OK)$ interaction event. But, in some larger models Φ is not satisfied, because they include interaction paths that contain no such events.

From classic recursion theory: a set of natural numbers A is said to be *simple* if A is recursively enumerable, the complement of A is infinite, and no infinite subset of the complement of A is recursively enumerable. Emil Post constructed the first simple set [20] in order to show that not all non-recursive, recursively enumerable sets, were creative. Choose A simple. By definition, let f be a partial recursive function such that

- $f(y) = 1$ if $y \in A$

- $f(y)$ is undefined otherwise.

The configuration, C, will consist of, initially, two actors. A ticker, t, and a spawner, s, a tick message to t, and a time message from s addressed to t, $t \triangleleft \texttt{time} \mathbin{@} s$.

$$(\emptyset, \{o\})(t : \mathit{Ticker}(0),\ t \triangleleft \texttt{tick},\ t \triangleleft \texttt{time} \mathbin{@} s,\ s : \mathit{Spawner}(f, t, o))$$

- $\mathit{Spawner}(f, t, o)$ does the following ad infinitum: It sends a `time` message to t, then waits for the reply. Upon receipt of a $\texttt{reply}(n)$ from t, it spawns off a new $\mathit{Compute}(f, n, o)$ actor, and sends that actor a $\texttt{param}(n)$ message.

$$\begin{aligned}\mathit{Spawner} = \lambda(f, t, o).\lambda(a, M, \nu)\ & \\ & \texttt{if}(M = \texttt{reply}(n) \mathbin{@} t) \\ & \quad \texttt{then}\ \texttt{let}\{c := \nu(0)\} \\ & \qquad (\mathit{Spawner}(f, t, o), \\ & \qquad\ c : \mathit{Compute}(f, o), \\ & \qquad\ c \triangleleft \texttt{param}(n), \\ & \qquad\ t \triangleleft \texttt{time} \mathbin{@} s) \\ & \quad \texttt{else}\ (\mathit{Spawner}(f, t, o), \emptyset)\end{aligned}$$

- $\mathit{Compute}(f, o)$ does the following: Upon receipt of a $\texttt{param}(n)$ message it computes $f(n)$ sequentially, if that terminates, it sends o the OK message, and becomes a sink. Any other message is simply discarded.

$$\begin{aligned}\mathit{Compute} = \lambda(f, o).\lambda(a, M, \nu)\ & \\ & \texttt{if}(M = \texttt{param}(n)) \\ & \quad \texttt{then}\ \texttt{let}\{b := f(n)\} \\ & \qquad (\mathit{Sink}, o \triangleleft \mathit{OK}) \\ & \quad \texttt{else}\ (\mathit{Compute}(f, o), \emptyset)\end{aligned}$$

A computation path starting from our configuration in state

$$(\emptyset, \{o\})(t : \mathit{Ticker}(n),\ t \triangleleft \texttt{tick},\ t \triangleleft \texttt{time} \mathbin{@} s,\ s : \mathit{Spawner}(f, t, o))$$

will be an infinite number of segments, all of the form

$$\begin{aligned}&(\emptyset, \{o\})(t : \mathit{Ticker}(n),\ t \triangleleft \texttt{tick},\ t \triangleleft \texttt{time} \mathbin{@} s, \\ &\qquad s : \mathit{Spawner}(f, t, o)) \\ &\xrightarrow{d(t \triangleleft \texttt{tick})^m} \\ &(\emptyset, \{o\})(t : \mathit{Ticker}(n + m),\ t \triangleleft \texttt{tick},\ t \triangleleft \texttt{time} \mathbin{@} s, \\ &\qquad s : \mathit{Spawner}(f, t, o)) \\ &\xrightarrow{d(t \triangleleft \texttt{time} \mathbin{@} c)}\ \xrightarrow{d(t \triangleleft \texttt{tick})^{m_1}}\end{aligned}$$

