Structured Reasoning About Actor Systems

David R. Musser Carlos A. Varela

Rensselaer Polytechnic Institute

{musser,cvarela}@cs.rpi.edu

Abstract

The actor model of distributed computing imposes important restrictions on concurrent computations in order to be valid. In particular, an actor language implementation must provide *fairness*, the property that if a system transition is infinitely often enabled, the transition must eventually happen. Fairness is fundamental to proving progress properties. We show that many properties of actor computation can be expressed and proved at an abstract level, independently of the details of a particular system of actors. As in abstract algebra, we formulate and prove theorems at the most abstract level possible, so that they can be applied at all more refined levels of the theory hierarchy. Our most useful abstract-level theorems concern persistence of actors, conditional persistence of messages, preservation of unique actor identifiers, monotonicity properties of actor local states, guaranteed message delivery, and general consequences of fairness. We apply the general actor theory to a concrete ticker and clock actor system, proving several system-specific properties, including conditional invariants and a progress theorem. We develop our framework within the Athena proof system, in which proofs are both human-readable and machine-checkable, taking advantage of its library of algebraic and relational theories.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—correctness proofs

Keywords actor model; open distributed systems; fairness; persistence properties; progress properties; transition induction; Athena

1. Introduction and motivation

The actor model [1, 14] is useful both as a theoretical framework for reasoning about concurrent computation [2, 22] and as a practical paradigm for building distributed systems [6, 25]. An actor is simultaneously a unit of state encapsulation and a unit of concurrency, which makes it a natural unit of distribution, mobility, and adaptivity in open systems [11]. Actors have unique identifiers and communicate via asynchronous message passing. In response to a message, an actor may change its internal state, create new actors with a specified behavior (including an initial state), and/or send messages to known actors.

[This is a minor revision of a paper of the same title that the authors presented at AGERE!'13: 3rd International Workshop on Programming based on Actors, Agents, and Decentralized Control, October 2013, Indianapolis, Indiana. That paper referenced specification and proof files available with the initial Athena release on http://proofcentral.org (reference [19]); in this revision the reference is updated to correspond to improved files available with Athena version 1.1.]

Actor theories formalize computation as a labeled transition system between actor configurations, where an actor configuration represents the potentially distributed state of a system at a single logical point in time. Transitions from an actor configuration specify the possible ways in which an actor computation may evolve. The actor model imposes *fairness* on computation sequences in order to be valid. Fairness means that if a transition (from an actor configuration) is infinitely often enabled, the transition must eventually happen. Without fairness, it is not possible to reason *compositionally*. An actor system correctness property (e.g., a web server always replying to a client request) would no longer hold when composed with another system (e.g., a web crawler), since the computation may evolve as a sequence of transitions that consistently ignores the web server actors, effectively constituting a denial of service attack. The actor model precludes such *unfair* computation paths.

Actor languages can use different models for representing sequential computation within an actor. Agha, Mason, Smith, and Talcott use the untyped call-by-value lambda calculus to represent an actor's internal behavior [2], whereas Varela and Agha use an object's instance and class to represent an actor's state and its behavior [25]. In this paper, we define behavior within an actor with axioms on certain functions and relations on their local states.

While local state axioms are specific to a concrete actor system, it is desirable to describe the way that actors send and receive messages more abstractly, so that one can derive general theorems—ones that can be applied to many different actor systems—about properties such as actor persistence, fairness, name uniqueness, and infinitely-often-enabled transitions.

Let us illustrate the actor model using a simple example with two actors: *Ticker*, which repeatedly sends *tick* messages to *Clock*, which, upon receipt of each tick increments an internal counter representing a time value. The main complexity of the example lies in the *Ticker*'s behavior: it keeps going by sending a *continue* message to itself, sending a tick message to *Clock*, and receiving the continue message, thus repeatedly moving through a sequence of three control states. This pattern of (self-)message passing to implement iterative behavior is typical of actor systems [14, 24].

The *unbounded nondeterminism* property means that messages are eventually received but there is no bound assumed on how many transitions may take place beforehand. In the context of the Ticker-Clock example, we can rephrase unbounded nondeterminism as the property whereby the clock may wait an arbitrarily long time (as measured by the number of accumulated ticks) to receive a tick, but eventually it does and therefore makes progress in incrementing its own time value.

Fairness is critical to proving this progress property, since without fairness, the ticker could keep producing tick messages indefinitely without any of them being received by the clock. Yet, as will be seen from the definitions of fair and infinitely-often enabled transitions in actor systems in Section 4, we impose no bounds on responses to messages, so unbounded nondeterminism holds.

In this paper we show how to carry out abstract reasoning about actor systems, including issues of fairness and other key properties such as actor persistence and preservation of uniqueness of actor identifiers, in the Athena proof language and system [4]. In Section 2 we begin the development of the actor model theory at an abstract level. In terms of conceptual organization, our development carves out a richly structured hierarchy of formal theories that can be used to represent and reason about actor systems. In Section 3 we show how to specify an actor's local behavior, using the Ticker and Clock actors as examples. In Sections 4 and 5 we return to the abstract level, presenting the most useful theorems we have conjectured and proved, with some discussion of their application to the Ticker-Clock system. In Section 6, we apply a combination of the general actor theory and the Ticker-Clock specifics to prove several more properties of that system, including a progress theorem that depends on fairness. Section 7 discusses related work and Section 8 concludes with thoughts on future extensions of this work. Appendix A describes highlights of the Athena proof language. Appendix B contains the full Athena proof of a result we call *actor-persistence*, except that five lemmas used in the proof are not shown, though they are briefly described in the body of the paper. But the entire theory and proof development can be found online [5, 19], together with tutorial material and links to download Athena and its libraries [3].

2. Developing actor system axioms, theorems, and proofs

Purposes of proofs about computation include (1) correctness: getting it right; (2) design debugging: identifying flaws at an early stage of development, and (3) education: better understanding of how computations work. Although we consider the first two purposes important, this work is aimed more at the third. Consequently, we choose to develop proofs in a style that is not only machine-checkable but also human-readable. This approach may require greater human effort than when a more fully automated prover is the main tool, but keep in mind that it is an effort invested in education, not just correctness or debugging. To overcome some of the difficulty, we emphasize moving theory and proofs to an abstract level wherever possible, so that the abstract theory can be applied to other problems (and, often, multiple times within a single problem). This kind of activity is much closer to traditional mathematical theory development than most work based on automated theorem proving: rather than relying so much on the computational power of automated provers, we concentrate instead on providing a readable and well-organized theory development. A major benefit of this approach is insight: when a proof attempt succeeds we can study it in detail for lessons to apply in other efforts, and when it fails we can much more easily track down what is wrong than we can when an automated prover simply reports failure.

In this and the following sections we give an overview of our actor theory and its application to the Ticker-Clock example. In Section 7, we note how both the theory and the Ticker-Clock example bear many similarities to, but some differences from, earlier formulations by other authors. The main innovations of our work lie in putting the theory and proofs in a machine-checkable form that is also suitable for educational purposes.

In the rest of this section, we first build a foundation consisting of a theory of configurations based on the Abelian Monoid algebraic theory, a refinement of that theory into a theory of actor-system configurations, and a theory of actor-system transitions. Each of these theories is abstract, in the sense of being applicable to many different cases via further refinement and/or instantiation. Figure 1 illustrates the refinement relation between the algebraic and relational theories and the actor theory development to be presented.

