Actario: A Framework for Reasoning About Actor Systems

Shohei Yasutake
Department of Computer Science
Tokyo Institute of Technology
2-12-1 Ookayama, Meguroku, Tokyo
152-8552, Japan
yasutake@psg.cs.titech.ac.jp

Takuo Watanabe
Department of Computer Science
Tokyo Institute of Technology
2-12-1 Ookayama, Meguroku, Tokyo
152-8552, Japan
takuo@acm.org

ABSTRACT

The two main characteristics of the Actor model are asynchronous message passing and dynamic system topology. The latter relies on the on-the-fly creation of actor names that often complicates the formal treatment of systems described in the Actor model. In this paper, we introduce Actario, a formalization of the Actor model in Coq. Actario incorporates a name creation mechanism that is formally proven to generate a consistent set of actor names. The mechanism helps proper handling of names in modeling and reasoning about actor-based systems. Actario also provides a code extraction mechanism that generates Erlang programs.

Categories and Subject Descriptors

F.3.2 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Mechanical verification*

General Terms

Actors, Formal Models

Keywords

Actor Model, Formalization, Actario, Coq, Erlang

1. INTRODUCTION

The Actor model[3] is a kind of concurrent computation model, in which a system is expressed as a collection of autonomous computing entities called actors that communicate each other only with asynchronous messages. On receiving a message, an actor may (1) send messages to other actors (or itself) whose names are known to the sender, (2) create new actors and (3) change its behavior for the next message.

Starting from the 1970s, the Actor model and its variations such as concurrent objects[15] have a long research history. They are today regarded as popular high-level abstractions for concurrent and parallel programming used in some industrial strength language and libraries such as Erlang[7], Scala[10] and Akka[4]. Because of this situation, establishing a mechanized formal verification method for actor-based systems is a pressing issue.

Several methods and systems for formally verifying actor-based systems have been presented recently. Rebeca[11] is a modeling language that allows model-checking. For deductive verification using proof assistants, formalizations using Athena[9] and Coq[8] have been presented.

A *name*[1] in the Actor model is a unique conceptual location associated with each actor. The concept of *name uniqueness* denotes that each name uniquely refers an actor and each actor should be referred by a single name. In the implementations of actor systems including Erlang, Scala, and Akka, naming of actors is implicit; we don't need to manually assign a fresh name to a newly created actor. The name uniqueness may be broken if the naming is explicit in complex systems. Implicit naming, however, might complicates the formal treatment of actor-based systems. Thus, some formalization adopts explicit naming.

In this paper, we propose Actario[1], a Coq framework for implementing and verifying actor-based systems. The framework (1) supports Erlang-like notation for describing an actor system, (2) allows verifying desired properties of the system using the proof mechanism in Coq, and (3) generates executable Erlang code from the system description.

To be close to realistic actor languages and libraries, we designed Actario to support implicit naming. The naming mechanism behind the scene is formally proven to satisfy the name uniqueness. We also proved other properties including the persistence of actors and messages. The proof scripts of these properties are available in the GitHub repository of Actario[1].

The layout of the rest of this paper is as follows. The next section describes the overview of Actario. In Section 3, we give the operational semantics of the Actor model formalized in Actario. Section 4 outlines the proof of the uniqueness property on dynamically generated names. In Section 5, we discuss fairness properties formalized in Actario. The code extraction mechanism is described in Section 6. Finally, Section 7 overviews related work and Section 8 concludes the paper.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
AGERE!@SPLASH Oct., 2015, Pittsburgh, PA, USA
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.

[1]The term *mail address* is used in some other literature.

2. OVERVIEW OF ACTARIO

2.1 Programming in Actario

Actario is a Coq framework for defining and verifying actor-based systems. A typical workflow using Actario is as follows.

1. Describe an actor system using types and notations defined in the framework.
2. Specify and verify desired properties of the system.
3. Extract the Erlang version of the system using the code extraction mechanism of Coq.

Note that Actario does not provide a dedicated language for describing actor systems. The framework offers a set of Coq vocabularies (types and notations described in Section 2.2) for that purpose.

Example: Recursive Factorial System.