$$(\emptyset, \{o\})(t : \mathit{Ticker}(n+m+m_1),\ t \triangleleft \mathtt{tick},$$
$$s : \mathit{Spawner}(f,t,o),\ s \triangleleft \mathtt{reply}(n+m)\ @\ t)$$
$$\overset{d(s \triangleleft \mathtt{reply}(n+m)@t)}{\Longrightarrow}\ \overset{d(t \triangleleft \mathtt{tick})^{m_2}}{\Longrightarrow}$$
$$(\emptyset, \{o\})(t : \mathit{Ticker}(n+m+m_1+m_2),\ t \triangleleft \mathtt{tick},\ t \triangleleft \mathtt{time}\ @\ s,$$
$$s : \mathit{Spawner}(f,t,o),$$
$$c : \mathit{Compute}(f,o),\ c \triangleleft \mathtt{param}(n+m))$$
$$\overset{d(c \triangleleft \mathtt{param}(n+m))}{\Longrightarrow}\ \overset{d(t \triangleleft \mathtt{tick})^{m_3}}{\Longrightarrow}$$
$$(\emptyset, \{o\})(t : \mathit{Ticker}(n+m+m_1+m_2+m_3),\ t \triangleleft \mathtt{tick},\ t \triangleleft \mathtt{time}\ @\ s,$$
$$s : \mathit{Spawner}(f,t,o),$$
$$c : \mathit{Compute}(f,o)(c, \mathtt{param}(n+m), \nu))$$

for various values of n, m, m_1, m_2, and m_3. Where the notation $\overset{d(t \triangleleft \mathtt{tick})^m}{\Longrightarrow}$ is an abbreviation for the delivery of m tick messages to the ticker actor t:

$$\overset{d(t \triangleleft \mathtt{tick})^m}{\Longrightarrow} \quad \text{abbreviates} \quad \underbrace{\overset{d(t \triangleleft \mathtt{tick})}{\Longrightarrow}\ \overset{d(t \triangleleft \mathtt{tick})}{\Longrightarrow}\ \ldots\ \overset{d(t \triangleleft \mathtt{tick})}{\Longrightarrow}}_{m \text{ times}}$$

Note that we are using fairness in assuming that the system does not ignore pending messages. Each message sent to an actor is eventually processed. Without such an assumption we would be forced to consider paths where the only events that ever took place are, for example, the ticker processing it's tick message, and no replies to the spawner's time messages are ever sent.

Thus from an observational view point an infinite computation will boil down to computing

$$c_i : \mathit{Compute}(f,o)(c_i, \mathtt{param}(n_i), \nu))$$

or more explicitly:

$$c_i : \mathtt{let}\{b := f(n_i)\}(\mathit{Sink}, o \triangleleft \mathit{OK})$$

for an increasing sequence $\{n_i\}_{i \in \mathbf{Nat}}$. Now in a small model, this sequence will be a recursively enumerable set of increasing integers (hence actually recursive). Hence it cannot be a subset of the complement of A. Thus it must intersect A, so for some ticker response, n_i, n_i will be in A, and hence $f(n_i)$ will terminate, and an $\mathtt{out}(o \triangleleft \mathit{OK})$ event will be generated.

On the other hand, in a large model, take a ticker path $\{n_i\}_{i \in \mathbf{Nat}}$ that enumerates an infinite subset of the complement of A (e.g. the complement of A itself will do). In such a path no OK message will ever get sent. Thus small models satisfy Φ while large models do not.

5 Conclusions and Future Directions

We have described work in progress to formalize the interaction semantics model of actor computation in a logical theory. This work has two objectives: to study the logical and recursion theoretic properties of interactive computation; and to develop principles and techniques for specifying and reasoning about interactive systems.

We have analyzed a particular formalism for interactive computation based on the actor model. There are many other models / formal systems for which a similar analysis could be carried out: Communicating Sequential Processes [14,21], Pi calculus [19], IO Automata [17], Game semantics [1].

Our example system distinguishing small and large models relied on fairness. It would be interesting to determine if fairness is essential to make the distinction or if an example can be found that doesn't require fairness. Work in the theory of computable functions has show the equivalence of many formal systems for computation (Turing machines, Lambda calculus, and so on) in the sense that they all define the same set of computable functions on the natural numbers, namely the partial recursive functions. An interesting topic for future work is to determine whether different formalisms for interactive computation are equivalent, for example in the sense of definable sets of interaction paths. Another possible direction is to use alternative logics to carry out formal analyses, for example temporal logics.