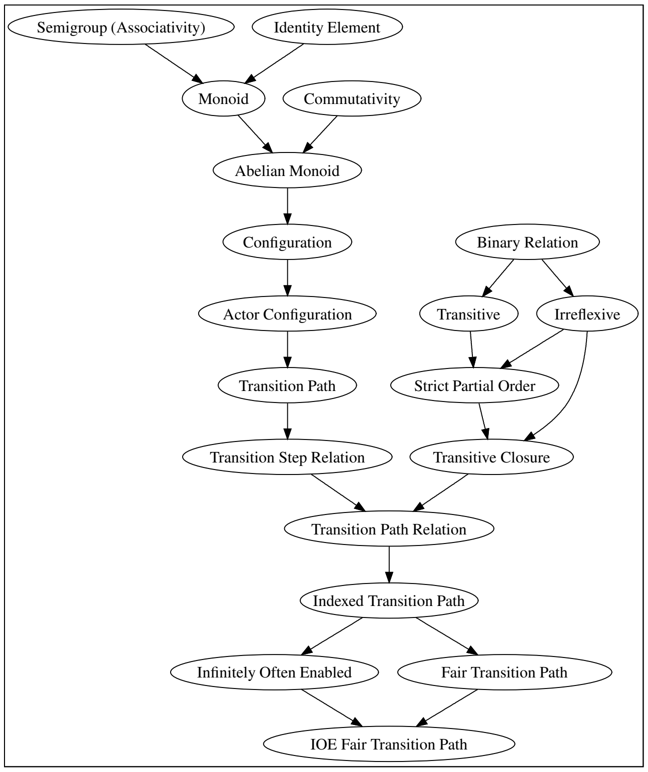

Figure 1. Algebraic, Relational, and Actor Theories

2.1 Configurations

A configuration is essentially a "soup" of components. Formally, we have a polymorphic structure `(Cfg T)` with constructors `Null`, `One`, and `++`:

```
structure (Cfg T) :=
  Null | (One T) | (++ (Cfg T) (Cfg T))
```

where `T` is any sort, and `Null` and `++` are subsequently axiomatized to form an Abelian Monoid; i.e., `Null` is the Monoid identity element for the binary `++` operator, which is associative (a Monoid property) and commutative (Abelian).

At this level we introduce composition theorems, which provide for combining the information from two different views of a configuration into a single, unified view. For example,[1]

```
define [a b s s1 s2] := [?a ?b ?s ?s1 ?s2]
define Together :=
```

[1] In input to Athena, any binary function symbol, such as ++, can be written in infix position and is right-associative by default. Associativity and precedence can be controlled with user declarations. Unary function symbols, such as `One`, can be applied with or without surrounding parentheses. With a function symbol f of arity $n > 2$, prefix syntax $(f\ a_1\ \ldots\ a_n)$ is used. Athena's logic is many-sorted first-order, with the added advantage of Hindley-Milner-style polymorphism and sort-inference to deduce sorts automatically. The logical operators are, in order of decreasing precedence: `~` (not), `&` (and), `|` (or), `==>` (implies), `<==>` (equivalent to). Quantified variables begin with `?`, but other identifiers can be used by defining them to be quantified variables.

```
  (forall s s1 s2 a b .
    s = s1 ++ One a &
    s = s2 ++ One b &
    a =/= b
    ==> exists s3 . s = s3 ++ One a ++ One b)
```

This and similar theorems for larger numbers of components (`Three-Together`, `Four-Together`) are simple and intuitive, but are crucial lemmas in proofs in the more complex theories that follow.

2.2 Actors and actor configurations

Actors and messages are then introduced, defined as applications of constructors of an abstract data type, `Actor`, parametrized by an identifier sort, `Id`, and an actor local state sort, `LS`.

```
datatype (Actor Id LS) :=
 (actor' Id LS) | (message' Id LS Id Ide)
```

where `Ide` is a predefined sort of quoted strings. Using the `One` constructor of the configuration structure we coerce `actor'` and `message'` values into configuration components:[2]

```
define actor := lambda (id ls) (One (actor' id ls))
define message :=
  lambda (fr to c) (One (message' fr LS0 to c))
```

For example, a configuration that is possible with the Ticker-Clock example is

```
  s ++ (actor Ticker (ticker local3)) ++
  (actor Clock (clock local zero)) ++
  (message Ticker Ticker 'continue) ++
  (message Ticker Clock 'tick)
```

where `s` denotes a configuration whose composition is unknown but potentially contains other messages for the `Ticker` and `Clock` actors and potentially other actors and messages that are entirely independent of the `Ticker` and `Clock` interactions.

A `unique-ids` predicate is introduced for expressing the precondition on a configuration that the identifiers of the actors in it are unique. Several simple theorems about `unique-ids` are proved. Uniqueness of actor identifiers is crucial; how this property is maintained is discussed in Section 2.4.

2.3 Transition paths

We continue with the development of a `Transition-Path` datatype and theory, used to describe how configurations change in response to transition steps—either message receiving, message sending, or actor creation steps.

```
datatype (Step Id) :=
   (receive Id Id Ide)
 | (send Id Id Ide)
 | (create Id Id)
datatype (TP Id LS) :=
   Initial
 | (then (TP Id LS) (Step Id))
declare config:
  (Id, LS) [(TP Id LS)] -> (Cfg (Actor Id LS))
declare ready-to:
  (Id, LS) [LS (Step Id)] -> Boolean
declare next:
  (Id, LS) [LS (Step Id)] -> LS
```

[2] The local state constant `LS0` is included in messages for a purely technical reason, as explained in Appendix B.

For example, we might have a transition path `T0` with `config T0` equal to the configuration shown above and another transition path

```
  T1 = (T0 then (receive Clock Ticker 'tick))
```

for which

```
config T1 = s ++ (actor Ticker (ticker local3)) ++
            (actor Clock (clock local (S zero))) ++
            (message Ticker Ticker 'continue)
```

where `S` is the natural number successor function. This result is determined by a theorem in `Transition-Path` theory,

```
 define trans-receive :=
  (forall T id s ls fr c .
    config T = s ++ (actor id ls) ++
               (message fr id c) &
    ls ready-to (receive id fr c)
    ==>
    config (T then (receive id fr c)) =
    s ++ (actor id (next ls (receive id fr c))))
```

combined with the next-state function of the `Clock` actor (see Section 3, which also discusses other examples of computation in the Ticker-Clock system).

Upon this fundamental `Transition-Path` theory we define a binary relation on transition paths, written `T0 -->> T1`; we say `T0` "directly-leads-to" `T1`. The resulting theory is called `Transition-Step-Relation`. `Transition-`*`Path`*`-Relation` theory then refines both `Transition-Step-Relation` theory and a fundamental `Transitive-Closure` theory from Athena's main library, yielding binary relations `-->>+` and `-->>*`, the irreflexive and reflexive transitive closures of `-->>`, resp. One additional axiom is adopted in this theory:

```
  define nothing-leads-to-Initial :=
    (forall T . ~ T -->>+ Initial)
```

2.4 Proving unique-ids persistence with transition induction

The `-->>+` and `-->>*` relations are useful for expressing theorems about how a point in a transition path relates to a point arbitrarily further along in the path. For example,

```
define unique-ids-persistence :=
  (forall T T0 .
    (unique-ids config T0) & T0 -->>* T
    ==> (unique-ids config T))
```

Maintaining uniqueness of actor identifiers is crucial for many basic properties of actor systems, the most obvious being composability. Our approach in the logic is to start with the assumption of identifier uniqueness and show that it persists, by proving the above theorem. The proof is by induction. In the basis case, `T = Initial`, the assumption `(T0 -->>* Initial)` is divided by the definition of `-->>*` into two subcases, `(T0 = Initial)` or `(T0 -->>+ Initial)`. In the = case, we get the conclusion from the assumption `(unique-ids config T0)`, and the `-->>+` case cannot occur.