We use a simple example to illustrate a system description in Actario. Figure 1 shows the definition of an actor system that implements the continuation-passing style factorial function adapted from [3]. In this definition, the function `factorial_system` sets up a system that initially consists of a single factorial actor whose behavior is defined as `factorial_behv`. The actor can receive a tuple of a natural number and the name of a *customer* actor (`cust`) that is intended to receive the result. If the first component of tuple is more than zero, *i.e.*, it matches the successor pattern `S n`, the actor creates a new continuation actor (`cont`) and recursively sends itself a pair of `n` and `cont`. The behavior of continuation actors is specified as `factorial_cont_behv`.

```
Definition factorial_cont_behv (val : nat)
                               (cust : name) :=
  receive (fun msg ⇒
    match msg with
     | nat_msg arg ⇒ cust ! nat_msg (val * arg);
                     become empty_behv
     | _ ⇒ become empty_behv
    end).

CoFixpoint factorial_behv :=
  receive (fun msg ⇒
    match msg with
     | tuple_msg (nat_msg 0) (name_msg cust) ⇒
       cust ! nat_msg 1;
       become factorial_behv
     | tuple_msg (nat_msg (S n)) (name_msg cust) ⇒
       cont ← new (factorial_cont_behv (S n) cust);
       me ← self;
       me ! tuple_msg (nat_msg n) (name_msg cont);
       become factorial_behv
     | _ ⇒ become factorial_behv
    end).

Definition factorial_system (n : nat) (cust : name) :=
  init "factorial" (
    x ← new factorial_behv;
    x ! tuple_msg (nat_msg n) (name_msg cust);
    become empty_behv
  ).
```

Figure 1: Recursive Factorial System in Actario

2.2 Types and Notations

2.2.1 Types

Figure 2 shows the inductively defined type of messages delivered among actors, each of whose constructors corresponds to a kind of messages. In the current version of Actario, a message may be empty, an actor name, a value of basic types (Boolean, natural number or string), or a tuple of two messages.

Figure 3 defines two mutually coinductive types: `actions` and `behavior`. They specify sequences of actions performed by actors and behaviors of actors respectively. Each constructor of `actions` corresponds to a single action embedded in an action sequence as follows.

`new` b f creates a new actor with initial behavior b and applies f to the name of the created actor. Then continues to the action sequence that f returns.

`send` n m α sends message m to the actor with name n and then continues to action sequence α.

`self` f retrieves the name of the actor that executes this action and applies f to it. Then continues to the action sequence that f returns.

`become` b sets b as the next behavior of the actor that executes this action. This action should end an action sequence.

```
Inductive message : Set :=
 | empty_msg : message
 | name_msg : name → message
 | str_msg : string → message
 | nat_msg : nat → message
 | bool_msg : bool → message
 | tuple_msg : message → message → message.
```

Figure 2: Message Type

```
CoInductive actions : Type :=
 | new : behavior → (name → actions) → actions
 | send : name → message → actions → actions
 | self : (name → actions) → actions
 | become : behavior → actions
with behavior : Type :=
 | receive : (message → actions) → behavior.
```

Figure 3: Types for Actions and Behaviors

```
Notation ``n '← ' 'new' behv ; cont'' :=
    (new behv (fun n ⇒ cont))
    (at level 0, cont at level 10).
Notation ``n '!' m ';' a'' :=
    (send n m a) (at level 0, a at level 10).
Notation ``me '← ' 'self' ';' cont'' :=
    (self (fun me ⇒ cont))
    (at level 0, cont at level 10).
```

Figure 4: Notations for Actions

```
new b (fun x ⇒
self (fun s ⇒
send x (name_msg s)
(become b')))
```

(a) without notations

```
x ← new b;
s ← self;
x ! (name_msg s);
become b'
```

(b) with notations

Figure 5: Example Use of Notations

In the Actor model, an actor persists indefinitely. Thus, as shown in Figure 1, that a behavior may have `become` actions that specify itself or other behaviors eventually recurring to the original one. The reason for using `CoFixpoint` and defining `actions` and `behavior` coinductively is to model such behaviors.

2.2.2 *Notations*

In addition to the types defined above, Actario provides a collection of notations (syntactic sugaring) described in Figure 4. Using the notations, we can write actor behaviors intuitively without being complicated by CPS. Figure 5 compares the descriptions of a simple action sequence without/with the notations.

3. SEMANTICS

In this section, we explain the formalization of the operational semantics of the Actor model in Actario. First, for the explanation of formalization of operational semantics, we describe `name` type, `actor` type, `in_flight_message` type, and `config` type. And then, we explain how to formalize the operational semantics in Actario.