There are several additional directions for future work. One is formalizing the component algebra operations to help modularize specification and reasoning, and looking for a set of reasoning principles, including for example principles for reasoning about coordination abstractions. Another is working out a more detailed proof-theoretic analysis of interactive computation models. Finally, we intend to explore variants of the actor model that incorporate notions of time and uncertainty.

References

[1] S. Abramsky. Sequentiality vs. concurrency in games and logic. *Mathematical Structures in Computer Science*, 13:531–565, 2003.

[2] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, 1986.

[3] G. Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[4] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1997.

[5] Henry G. Baker and Carl Hewitt. Laws for communicating parallel processes. In *IFIP Congress*, pages 987–992. IFIP, August 1977.

[6] W. D. Clinger. Foundations of actor semantics. AI-TR- 633, MIT Artificial Intelligence Laboratory, May 1981.

[7] R. de Nicola and M. C. B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[8] S. Feferman. A language and axioms for explicit mathematics. In *Algebra and Logic*, volume 450 of *Springer Lecture Notes in Mathematics*, pages 87–139. Springer Verlag, 1975.

[9] S. Feferman. Constructive theories of functions and classes. In *Logic Colloquium '78*, pages 159–224. North-Holland, 1979.

[10] S. Feferman. Polymorphic typed lambda-calculi in a type-free axiomatic framework. In *Logic and Computation*, volume 106 of *Contemporary Mathematics*, pages 101–136. A.M.S., Providence R. I., 1990.

[11] I. Greif. Semantics of communicating parallel processes. Technical Report 154, MIT, Project MAC, 1975.

[12] C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–364, 1977.

[13] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of 1973 International Joint Conference on Artificial Intelligence*, pages 235–245, August 1973.

[14] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[15] F. Honsell, I. A. Mason, S. F. Smith, and C. L. Talcott. A variable typed logic of effects. *Information and Computation*, 119(1):55–90, 1995.

[16] P. J. Landin. The next 700 programming languages. *Comm. ACM*, 9:157–166, 1966.

[17] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. Technical Report MIT/LCS/TM-373, MIT Laboratory for Computer Science, 1988.

[18] I. A. Mason and C. L. Talcott. Feferman–Landin Logic. In W. Sieg, R. Sommer, and C.L. Talcott, editors, *Reflections on the Foundations of Mathematics: Essays in honor of Solomon Feferman*, Lecture Notes in Logic, pages 299–344. Association of Symbolic Logic, 2002.

[19] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[20] Emil L. Post. Recursively enumerable sets of positive integers and their decision problems. *Bulletin of the American Mathematical Society*, 50:284–316, 1944.

[21] W. Roscoe. *Theory and Practice of Concurrency*. Prentice Hall, 1998.

[22] S. F. Smith and C. L. Talcott. Modular reasoning for actor specification diagrams. In P. Ciancariani, A. Fantechi, and R. Gorrieri, editors, *Formal Methods for Open Object-based Distributed Systems*, pages 313–330. Kluwer, 1999.

[23] S. F. Smith and C. L. Talcott. Specification diagrams for actor systems. *Higer-Order and Symbolic Computation*, 15(4):301–348, 2002.

[24] C. L. Talcott. A theory for program and data specification. *Theoretical Computer Science*, 104:129–159, 1993.

[25] C. L. Talcott. Interaction semantics for components of distributed systems. In E. Najm and J-B. Stefani, editors, *1st IFIP Workshop on Formal Methods for Open Object-based Distributed Systems, FMOODS'96*, 1996. proceedings published in 1997 by Chapman & Hall.

[26] C. L. Talcott. Reasoning about functions with effects. In *Higher Order Operational Techniques in Semantics*. Cambridge University Press, 1996.

[27] C. L. Talcott. Composable semantic models for actor theories. *Higher-Order and Symbolic Computation*, 11(3):281–343, 1998.

[28] C. L. Talcott. Actor theories in rewriting logic. *Theoretical Computer Science*, 285(2), 2002.

[29] Prasanna Thati, Carolyn Talcott, and Gul Agha. Techniques for executing and reasoning about specification diagrams. In *10th International Conference on Algebraic Methodology and Software Technology AMAST2004*, 2004.