In the inductive step, `send` and `receive` transitions don't affect uniqueness of ids, and the `create` transition has as a precondition the uniqueness of the identifiers in the configuration that includes the new actor.

In Athena, this induction proof is encased in some preparatory code, such as introducing the lemma from `Transitive-Closure` theory:

```
let {R*L := (!lemma ['-->> R*-lemma])}
```

and is more detailed, but this form of induction, which we call *transition induction*, is naturally expressed using Athena's built-in **by-induction** form, which adapts the basis and induction step cases according to the relevant **datatype** declaration:

```
by-induction unique-ids-persistence {
  Initial => ...
| (T then step) =>
  let {ind-hyp := (forall ?T0  .
                     (unique-ids config ?T0) &
                     (?T0 -->>* T)
                    ==> (unique-ids config T));
       goal := (unique-ids
                   config (T then step))}
     ...
  }
```

The basis case proof reads more like program code than the informal language we used above—it *is* program code, as Athena's proof language is a programming language specialized for inference—but is nevertheless faithful to the basic line of argument:

```
Initial =>
 pick-any T0
  let {A1 := (unique-ids config T0);
       A2 := (T0 -->>* Initial)}
  assume (A1 & A2)
    let {B1 := (!chain->
                [A2 ==> (T0 = Initial |
                         T0 -->>+ Initial)
                             [R*L]])}
     (!cases B1
      assume C1 := (T0 = Initial)
        (!chain->
         [A1 ==> (unique-ids config Initial)
                 [C1]])
      assume C2 := (T0 -->>+ Initial)
        (!from-complements
         (unique-ids config Initial)
         C2
         (!chain->
          [true ==> (~ C2)
               [nothing-leads-to-Initial]])))
```

This applies chain->, a **method**—Athena's proof-language counterpart of a procedure—that implements *implication chaining*. In general, the implication $S_0 \Rightarrow S_n$ can be proved by

$$(!\texttt{chain}\ [S_0 \Rightarrow S_1\ [J_1] \Rightarrow S_2\ [J_2] \Rightarrow \cdots \Rightarrow S_n\ [J_n]])$$

where the S_i are sentences and the J_i are justifications that prove $(S_{i-1} \Rightarrow S_i), i = 1, \ldots, n$. If S_0 has already been proved, the variant

$$(!\texttt{chain->}\ [S_0 \Rightarrow S_1\ [J_1] \Rightarrow S_2\ [J_2] \Rightarrow \cdots \Rightarrow S_n\ [J_n]])$$

proves S_n. (The chain method also implements equality chaining to prove equations; see Appendix B.)

The Athena proof of the induction step case is a bit too long to include here but is available online [19, transition.ath]. The complete proof is one of the simpler applications of transition induction, and is thus a good first exercise for carrying out such proofs in the proof system. A somewhat more complex proof by transition induction is given in full in Appendix B.

3. Expressing actor local computation

Local actor behavior is expressed in our actor language using fun, a function-defining mechanism with syntax similar to ML's fun declarations. But a fun definition is translated by Athena into a set of equational axioms, which, together with Athena's built-in capabilities for equational reasoning (basically a targeted form of term or predicate rewriting), become the basis for reasoning about the function.

An actor's local behavior is defined with two fun definitions, one of a ready-to binary predicate that expresses that the actor's local state is ready to participate in some (perhaps more than one) kind of transition, and the other a next function of the actor's local state and a transition step, producing the new local state the actor acquires when it actually does participate in the transition. For example, in the Ticker-Clock example, the Ticker actor has three states: local1, local2, and local3. In state local1, it is ready to send a "continue" message to itself, and when that transition occurs it moves into state local2. In that state, it is ready to send a "tick" message to Clock and move into state local3. Finally, in state local3, it is ready to receive a continue message and return to state local1, starting the processing over. This kind of self-message-passing looping is frequently used in actor systems in order to avoid using loops in actor local computations (thereby avoiding infinite loops, which would disable the actor's participation in system transitions).

Thus, we have the following Athena fun definitions for the Ticker actor:[3]

```
module Clock-Actors {
 ...
 datatype Name := Ticker | Clock
 # For a more general treatment, we would use
 # a datatype with infinitely many values, like
 #   datatype Name := Zeroth | (Next Name)
 datatype TLS := local1 | local2 | local3
 datatype CLS := (local N)
 datatype TCLS := (ticker TLS) | (clock CLS)
 ...
 declare ready-to: [TCLS (Step Name)] -> Boolean
 declare next: [TCLS Name (Step Name)] -> TCLS

 module Ticker {
 assert ready-to-definition :=
  (fun
   [(ls ready-to
      (send Ticker Ticker 'continue))
        <==> (ls = ticker local1)
    (ls ready-to
      (send Ticker Clock 'tick))
        <==> (ls = ticker local2)
    (ls ready-to (send Ticker id c)) <==>
      [false  when
        (~ (id = Ticker & c = 'continue) &
         ~ (id = Clock & c = 'tick))]
    (ls ready-to
      (receive Ticker Ticker 'continue))
        <==> (ls = ticker local3)
    (ls ready-to
      (receive Ticker fr c)) <==>
        [false  when
          (~ (fr = Ticker & c = 'continue))]
    (ls ready-to (create Ticker id'))
       <==> false])
```

[3] Athena **module**s are a syntactic construct providing separate namespaces in a fairly standard manner: An identifier I defined in module M can be referenced simply as I within the scope of M; immediately outside of M it is referenced as $M.I$.

```
let {continue :=
      (send Ticker Ticker 'continue);
     tick := (send Ticker Clock 'tick)}
assert next-definition :=
  (fun
   [(next (ticker local1) step) =
     [(ticker local2) when (step = continue)
      (ticker local1) when (step =/= continue)]
    (next (ticker local2) step) =
     [(ticker local3) when (step = tick)
      (ticker local2) when (step =/= tick)]
    (next (ticker local3) step) =
     [(ticker local1)  when
       (step = (receive Ticker Ticker 'continue))
      (ticker local3)  when
       (step =/=
            (receive Ticker Ticker 'continue))]])
  ...
 } # module Ticker
 ...
} # module Clock-Actors
```

Clock's behavior is simpler, in that there is only one control state, but also richer in that its state contains a natural number (used to record the number of tick messages it has received):

```
module Clock {
assert ready-to-definition :=
 (fun
  [(ls ready-to (receive Clock fr c))
      <==> (fr = Ticker & c = 'tick &
            exists t . ls = clock local t)
   (ls ready-to (send Clock to c)) <==> false
   (ls ready-to (create Clock id')) <==>
     (exists t . ls = clock local t)])

assert next-definition :=
  (fun
   [(next (clock (local t)) (receive id fr c)) =
     [(clock (local (S t)))
        when (id = Clock & fr = Ticker & c = 'tick)
      (clock (local t))
        when (~ (id = Clock & fr = Ticker
                            & c = 'tick))]
    (next (clock (local t)) (send id to c)) =
       (clock (local t))
    (next (clock (local t)) (create id id')) =
       (clock (local t))])
...
} # module Clock
```

In general an actor's behavior could combine these features of multiple control states and data-bearing states.