3.1 Actor Name

In Actario, actor name is defined as the disjoint sum of the case of an actor with no parent and the case of an actor generated by another actor (Figure 6). We call the actors having no parent *top level actor*. Top level actor represents initial actors in the system. And we call the actors generated by another actor *generated actor*. The name of a generated actor consists of the name of parent actor and the number that the parent actor generated so far. We call the number *generation number*. To keep name uniqueness, we introduce generation number. For more detail about name uniqueness, see Section 4.

3.2 actor

We explain how `actor` is defined in Actario. Actor consists of its name, sequence of remaining actions, and next generation number to use in generating next child (Figure 7). If remaining actions are only `become`, the actor is ready for receiving a message.

```
Inductive name : Set :=
| toplevel : string → name
| generated : nat → name → name.
```

Figure 6: name

```
Record actor := {
  actor_name : name;
  remaining_actions : actions;
  next_num : gen_number
}.
```

Figure 7: actor

3.3 in flight message

Next, we define `in_flight_message` type which represents messages in flight in the configuration. `in_flight_message` consists of the name of the destination, the name of the sender, and the content of the message (Figure 8).

3.4 configuration

configuration represents a snapshot of the actor system. configuration is used to formulate operational semantics of the Actor model. In Actario, a configuration consists of a list of actors and a list of messages in flight.

3.5 label

Actario formulates operational semantics of the Actor model as labeled transition system, so we define label (Figure 10). The explanations of each label are as follows.

`Receive (to : name) (from : name) (content : message)`
This represents that the actor named `to` receives the message `content` sent from the actor named `from`.

`Send (from : name) (to : name) (content : message)`
This represents that the actor named `from` sends the message `content` to the actor named `to`.

`New (child : name)`
This represents that the actor named `child` is generated.

`Self (me : name)`
This represents that the actor named `me` gets the name itself.

3.6 semantics

We formulate operational semantics of the Actor model as labeled transition system. For the later explanation, we define the symbols as shown in Figure 11.

The labeled transition system used in Actario is defined like Figure 12. The explanations for each of transitions are the followings.

```
Record in_flight_message := {
  to : name;
  from : name;
  content : message
}.
```

Figure 8: in flight message

```
Record config := {
  in_flight_messages : list in_flight_message;
  actors : list actor
}.
```

Figure 9: config

Receive
RECEIVE is the transition for Receive label. The actor which is ready to receive a message, in other words, the actor whose remaining actions are only become, receives a message and generate new remaining actions decided by the behavior and the content of the message.

Send
SEND is the transition for Send label. The actor which want to send a message sends a message, and then the message is added into messages in flight.

New
NEW is the transition for New label. An actor generates its child actor by the given behavior. And then, do the followings:

- The child actor is added into the configuration. The next generation number of child actor is 0.
- The next generation number of the parent actor increases by 1.
- The child actor is ready to receive a message.

Self
SELF is the transition for Self label. An actor gets the self name and applies it to the continuation.

The definition in Actario is in Appendix A.

4. NAME UNIQUENESS

In programming languages or libraries providing the Actor model such as Erlang or Akka, the system automatically generates actors with fresh names without specifying the name explicitly by the programmer. In Actario, the proposition that all actor names in the configuration are not duplicate by any transitions is proven.

To prove, we define an invariant about actor names preserved between any transitions. It is named *trans invariant*. The trans invariant consists of the following three predicates for configuration.

$$\texttt{trans\_invariant}(c) := \\ \texttt{chain}(c) \land \texttt{gen\_fresh}(c) \land \texttt{no\_dup}(c)$$

```
Inductive label :=
| Receive (to : name) (from : name) (content :
    message)
| Send (from : name) (to : name) (content :
    message)
| New (child : name)
| Self (me : name).
```

Figure 10: label

$$\begin{array}{rcll}
c & \in & \textit{Configuration} & = & \textit{Set(InFlight)} \times \textit{Set(Actor)} \\
a & \in & \textit{Actor} & = & \textit{Name} \times \textit{Actions} \times \mathbb{N} \\
n & \in & \textit{Name} & ::= & \mathsf{toplevel}(s) \mid \mathsf{generated}(g, n) \\
m & \in & \textit{Message} & = & \textit{Name} + \textit{PrimVal} + \\
 & & & & \textit{Message} \times \cdots \times \textit{Message} \\
i & \in & \textit{InFlight} & = & \textit{Name} \times \textit{Name} \times \textit{Message} \\
b & \in & \textit{Behavior} & = & \textit{Message} \to \textit{Actions} \\
\alpha & \in & \textit{Actions} & ::= & \mathsf{send}(n, m, \alpha) \\
 & & & \mid & \mathsf{new}(b, \kappa) \\
 & & & \mid & \mathsf{self}(\kappa) \\
 & & & \mid & \mathsf{become}(b) \\
l & \in & \textit{Label} & ::= & \mathsf{Receive}(n, n, m) \\
 & & & \mid & \mathsf{Send}(n, n, m) \\
 & & & \mid & \mathsf{New}(n) \\
 & & & \mid & \mathsf{Self}(n) \\
\kappa & \in & \textit{Name} \to \textit{Actions} & & \\
g & \in & \mathbb{N} & &
\end{array}$$

Figure 11: Configuration

The brief explanations of `chain`, `gen_fresh`, and `no_dup` are followings:

`chain`
For each actor in the configuration, if the actor is generated by another actor, then the parent actor is also in the configuration.

`gen_fresh`
For each actor in the configuration, actor name genereted by the actor in the next is fresh.

`no_dup`
For all actor name in the configuration are unique.

4.1 functions

Before starting the explanation and the proof, we define some functions used in this section.

`actors` : ***Configuration*** $\to$ ***Set(Actor)***
`actors` returns the set of actors in the given configuration.

`parent` : ***Actor*** $\to$ ***Actor***
`parent` returns the parent actor of the given actor. If the given actor is toplevel actor, the function returns nothing.

`gen_number` : ***Actor*** $\to \mathbb{N}$
`gen_number` returns generated number of the name of the given actor. If the given actor is toplevel actor, the function returns nothing.

$$(I \uplus \{(n_{\mathrm{to}}, n_{\mathrm{from}}, m)\}, A \cup \{(n_{\mathrm{to}}, \mathsf{become}(b), g)\}) \overset{\mathsf{Receive}(n_{\mathrm{to}}, n_{\mathrm{from}}, m)}{\rightsquigarrow} (I, A \cup \{(n_{\mathrm{to}}, b(m), g)\}) \quad (\textsc{Receive})$$

$$(I, A \cup \{(n_{\mathrm{from}}, \mathsf{send}(n_{\mathrm{to}}, m, \alpha), g)\}) \overset{\mathsf{Send}(n_{\mathrm{from}}, n_{\mathrm{to}}, m)}{\rightsquigarrow} (I \uplus \{(n_{\mathrm{to}}, n_{\mathrm{from}}, m)\}, A \cup \{(n_{\mathrm{from}}, \alpha, g)\}) \quad (\textsc{Send})$$

$$(I, A \cup \{(n, \mathsf{new}(b, \kappa), g)\}) \overset{\mathsf{New}(n')}{\rightsquigarrow} (I, A \cup \{(n, \kappa(n'), g+1), (n', \mathsf{become}(b), 0)\}) \text{ where } n' := \mathsf{generated}(g, n) \quad (\textsc{New})$$

$$(I, A \cup \{(n, \mathsf{self}(\kappa), g)\}) \overset{\mathsf{Self}(n)}{\rightsquigarrow} (I, A \cup \{n, \kappa(n), g\}) \quad (\textsc{Self})$$

Figure 12: labeled transition semantics

`next_number` : ***Actor*** $\to \mathbb{N}$
`next_number` returns next generation number of the given actor.

`name` : ***Actor*** $\to$ ***Name***
`name` returns the name of the given actor.

`names` : ***Set(Actor)*** $\to$ ***Set(Name)***
`names` returns names of the given set of actors.

4.2 chain

We define an predicate of configuration, called `chain`. `chain` is the predicate that, for each actor in the given configuration, if it is generated by another actor, the parent actor is also in the configuration. `chain` is defined as the following.

$$\texttt{chain}(c) := \forall a \in \texttt{actors}(c), \exists p, p = \texttt{parent}(a) \Rightarrow p \in \texttt{actors}(c)$$

Then, we can prove *chain preservation property* that chain is preserved between any transitions. The proof is by case analysis on the label. `chain` is decided by only actor names, and the transition which have a possibility to change the names in the configuration is only New transition. Therefore, we consider only the case of New transition.