3.1 Testing with specific transition paths

Before proceeding to develop theorems and proofs about the Ticker-Clock system, we can gain some understanding by constructing specific transition paths and observing their effect on actor configurations. Consider, for example, the transition path

```
define P :=
 (Initial then (create Clock Ticker)
          then (send Ticker Ticker 'continue)
          then (send Ticker Clock 'tick)
          then (receive Clock Ticker 'tick)
          then (receive Ticker Ticker 'continue)
          then (send Ticker Ticker 'continue)
          then (send Ticker Clock 'tick)
          then (receive Ticker Ticker 'continue)
          then (send Ticker Ticker 'continue)
          then (send Ticker Clock 'tick)
          then (receive Clock Ticker 'tick)))
```

in which actor Clock creates actor Ticker, which then sends a continue message to itself and a tick message to Clock; the tick is received; the continue message is received and resent; another tick message is sent; the continue message is received and resent; and another tick is sent and a tick is received. Assuming that the time value held by Clock in Initial is zero and actor identifiers are unique, the value held at the end of the path should be two, and there should be one tick message still to be received:

```
define M :=
  (One (message' Ticker ls0 Clock 'tick))
define CM :=
  (One (message' Ticker ls0 Ticker 'continue))

define P-result :=
 (config Initial =
   s0 ++ (actor Clock (clock local zero)) &
  (unique-ids
    (config Initial) ++
    (actor Ticker
           (new-ls (clock local zero)))) &
  (clock (local zero))
    ready-to (create Clock Ticker)
  ==>
  config P =
    s0 ++ (actor Clock
            (clock local (S (S zero)))) ++
    (actor Ticker (ticker local3)) ++ M ++ CM)
```

where new-ls is a function that creates a local state for the new actor from the creating actor's local state. In [19, clock-actors_run.ath], we show how to develop a proof of this specific result based on the general transition path theory and the concrete implementation of the Ticker-Clock system. For more general theorems about the actor model, we need to extend the theory further.

4. Fairness

Fairness in actor systems is basically the property that if a transition is infinitely-often enabled, it will eventually happen. To express fairness in terms we can reason about in formal logic, we need a way of expressing eventuality in the logic. Athena's many-sorted first-order logic is quite expressive but offers no eventuality quantifier of the kind one would have in a temporal logic [17]. But one can easily express eventuality by indexing the points in a transition path with natural numbers. We can then say that a transition must eventually occur in an indexed path by saying there exists an index position in the path, greater than or equal to the index of the current position, at which the transition in question appears. This approach seems quite natural, being both readable and not at all difficult to reason about. For example, reflexivity, asymmetry, and transitivity properties of eventuality follow easily from corresponding properties of natural numbers, which are both familiar and already present in Athena's library of natural number theorems and their proofs.

We first need to express "infinitely-often enabled" in first-order logic. In terms of indexed paths, we simply say that, at any index in a transition path at which a given transition is enabled, either it occurs at that index or there is a greater index at which it is enabled. Here is how this works out for a receive transition:

```
define IOE-receive :=
 (forall T n s id ls fr c .
   config (itp T n) =
     s ++ (actor id ls) ++
     (message fr id c) &
   ls ready-to (receive id fr c)
   ==> (itp T (S n)) =
        (itp T n) then (receive id fr c)
    | exists k s' ls' .
       k > n &
       config (itp T k) =
         s' ++ (actor id ls') ++
         (message fr id c) &
       ls' ready-to (receive id fr c))
```

In checking proofs of abstract level theorems we derive `IOE-receive` and corresponding theorems for `send` and `create` transitions from a more general `IOE` axiom, but in working with a concrete example we retract that axiom and derive it as a theorem instead from the actor implementations. Thus in the Ticker-Clock example, we derive it from the implementations discussed in Section 3.

The expression of fairness in our formulation is also general, but we discuss it here only for the `receive` case: if a `receive` transition is enabled at index n_1 then either (1) at some index $n_2 \geq n_1$ the `receive` transition occurs, or (2) there is no index $n_3 > n_2$ at which the `receive` transition is (again) enabled:

```
define fair-receive :=
 (forall T n1 s id ls fr c .
   config (itp T n1) =
     s ++ (actor id ls) ++
     (message fr id c) &
   ls ready-to (receive id fr c)
   ==> (exists n2 .
         n2 >= n1 &
         (itp T (S n2)) =
          (itp T n2) then (receive id fr c)
       | ~ exists n3 s3 ls3 .
           n3 > n2 &
           config (itp T n3) =
             s3 ++ (actor id ls3) ++
             (message fr id c) &
           ls3 ready-to (receive id fr c))
```

Under the assumption of infinitely-often-enabled, the second result is ruled out and we only have the first result: the transition does eventually occur:

```
define receive-happens :=
  (forall T n s id ls fr c .
    config (itp T n) =
      s ++ (actor id ls) ++
      (message fr id c) &
    ls ready-to (receive id fr c)
    ==> exists k .
          k >= n &
          (itp T (S k)) =
            (itp T k) then (receive id fr c))
```

One way to view the IOE and fairness concepts is as a bargain between actors and the overall actor system. In terms of receiving a message (similar observations can be made in the `send` and `create` case), an actor agrees to be, if not constantly then at least periodically, ready to receive the message until it is actually received. The system, on the other hand, agrees to eventually enter a `receive` transition at some future point when the actor is ready. The actor's behavior must be specified, and correctly implemented, to avoid infinite loops or other problems that would prevent it from ever getting back to being ready to receive, and system scheduling must be such that other transitions cannot preempt the message reception forever.

5. Additional theorems at the abstract level

In this section we discuss various theorems we proved at an abstract-level, in addition to the composition, transition, and fairness theorems already discussed in previous sections. These theorems fall into three broad categories: persistence, monotonicity, and progress theorems. We discuss them in turn, in some cases also mentioning applications to the Ticker-Clock example.

5.1 Persistence theorems

We developed various "persistence" properties of actor systems. Unique-ids persistence has already been discussed in Section 2.4.

5.1.1 Actor persistence

Actor persistence in our formalism is even more obvious than unique-ids persistence, since in none of the available transitions does an actor disappear. But working out the proof using transition induction is a bit more involved, since to show that a given actor, with identifier `id0`, say, persists, one must consider for a `receive` transition involving actor `id`, two cases: one in which $\texttt{id} = \texttt{id0}$, and one in which $\texttt{id} \neq \texttt{id0}$; and similarly for `send` and `create` transitions. The unequal case, where some other actor is the subject of the transition, is easier and almost identical for all three kinds of transitions, making it worthwhile to develop a set of lemmas for dealing with it. These "other" lemmas have helped to shorten not just the actor persistence proof but almost all of the transition induction proofs in our work. See Appendix B for the Athena proof of actor-persistence and of an "other" lemma.

5.1.2 Message persistence (conditional)

Of course, messages do disappear from a configuration when they are received, so any claim about message persistence in a transition path must be conditional on either it not being received in the path or, if it is, it being subsequently restored by being resent. In the Ticker-Clock example, we have the latter situation with the continue message that the ticker sends itself. Upon receiving it in state `local3`, it moves into state `local1`, and waits for a `send` transition to resend it, thereupon moving into state `local2`, in which it is ready to send a tick message to the clock, which will get it back to state `local3`. We can thus say that the continue message persists in the sense that the following conditional invariant is maintained:

```
 config T = s ++ (actor Ticker (ticker ls))
 ==> (ls =/= local1 ==>
          exists s' .
           s = s' ++ (message Ticker Ticker 'continue))
```

We have proved this invariant by transition induction, but the proof is unpleasantly long, dealing as it does with the many cases that arise due to `Ticker`'s three local states, the three kinds of transitions, and the possibility that a transition does or does not involve `Ticker`. Even though it is much shortened by use of the "other" lemmas (Section 5.1.1) for the transition cases that do not involve `Ticker`, in the cases where it does it is not as easy to see regularities that would allow them to be covered by general lemmas. But we continue studying how to lift such theorems and proofs to an abstract level, so that they can be applied in many different cases with little further proof effort.

5.2 Monotonicity (of actor local states)

We might need to know that not only does an actor persist as transition paths are traversed, but also some monotonicity property of its local state is preserved. E.g., in the Ticker-Clock system, it should be the case that the time value held by the clock is nondecreasing from one point in the path to the next.

At the abstract level, we have the following theorem, in which binary relation `R` is only assumed to be a preorder (reflexive and transitive).

```
define actor-monotonicity :=
  (forall T T0 s0 id ls0 .
    (forall ls step .
      ls R (next ls step))
    ==>
    ((unique-ids config T0) &
     config T0 = s0 ++ (actor id ls0) &
     T0 -->>* T
     ==>
     exists s ls .
       config T =
         s ++ (actor id ls) & ls0 R ls))
```

The proof [19, monotonic-transition.ath] is again by induction on transitions.

With R taken to be the relation between the times held in clock local states, the preconditions of the theorem are satisfied by the clock axioms [19, clock-actors.ath]. We thus have

```
define clock-monotonicity :=
 (forall T T0 s0 t0 .
   (unique-ids config T0) &
   config T0 =
     s0 ++ (actor Clock (clock local t0)) &
   T0 -->>* T
   ==> exists s t .
        config T =
         s ++ (actor Clock (clock local t)) &
        t >= t0)
```

5.3 Progress

The main result here is a guaranteed message delivery theorem, which states that if an actor is ready to send a message and actor ids are unique, then the message will eventually be received by its intended recipient.

```
define
  guaranteed-message-delivery :=
 (forall T n0 s0 fr to ls0 ls1 c .
  config (itp T n0) =
    s0 ++ (actor fr ls0) ++
    (actor to ls1) &
  ls0 ready-to (send fr to c) &
  (unique-ids config (itp T n0))
  ==>
  exists n .
   n >= n0 &
   (itp T (S n)) =
     (itp T n) then (receive to fr c))
```

Stating and proving other progress theorems at the abstract level is a subject of our ongoing research. As we have done in developing all of the abstract-level theorems discussed above, we are working on details in concrete examples until it becomes apparent how to lift theorems at that level to an abstract level. In the next section we discuss parts of the proof of a progress theorem in the Ticker-Clock example.

6. Proving progress at the concrete level

```
define Clock-progress :=
  (forall t T n0 s0 t0 .
    config (citp T n0) =
      s0 ++ (actor Ticker (ticker local1)) ++
      (actor Clock (clock local t0)) &
    (unique-ids config (citp T n0))
    ==> exists n s ls u .
          n >= n0 &
          config (citp T n) =
            s ++ (actor Ticker (ticker ls)) ++
            (actor Clock (clock local u)) &
          u >= t)
```

This theorem states that for any arbitrarily large time t, if we start from a configuration satisfying the given preconditions, the clock will eventually hold a value $u \geq t$. A slightly stronger version of this theorem is proved by induction on t, i.e., ordinary natural-number induction, not transition induction. In the inductive step, one has to show that eventually Ticker emits a tick message, and eventually Clock receives it and increments its internal counter. The details are quite lengthy and depend crucially on specializations of the abstract-level persistence, fairness, and monotonicity theorems, together with the Ticker-Clock axioms and a few lemmas that are specific to that system's details. For example,

```
define Ticker-eventually-ready-to-send-tick :=
  (forall ls0 T n0 s0 .
    config (citp T n0) =
      s0 ++ (actor Ticker (ticker ls0)) &
    (unique-ids config (citp T n0)) &
    (ls0 =/= local1 ==> exists s1 . s0 = s1 ++ CM)
    ==> exists n s ls .
          n >= n0 &
          config (citp T n) =
            s ++ (actor Ticker ls) &
          ls ready-to (send Ticker Clock 'tick))
```

where citp is a clock-specialized version of the abstract level itp (indexed transition path) function and CM is the Ticker's continue message. Note the conditional message-persistence assumption; to satisfy it, we conjectured and proved the invariant previously discussed in Section 5.1.2. Note also that the lemma doesn't specify starting in a particular local state. To prove it, therefore, we must consider each of the three values ls0 can have: local1, local2, and local3. To avoid repetition in the proof, we introduced three mutually recursive methods: proof1 applies the send-happens theorem to advance by sending a continue message and moving Ticker into state (ticker local2), then calling method proof2 to finish the overall proof. State local2 is the Ticker state in which it *is* ready to send a tick to Clock, so again applying the send-happens theorem gets the job done without recursion. In state local3, the receive-happens theorem (Section 4) is applied to show the continue message is eventually received, putting Ticker back in state local1, so we can call upon the proof1 method to finish the proof. See [19, fair-clocks.ath].

7. Related work

Inspiration for two aspects of our work—using abstraction in formulating actor theories with subsequent specialization to concrete systems, and manipulating configurations using AC-rewriting—came from the work of Talcott et al. in Maude [8]. Maude's high level language and powerful AC rewriting are the foundation of its system specification and model checking capabilities, but for actually developing proofs Maude is limited to what can be expressed as equations and proved by rewriting. To express the axioms, theorems, and proofs we have developed, one needs a full-fledged theorem prover. Exploring fairness, for example, was done in [8] based on a built-in "fair rewriting" capability, frewrite, a breadth-first strategy for applying rules and equations. Our expression of fairness is more fundamental and subject to many different implementation strategies.

Athena also provides general support for model checking, which we have yet to exploit. We view it as a complementary tool, useful for "stress-testing" specifications before attempting full proofs. To date, we have only explored such testing in a limited way, such as exercising the clock system as outlined in Section 3.1. Model checking for concurrent object systems has also been explored in Creol [15], an actor-like type-safe model with an executable operational semantics using rewriting logic in Maude. The rewriting-based executable semantics provides a framework for interpretation and analysis, though model checking in general has to consider the state explosion problem. Even though state-reduction techniques such as partial order reduction [16], exploiting symmetry, and slicing [21] have been used to model check actor systems, proving program properties requiring unbounded nondeterminism such as the one illustrated in this paper is beyond the scope of model checking approaches.