Lemma 1. *chain preservation*

$$\forall c, c' \in \mathit{Configuration}, \forall l \in \mathit{Label}, \mathit{chain}(c) \wedge c \overset{l}{\rightsquigarrow} c' \Rightarrow \mathit{chain}(c')$$

4.3 gen_fresh

We define `gen_fresh` predicate that, for each actor in the configuration, the name of its child is always fresh. The definition of `gen_fresh` is complicated a little. We translate the proposition that next generated name is fresh to the following.

$$\texttt{gen\_fresh}(c) := \forall a \in \texttt{actors}(c), \exists p, p = \texttt{parent}(a) \wedge p \in \texttt{actors}(c) \Rightarrow \texttt{gen\_number}(a) < \texttt{next\_number}(p)$$

It is guaranteed that the actor name generated in the next is fresh if satisfying `gen_fresh` predicate by the relation of next generation numbers and actor names. However, the actor name generated after the next is not always fresh name. For example, if there are two actors (A and B) that have the same name and the same next generation number and actor A generates a child actor and actor B generates a child actor, although `gen_fresh` holds, these child actors have the same name. Furthermore, if the parent of the actor A does not exist in the configuration and the parent of the parent exists in the configuration, and the parent of the parent actor generates an actor and it also generates an actor, then the name is possible to have the same as A's one.

Thus, to prove *gen fresh preservation* proposition that `gen_fresh` is preserved between transitions, it is necessary to use `chain` and `no_dup` as hypotheses.

Lemma 2. *gen fresh preservation*

$$\forall c, c' \in \mathit{Configuration}, \forall l \in \mathit{Label}, \mathit{chain}(c) \wedge \mathit{gen\_fresh}(c) \wedge \mathit{no\_dup}(c) \wedge c \overset{l}{\rightsquigarrow} c' \Rightarrow \mathit{gen\_fresh}(c')$$

4.4 no_dup

We define `no_dup` predicate that all actor names in the given configuration are unique. This is the property we have to prove. `no_dup` is defined as the following.

$$\texttt{no\_dup}(c) := \forall a \in \texttt{actors}(c), \texttt{name}(a) \notin \texttt{names}(\texttt{actors}(c) \setminus \{a\})$$

We proved *no dup preservation* property defined as the following. It represents that if the actor names in the configuration is not duplicate and the next generated actor name is fresh, then `no_dup` holds in the next configuration.

Lemma 3. *no dup preservation*

$$\forall c, c' \in \mathit{Configuration}, \forall l \in \mathit{Label}, \mathit{gen\_fresh}(c) \wedge \mathit{no\_dup}(c) \wedge c \overset{l}{\rightsquigarrow} c' \Rightarrow \mathit{no\_dup}(c')$$

4.5 uniqueness

Then, we start to prove name uniqueness. First, we prove trans invariant preservation that trans invariant is preserved between transitions. This is obvious by chain preservation, gen fresh preservation and no dup preservation.

Lemma 4. *trans invariant preservation*

$$\forall c, c' \in \mathit{Configuration}, \forall l \in \mathit{Label}, \mathit{trans\_invariant}(c) \wedge c \overset{l}{\rightsquigarrow} c' \Rightarrow \mathit{trans\_invariant}(c')$$

Next, we prove that if trans invariant holds in initial configuration, trans invariant holds after arbitrary transitions.

Lemma 5. *trans invariant preservation star*

$$\forall c, c' \in \mathit{Configuration}, \forall l \in \mathit{Label}, \mathit{trans\_invariant}(c) \wedge c \overset{l}{\rightsquigarrow} \star\, c' \Rightarrow \mathit{trans\_invariant}(c')$$

$c \overset{l}{\rightsquigarrow}\star\, c'$ represents reflexive transitive closure of transition. The proof is by induction of reflexive transitive closure of transition and trans invariant preservation.

Finally, we can prove name uniqueness.

THEOREM 1. *name uniqueness*

$$\forall c, c' \in \mathit{Configuration}, \forall l \in \mathit{Label},$$
$$\mathit{trans\_invariant}(c) \wedge c \overset{l}{\rightsquigarrow}\star\, c' \Rightarrow \mathit{no\_dup}(c')$$

This is obvious by trans invariant preservation star because `no_dup` is in `trans_invariant`.