Reasoning about the Ticker-Clock actor system to prove that the clock actually makes progress requires application of many of the axioms and theorems developed in this paper. Indeed, the detailed development of much of that theory was inspired by what was needed in the proofs about the Ticker-Clock system, but formulating the needed axioms and theorems at an abstract level, as we have done, makes them available for reasoning about many other actor systems. This kind of theory development and application is much closer to what mathematicians conventionally do than to the kind of proof search carried out with automatic theorem provers (ATPs). In most cases, a resolution proof or other fully-automated proof is virtually inaccessible to human understanding, so there is little insight gained from it or carry-over benefit to other proof efforts. Even in the case of proofs developed more interactively, as with Coq [9], HOL [13], Isabelle [20], PVS [10], and other "tactic-based" provers, the proofs cannot really be understood without replaying them to obtain a transcript of the steps taken toward deriving the theorem. One partial exception is Isabelle-ISAR [27], which does provide a much more readable proof language. It might be possible to replicate much of our actor model and proofs in ISAR with a similar level of readability, but some of our proofs take such full advantage of Athena's proof language (e.g., introducing several mutually-recursive methods within a proof, as discussed in Section 6) that it is not apparent to us how one could express them nearly as easily in ISAR's declarative style.

Even if human-readability is a major goal, sometimes employing a greater degree of automation can be quite useful. In Athena, one is not always restricted to writing out detailed proofs. The high-level skeleton of the proof (the important ideas) can be expressed in the language's readable proof format, while lower-level details may be outsourced to an ATP. In this work we have made only minimal use of an external ATP (Vampire [23]), in some step cases of transition induction proofs, but we are exploring additional ways in which they can be successfully applied.

8. Conclusions and future work

Concurrent software is inherently harder to verify than sequential software because of the combinatorial explosion of potential execution paths that can be produced by different schedulers. When concurrency is combined with shared state and synchronous communication, the potential for race conditions, deadlocks, and livelocks make software verification a very challenging task. Formal models of concurrent computation, e.g., the π calculus [18], the actor model [1, 2], the join calculus [12], and the ambient calculus [7] make different assumptions on shared vs distributed state and synchronous vs asynchronous communication [24]. Actor systems have significant advantages from a software development and verification perspective: first, they assume no shared state, thereby enabling modular reasoning; second, communication is purely asynchronous, thereby virtually eliminating deadlocks; third, the actor model requires fairness, helping prove application progress properties. Finally, since actors are a unit of concurrency and state encapsulation (i.e., actors process only one message at a time and share no state,) the combinatorial number of potential execution paths that need to be reasoned about to verify concurrent actor software can be significantly reduced.

Even with all these benefits, formally proving properties of actor systems requires significant expertise, time, and attention to detail. Our modular approach to reasoning advocates for a hierarchy of well-defined theories to lower the complexity of proving properties of a concrete actor system, by creating proofs that hold true for all actor programs (e.g., fair computation, actor identifier uniqueness, guaranteed message delivery) and reusing them or refining them for concrete actor system instances. Furthermore, general proof strategies, such as structural induction on operational semantics transitions, can be encapsulated in methods and applied to concrete actor systems to facilitate formal reasoning.

The results of this paper allow us to explore the interplay between different actor languages and the ability to reason about systems developed using these different languages. For example, an actor language can limit the behavior definition of an actor to move between the following four states:

1. Ready to receive a new message
2. Creating new actors in response to a message
3. Updating local state as a function of previous state, message contents, and newly created actor identifiers, and
4. Sending messages to acquaintances

Such an actor language (e.g., FeatherWeight SALSA [24]), while still Turing-complete, requires iterations to be implemented in terms of patterns of message passing [14], since there are no explicit looping constructs. However, the property that a transition is infinitely-often enabled can be proved at an abstract level, that is, it holds for all concrete actor systems implemented in such actor language. Since infinitely often enabled transitions are a pre-requisite for fairness to guarantee that the transition must eventually happen, this would greatly simplify the formalization of progress proofs. So, a general open question is: how can we restrict an actor language to facilitate formal reasoning, yet maintaining its expressive power?

Since Athena is both a computation and a deduction language, it is possible to write executable actor code that can be used to generate the axioms needed to verify its properties within the same language framework. In future work, we intend to explore this use of Athena as a provably correct actor software development framework.

Acknowledgments: The authors wish to thank Konstantine Arkoudas, Stéphane Lavirotte, and Brandon Moore for many useful comments and suggestions on earlier drafts of this paper.

References

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

[2] 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.

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

[4] K. Arkoudas. Specification, abduction, and proof. In F. Wang, editor, *Second International Symposium on Automated Technology for Verification and Analysis (ATVA 2004)*, volume 3299 of *LNCS*, pages 294–309. Springer-Verlag, 2004.

[5] K. Arkoudas and D. Musser. Athena libraries. `http://proofcentral.org/athena/1.1/lib`.

[6] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang, Second Edition*. Prentice-Hall, 1996.

[7] Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1), June 2000.

[8] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and C. Talcott. *All about Maude—a high-performance logical framework: how to specify, program and verify systems in rewriting logic*. Springer-Verlag, 2007.

[9] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[10] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A tutorial introduction to PVS. April 1995.

[11] T. Desell, K. E. Maghraoui, and C. A. Varela. Malleable applications for scalable high performance computing. *Cluster Computing*, pages 323–337, June 2007.

[12] Cédric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In *POPL*, pages 372–385, 1996.

[13] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

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

[15] Einar Broch Johnsen, Olaf Owe, and Ingrid Chieh Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365:23–66, 2006.

[16] Steven Lauterburg, Mirco Dotta, Darko Marinov, and Gul A. Agha. A framework for state-space exploration of Java-based actor programs. In *Automated Software Engineering*, pages 468–479, 2009.

[17] Zohar Manna and Amir Pnueli. *Temporal Logic*. Springer, 1992.

[18] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, Part I/II. *Journal of Information and Computation*, 100:1–77, Sept. 1992.

[19] D. Musser and C. Varela. Actor model Athena files. `http://proofcentral.org/athena/1.1/examples/actor-model`.

[20] L. Paulson. *Isabelle: A Generic Theorem Prover*. Springer, 1994. LNCS 828.

[21] Marjan Sirjani and Mohammad Mahdi Jaghoori. Ten years of analyzing actors: Rebeca experience. In Gul Agha, Olivier Danvy, and Jos Meseguer, editors, *Formal Modeling: Actors, Open Systems, Biological Systems*, volume 7000 of *Lecture Notes in Computer Science*, pages 20–56. Springer, 2011.

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

[23] The Vampire Prover development team. Vampire's Home Page. `http://www.vprover.org`.

[24] Carlos A. Varela. *Programming Distributed Computing Systems: A Foundational Approach*. MIT Press, May 2013.

[25] Carlos A. Varela and Gul Agha. Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices. OOPSLA'2001 Intriguing Technology Track Proceedings*, 36(12):20–34, December 2001.

[26] Aytekin Vargun and David Musser. Code carrying theory. In *SAC '08 Proceedings of the 2008 ACM symposium on applied computing*, pages 376–383.

[27] Markus Wenzel. *Isabelle/Isar — a versatile environment for human-readable formal proof documents*. PhD thesis, Institut für Informatik, Technische Universität München, 2002. `http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2002/wenzel.html`.