5. FAIRNESS

fairness is a property that reception of a message does not delay infinitely. There are two variants of fairness property, weak fairness and strong fairness. Weak fairness is that if an actor is infinitely always ready to receive the message, the message is eventually received. Strong fairness is that if an actor is infinitely often ready to receive the message, the message is eventually received. The Actor model satisfies strong fairness. In this section, we define strong fairness in Actario.

5.1 Transition Path

Generally, fairness is represented in using operators of temporal logic. We have to encode temporal logic because Coq does not support temporal logic. We use transition path, which represents transition sequence of configuration, to define fairness as a predicate of transition path. This method is used in Applπ [2].

We define transition path as a function of $\mathbb{N}$ to `option config`. In this definition, $\mathbb{N}$ represents the number of transitions from initial configuration and the reason why the return value is wrapped with `option` is that it may be no more transitions.

```
Definition path := nat → option config.
```

And we define the predicate that the given path is correct transition path.

```
Definition is_transition_path (p : path) : Prop :=
  ∀ n,
    (∀ c, p n = Some c →
              (∃ c' l, p (S n) = Some c' /\ c ~(l
    )⇝ c') \/
              p (S n) = None) /\
    (p n = None → p (S n) = None).
```

5.2 enabled

We define the predicate that the transition from the given configuration with the given label is possible, called `enabled`. In Actario, `enabled` is defined as there exists a configuration after transitioning from the configuration with the label, as follows.

```
Definition enabled (c : config) (l : label) : Prop
    :=
  ∃ c', c ~(l)⇝ c'.
```

5.3 infinitely often enabled

We define the predicate that the transition is infinitely often enabled in the transition path. It is named `infinitely often enabled`.

```
Definition infinitely_often_enabled (l : label) (p
    : path) : Prop :=
  ∀ n c, p n = Some c →
            enabled c l →
            ∃ m c', m > n /\
                        p m = Some c' /\
                        enabled c' l.
```

5.4 eventually processed

We define `eventually processed` that is the predicate of label and transition path. It represents that the transition with the label is processed eventually in the path. It is defined as follows.

```
Definition eventually_processed (l : label) (p :
    path) : Prop :=
  ∃ n c c',
    p n = Some c /\ p (S n) = Some c' /\ c ~(l)⇝
    c'.
```

5.5 Definition of fairness

Then we can define `fairness` predicate for transition path. For the given transition path and for each label, if `infinitely often enabled` holds, then `eventually processed` holds. `is postfix of` predicate is used for representing 'infinite'. If `is postfix of` is not used, the transition may not be processed after the transition is processed although the transition is processed in whole the path. To prevent it, if `inifinitely often enabled` holds then `eventually processed` holds for arbitrary postfix path by using `is postfix path`.

```
Definition is_postfix_of (p' p : path) : Prop :=
  ∃ n, (∀ m, p' m = p (m + n)).

Definition fairness : Prop :=
  ∀ p p', is_postfix_of p' p →
             (∀ l,
                infinitely_often_enabled l p' →
                eventually_processed l p').
```

6. EXTRACTION

Extraction is a Coq feature which enables to convert Coq programs to the programs of other languages. Normal Coq can extract programs to OCaml, Haskell, and Scheme. If we want to extract to other languages or use custom extraction algorithm, we have to implement it as plugins or patches. Actario has custom extraction mechanism for the programs using Actario. It can extract to Erlang. It is not proven that the extraction mechanism does not change the meanings of Actario programs and Erlang programs. In Actario, `ActorExtraction` command is defined for extracting actor systems. It is used like traditional `Extraction` command.

```
(* Inductive nat :=  *)
(* | O : nat         *)
(* | S : nat → nat. *)

O (* ⇒ {o} *)
S (S (S O)) (* ⇒ {s, {s, {s, o}}} *)
```

Figure 13: example of extraction of algebraic data types

```
CoFixpoint behvA :=
  receive (fun msg ⇒
    match msg with
      | name_msg sender ⇒
        me ← self;
        sender ! name_msg me;
        become behvA
      | _ ⇒
        child ← new behvB;
        child ! msg;
        become behvA
    end)
```

Figure 14: Extraction example: Actario code

6.1 Data Types

Values of algebraic data types are extracted to a tuple with the label. Value constructor is extracted to a label, and arguments are extracted to the second and the following elements of the tuple. Figure 13 is an example of extraction of the natural number type.

However, actions of actors, for example, send, new, self, become and behavior are implemented as value constructor of actions and behavior type We handle these constructors as special to generate the corresponding syntax of Erlang.