A. Appendix: Athena

Athena [3] is interactive and programmable, with separate but intertwined languages for computation and deduction. For conventional programming, the built-in computational domains include not only the usual ones of most languages—booleans, numbers, and strings—but also those typical of symbolic computation such as lists, terms and sentences of (first-order, multi-sorted) logic, substitutions, etc. The principal mechanism for program composition is the procedure call. Procedures are higher-order, i.e., they may take procedures as arguments and return procedures as results.

The principal tool for constructing proofs is the *method* call. A method call represents an inference step, and can be primitive or complex (derived). Like procedures, methods can accept arguments of arbitrary types, including other methods and/or procedures, and thus are also higher-order. Evaluation of a procedure call, if it does not raise an error or diverge, can result in a value of any type, but evaluation of a method call—again, if it does not raise an error or diverge—can result only in a *theorem*: a sentence of logic that is derived by inference from axioms and other theorems.

While there are generally many ways to express a proof, an Athena method (or a stand-alone, "straight-line" program in the proof language) is one such expression, and a key attribute of the Athena proof language is that such expressions of proofs are not only machine-checkable but also human-readable. The readability of Athena proofs rests mainly on the naturalness with which one can express important proof methods. In part, this is due to a fundamental mechanism of Athena: its *assumption base*. When a sentence is assumed or proved, it is entered into the assumption base, which is a set of sentences that each of Athena's primitive inference methods interacts with, checking one or more of its inputs to see if they are present in the set and/or making new entries. For example, `mp` is Athena's version of the *modus ponens* inference rule: `(!mp` P Q`)` checks that both P and Q are in the assumption base, and that P is an implication, $(Q \Rightarrow R)$, with Q as its antecedent. If these conditions are satisfied, then the consequent, R, of the implication is established as a theorem and entered into the assumption base. If any of the conditions fails, an error is reported. Modus ponens is one of Athena's built-in inference rules that form the foundation of its reasoning capability, but in most cases users do not need to invoke it directly. Instead, they will invoke higher-level inference methods: *equality and implication chaining, induction, case analysis*, and *proof by contradiction*. For a brief description of how Athena supports each of these methods, see [3].

In Athena, one can introduce axioms and theorems at an abstract level via *structured theories* [26]. Proofs are encapsulated in parametrized methods that allow the proofs to serve for proving theorems that are different specializations of an abstract theorem via different renamings of function symbols. A library of algebraic theories that have been developed as structured theories include *semigroup*, *monoid*, *group*, *ring*, *integral domain*, etc. Other theories collected in an Athena library include familiar relational theories: *binary-relation, reflexive, symmetric, transitive, preorder, strict weak order, total order, transitive closure*, etc. A few of these well-known algebraic and relational theories serve as building blocks for actor theories. See [5, group.ath] for the relevant algebraic theories. The relational theories upon which we build actor theories (see also Figure 1) are developed as successive *structured theory refinements*:[4]

```
module Binary-Relation {
 declare R: (T) [T T] -> Boolean [100]
 define Theory :=
```

[4] In function declarations the occurrence of a bracketed number such as `[100]` specifies a precedence value. Symbol names (and program identifiers) may include special symbols, so that `R+` and `R*` are legal symbols.

```
    (theory [] [] 'Binary-Relation)
}
module Irreflexive {
 open Binary-Relation
 define Irreflexive := (forall x . ~ x R x)
 define Theory :=
   (theory [Binary-Relation.Theory]
           [Irreflexive] 'Irreflexive)
}
module Transitive {
 open Binary-Relation
 define Transitive :=
   (forall x y z . x R y & y R z ==> x R z)
 define Theory :=
  (theory [Binary-Relation.Theory]
          [Transitive] 'Transitive)
}
module Strict-Partial-Order {
 open Irreflexive
 open Transitive
 define Theory :=
  (theory [Irreflexive.Theory Transitive.Theory]
    [] 'Strict-Partial-Order)
}
module Transitive-Closure {
 open Irreflexive
 open Strict-Partial-Order
 declare R+, R*: (T) [T T] -> Boolean [100]
 declare R**: (T) [N T T] -> Boolean
 define R**-zero :=
   (forall x y . (R** zero x y) <==> x = y)
 define R**-nonzero :=
  (forall x n y .
   (R** (S n) x y) <==>
   exists z . (R** n x z) & z R y)
 define R+-definition :=
  (forall x y . x R+ y <==>
              exists n . (R** (S n) x y))
 define R*-definition :=
  (forall x y . x R* y <==> exists n . (R** n x y))
 define Theory :=
  (theory
    [Irreflexive.Theory
     [Strict-Partial-Order.Theory 'TC [R R+]]]
    [R**-zero R**-nonzero R+-definition
     R*-definition] 'Transitive-Closure)
}
```

For further explanation of structured theories, see [3].

B. Appendix: Proof of actor-persistence

We give a listing of the proof of the theorem

```
define actor-persistence :=
 (forall T T0 s0 id ls0 .
   config T0 = s0 ++ (actor id ls0) &
   T0 -->>* T
   ==> exists s ls .
         config T = s ++ (actor id ls))
```

discussed in Section 5.1.1. The proof is encased in an Athena method that has standard parameters and begins with standard definitions according to requirements and conventions used in theorems and proofs in structured theories. The `theorem` parameter is passed the theorem to be proved when this proof is returned from a search of the theory structure (since such methods can contain the proof of more than one theorem). The `adapt` parameter is passed a function that maps function identifiers of the general theory such as `ready-to` and `next`, to function identifiers defined in a concrete actor system such as the Clock and Ticker `ready-to` and `next` functions defined in the `Clock-Actors` module (Section 3). In lines 10–11, `adapt` is applied to identifiers that appear in the text of the proof. (`LS0` is a local state constant that is included in `message` expressions only to eliminate sort-inference problems that would otherwise occur in a few places in Athena proofs. We do not use it to actually pass actor state values in messages, though such information could always be passed by including a serialization of it in the `Ide` parameter of a message structure.) The identifier `Transition-Path.Theory` in lines 3 and 4 refers to a theory definition within the `Transition-Path` module and is the entry point into the theory structure for searches for applicable axioms and theorems. For further explanation of these preliminaries, see [3].

Several of the most useful higher-level inference forms and methods supported by Athena are illustrated:

- Induction, with **`by-induction`**: lines 13–111.
- Case analysis, with `cases`: lines 21–27; with `two-cases`: lines 63–110, and with **`datatype-cases`** (like **`by-induction`**, but with no induction hypothesis): lines 65–103.
- Implication chaining, with `chain->`: lines 23–24.
- Equality chaining, with `chain`: lines 67–68. In general,

  `(!chain [`t_0 `=` t_1 `[`J_1`]` `=` t_2 `[`J_2`]` `=` $\cdots$ `=` t_n `[`J_n`]])`

  proves $t_0 = t_n$, where the justification J_i proves $t_{i-1} = t_i$, $i = 1, \ldots, n$.
- Mixed implication/equality chaining: lines 81–88.
- Proof by contradiction, with `from-complements`: lines 26–27. In general `(!from-complements` P Q R`)` proves P, where Q and R are complementary, i.e., one is the negation of the other. The most general form of proof by contradiction (not illustrated) is

  `(!by-contradiction` P
  `assume` Q
  D`)`

  which proves P, where the assumption Q is complementary to P and deduction D derives `false` using Q and other sentences that are in the assumption base.