For example, Actario code shown in Figure 14 is extracted to Erlang code shown in Figure 15.

6.2 Name

In Actario, a programmer does not make actor names from constructors, so that all of actor names are in variables. Therefore, all of actor names in extracted code are variables. These variables are bound by values of name type in Actario, but in Erlang, these variables are bound by process ids.

6.3 Execution

The program extracted by Actario is impossible to execute by itself. So Actario's programmers have to write executor to execute the extracted Actor system in Erlang. For example, we consider factorial system described in Section 2.

```
Definition factorial_system (n : nat) (parent :
    name) : config :=
  init "factorial" (
        x ← new factorial_behv;
        x ! tuple_msg (nat_msg n) (name_msg
    parent);
```

```
behvA() →
  receive Msg → case Msg of
    {name_msg, Sender} →
      Me = self(),
      Sender ! {name_msg, Me},
      behvA()
    _ →
      Child = spawn(fun() → behvB() end),
      Child ! Msg
      behvA()
  end.
```

Figure 15: Extraction example: Erlang code

```
        become empty_behv
      ).
```

factorial_system is extracted to the following Erlang code.

```
factorial_system(N, Parent) →
    X = spawn(fun() →
        factorial_behv()
    end),
    X ! {tuple_msg, {nat_msg, N}, {name_msg,
    Parent}},
    empty_behv().
```

To execute this, we have to write executor like the following. nat2int and int2nat are auxiliary functions for converting Coq's natural number and Erlang's integer.

```
-module(fact_main).
-export([fact/1]).

fact(N) →
    _ = spawn(factorial, factorial_system, [
    int2nat(N), self()]),
    receive
        {nat_msg, Result} →
            io:fwrite("fact(~w) = ~w~n", [N,
    nat2int(Result)]);
        _ →
            io:fwrite("error~n")
    end.

nat2int({o}) → 0;
nat2int({s, N}) → nat2int(N) + 1.

int2nat(0) → {o};
int2nat(N) when N > 0 → {s, int2nat(N - 1)};
int2nat(_) → {o}.
```

7. RELATED WORK

Applπ is a Coq library for modeling and verifying concurrent programs [2]. Actario is very inspired by Applπ, for example, the definition of fairness, continuation passing style in actions and framework design. The main difference

of Applπ and Actario is that Applπ adopts π-calculus for its concurrent computation basic, but Actario adopts the Actor model for its concurrent computation basic.

Musser and Varela are formalized the Actor model in the Athena theorem prover [5][9]. In this paper, name uniqueness is proven. However, a programmer has to name new actors explicitly. Therefore, a programmer has to select a fresh name. It is difficult to give always fresh name in complex system. In addition, it is impossible to run the program built in the formalization, while Actario can by extraction.

Verdi is a framework for constructing and verifying fault-tolerant distributed systems [14]. A system assumed no network failure is converted to the system which tolerates dropping packets, duplication of packets, and machine failure. One of the purposes of Actario is also to build and verify fault-tolerant distributed systems. We will introduce *supervisor* mechanism to achieve building fault-tolerant systems generally used in Erlang and Akka. NOTE: Supervisor についての説明はいるかどうか

Tony Garnock-Jones, Sam Tobin-Hochstadt, and Matthias Felleisen give a formalization of the Actor model using Coq [8]. In this paper, the operational semantics is formalized so that transition is decidable. Due to this, it is difficult to apply the formalization to realistic concurrent systems.

8. CONCLUDING REMARKS

In this paper, we present Actario, a Coq framework for describing and verifying actor-based systems. Actario is designed to support implicit naming of actors. This simplifies the description of actor systems. We have formally proved that the underlying execution model provided in the framework satisfies important properties including name uniqueness, actor persistence and message persistence. The fact implies that a system described using Actario is guaranteed to have these actor properties.

Actario is currently under development and still does not provide convenient libraries of predicates, lemmas, tactics and so forth. Thus, verifying a user-defined actor system may involve a large amount of work. Providing such libraries should be included in the future work.

In addition, we like to extend Actario to suppor extended Actor models. For example, extensions that supports high-level synchronization mechanisms such as [6], coordination models[12], and reflective models such as [13].

9. REFERENCES

[1] Actario. `https://github.com/amutake/actario`.

[2] R. Affeldt and N. Kobayashi. A Coq library for verification of concurrent programs. In *Fourth International Workshop on Logical Frameworks and Meta-Languages (LFM 2004)*, volume 199 of *Electronic Notes in Theoretical Computer Science*, pages 17–32, 2008.

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

[4] Akka. `http://akka.io/`.

[5] K. Arkoudas. Athena. `http://proofcentral.org/athena`.

[6] J. De Koster, T. Van Cutsem, and T. D'Hondt. Domains: Safe sharing among actors. In *Proceedings of the 2nd edition on Programming Systems, Languages and Applications based on Actors, Agents, and Decentralized Control Abstractions (AGERE!@SPLASH 2012)*, pages 11–22. ACM, ACM, 2012.

[7] Erlang programming language. `http://www.erlang.org/`.

[8] T. Garnock-Jones, S. Tobin-Hochstadt, and M. Felleisen. The network as a language construct. In *Programming Languages and Systems (ESOP 2014)*, volume 8410 of *Lecture Notes in Computer Science*, pages 473–492. Springer-Verlag, 2014.

[9] D. R. Musser and C. A. Varela. Structured reasoning about actor systems. In *Workshop on Programming based on Actors, Agents, and Decentralized Control (AGERE!@SPLASH 2013)*, pages 37–48. ACM, oct 2013.

[10] The Scala programming language. `http://scala-lang.org/`.

[11] M. Sirjani and M. M. Jaghoori. Ten years of analyzing actors: Rebeca experience. In G. Agha, O. Danvy, and J. Meseguer, editors, *Formal Modeling: Actors, Open Systems, Biological Systems*, volume 7000 of *Lecture Notes in Computer Science*, pages 20–56. Springer-Verlag, 2011.

[12] C. Talcott, M. Sirjani, and S. Ren. Comparing three coordination models: Reo, ARC, and PBRD. *Science of Computer Programming*, 76(1):3–22, 2011.

[13] T. Watanabe. Towards a compositional reflective architecture for actor-based systems. In *Workshop on Programming based on Actors, Agents, and Decentralized Control (AGERE!@SPLASH 2013)*, pages 19–24. ACM, oct 2013.

[14] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 357–368, New York, NY, USA, 2015. ACM.

[15] A. Yonezawa, J.-P. Briot, and E. Shibayama. Object-oriented concurrent programming in ABCL/1. In *OOPSLA '86 Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, pages 258–268, 1986.

APPENDIX

A. LABELED TRANSITION SEMANTICS IN ACTARIO

```
Reserved Notation "c1 '~(' t ')⇝' c2" (at level 60).
Inductive trans : label → config → config → Prop :=
(* receive transition *)
| trans_receive :
    ∀ to from content f gen sendings_l sendings_r
     actors_l actors_r,
      (sendings_l ++ Build_sending to from content ::
     sendings_r)
                 ⋈ (actors_l ++ Build_actor to (become
     (receive f)) gen :: actors_r)
        ~(Receive to from content)⇝
        (sendings_l ++ sendings_r) ⋈ (actors_l ++
     Build_actor to (f content) gen :: actors_r)
(* send transition *)
| trans_send :
    ∀ from to content cont gen sendings_l sendings_r
     actors_l actors_r,
      (sendings_l ++ sendings_r)
                ⋈ (actors_l ++ Build_actor from (send
     to content cont) gen :: actors_r)
         ~(Send from to content)⇝
         (sendings_l ++ Build_sending to from content
     :: sendings_r)
           ⋈ (actors_l ++ Build_actor from cont gen ::
     actors_r)
(* new transition *)
| trans_new :
    ∀ parent behv cont gen sendings actors_l actors_r,
      sendings ⋈ (actors_l ++ Build_actor parent (new
     behv cont) gen :: actors_r)
        ~(New (generated gen parent))⇝
        sendings ⋈
          (Build_actor (generated gen parent) (become
     behv) 0 ::
           actors_l ++
           Build_actor parent (cont (generated gen parent
     )) (S gen) ::
           actors_r)
(* self transition *)
| trans_self :
    ∀ me cont gen sendings actors_l actors_r,
      sendings ⋈ (actors_l ++ Build_actor me (self cont
     ) gen :: actors_r)
        ~(Self me)⇝
        sendings ⋈ (actors_l ++ Build_actor me (cont me
     ) gen :: actors_r)
where "c1 '~(' t ')⇝' c2" := (trans t c1 c2).
```

Figure 16: Labeled Transition Semantics in Actario