Finally, lines 113–144 of the listing are the proof of a lemma

```
define other-step-1 :=
  (forall T0 s0 id0 ls0 step .
    config T0 = s0 ++ (actor id0 ls0) &
    Enabled (T0 then step) &
    focus step =/= id0
    ==> exists s . config (T0 then step) =
                   s ++ (actor id0 ls0))
```

that is used in the proof of `actor-persistence` for the case in which the *focus* of a transition step—i.e., the actor doing the sending, receiving, or creating—is not the same as `id0`, the identifier that is singled out in the configuration. This proof actually serves merely to select and apply one of three other lemmas (not shown) that are each specific to one type of transition step. This is accomplished by defining an appropriately parametrized sub-method and then setting up a call to it via **`datatype-cases`**, illustrating another feature of Athena's programming-based approach to proof: one doesn't always have to introduce a lemma explicitly as a sentence of the logic, associate a proof with it, and carry out the proof by applying the lemma (with, say, implication chaining). As is illustrated here, it is sometimes more convenient just to define a parametrized proof method and call it directly.

```
define actor-persistence-proof :=
 method (theorem adapt)
  let {given := lambda (P) (get-property P adapt Transition-Path.Theory);
       lemma := method (P) (!property P adapt Transition-Path.Theory);
       chain := method (L) (!chain-help given L 'none);
       chain-> := method (L) (!chain-help given L 'last);
       ++A := (given ['++ Associative]);
       ++C := (given ['++ Commutative]);
       R*L := (!lemma ['-->> R*-lemma]);
       [LS0 next ready-to new-ls unique-ids before after config] :=
         (adapt [LS0 next ready-to new-ls unique-ids before after config]);
       [T-sort _ s-sort id-sort ls-sort] := (map sort-of (qvars-of (adapt theorem)))}
  by-induction (adapt theorem) {
    (bind I Initial:T-sort) =>
      pick-any T0:T-sort s0:s-sort id0:id-sort ls0:ls-sort
        let {A1 := (config T0 = s0 ++ (actor id0 ls0));
             A2 := (T0 -->>* I)}
        assume (A1 & A2)
          let {goal := (exists ?s ?ls . config I = ?s ++ (actor id0 ?ls));
               B1 := (!chain-> [A2 ==> (T0 = I | T0 -->>+ I)  [R*L]])}
           (!cases B1
             assume B1a := (T0 = I)
               (!chain-> [A1 ==> (config I = s0 ++ (actor id0 ls0))  [B1a]
                             ==> goal                            [existence]])
             assume B1b := (T0 -->>+ I)
               (!from-complements goal B1b
                  (!chain-> [true ==> (~ B1b) [nothing-leads-to-Initial]])))
  | (T:T-sort then step) =>
    let {ind-hyp := (forall ?T0 ?s0 ?id ?ls0 .
                       config ?T0 = ?s0 ++ (actor ?id ?ls0) &
                       ?T0 -->>* T
                       ==> exists ?s ?ls . config T = ?s ++ (actor ?id ?ls))}
    pick-any T0:T-sort s0:s-sort id0:id-sort ls0:ls-sort
      let {A1 := (config T0 = s0 ++ (actor id0 ls0));
           A2 := (T0 -->>* (T then step))}
      assume (A1 & A2)
        let {B1 := (!chain->
                     [A2 ==> (T0 = (T then step) | T0 -->>+ (T then step))  [R*L]])}
        (!cases B1
          assume B1a := (T0 = (T then step))
            (!chain-> [A1 ==> (config (T then step) = s0 ++ (actor id0 ls0))  [B1a]
                          ==> (exists ?s ?ls .
                                config (T then step) = ?s ++ (actor id0 ?ls)) [existence]])
          assume B1b := (T0 -->>+ (T then step))
            let {goal := (exists ?s:(Cfg (Actor 'Id 'LS)) ?ls:'LS .
                            config (T then step) = ?s ++ (actor id0 ?ls));
                 LT := (!lemma leads-to);
                 C := (!chain->
                        [B1b ==> (T0 -->>* T & Enabled (T then step)) [LT]
                         ==> (T0 -->>* T)                             [left-and]
                         ==> (A1 & T0 -->>* T)                        [augment]
                         ==> (exists ?s1 ?ls1 .
                               config T = ?s1 ++ (actor id0 ?ls1))    [ind-hyp]])}
            pick-witnesses s1 ls1 for C C-witnessed
              let {D1 := (!chain->
                           [B1b ==> (T0 -->>* T & Enabled (T then step))  [LT]
                            ==> (Enabled (T then step))          [right-and]]);
                   D2 := (!chain->
                           [D1 ==> (Enabled T &
                                     exists ?s ?ls . (before T step ?s ?ls))   [enabled-step]])}
              pick-witnesses s ls for (!right-and D2) D2-witnessed
                let {E1 := (!chain-> [D2-witnessed ==> (after T step s ls) [transition-step]])}
                (!two-cases
                 assume F1 := (focus step = id0)
                   datatype-cases goal on step {
                     (receive id:id-sort fr:id-sort c) =>
                      let {G1 := (!chain [id = (focus (receive id fr c))   [focus-definition]
                                            = id0           [(step = (receive id fr c)) F1]])}
                      (!chain->
                       [E1 ==> (after T (receive id fr c) s ls)  [(step = (receive id fr c))]
                        ==> (config (T then (receive id fr c))
                             = s ++ (actor id (next ls (receive id fr c))))    [after.receiving]
```

```
                              = (s ++ (actor id0 (next ls (receive id fr c))))   [G1]
                         ==> (exists ?s ?ls .
                               config (T then (receive id fr c)) = ?s ++ (actor id0 ?ls))
                                                                                [existence]])
                    | (send id:id-sort to:id-sort c) =>
                       let {M := (One (message' id LS0 to c));
                            G1 := (!chain [id = (focus (send id to c)) [focus-definition]
                                              = id0          [(step = (send id to c)) F1]])}
                       (!chain->
                        [E1 ==> (after T (send id to c) s ls)   [(step = (send id to c))]
                         ==> (config (T then (send id to c))
                              = s ++ (actor id (next ls (send id to c))) ++ M)      [after.sending]
                              = (s ++ (actor id0 (next ls (send id to c))) ++ M)   [G1]
                              = ((s ++ M) ++ (actor id0 (next ls (send id to c)))) [++A ++C]
                         ==> (exists ?s ?ls .
                              config (T then (send id to c)) = ?s ++ (actor id0 ?ls)) [existence]])
                    | (create id:id-sort id':id-sort) =>
                       let {G1 := (!chain [id = (focus (create id id'))   [focus-definition]
                                              = id0            [(step = (create id id')) F1]])}
                       (!chain->
                        [E1 ==> (after T (create id id') s ls)  [(step = (create id id'))]
                         ==> (config (T then (create id id'))
                              = s ++ (actor id (next ls (create id id')))
                                ++ (actor id' (new-ls ls)))     [after.creating]
                              = (s ++ (actor id0 (next ls (create id id')))
                                 ++ (actor id' (new-ls ls)))    [G1]
                              = ((s ++ (actor id' (new-ls ls))) ++
                                 (actor id0 (next ls (create id id'))))   [++A ++C]
                         ==> (exists ?s ?ls .
                              config (T then (create id id')) = ?s ++ (actor id0 ?ls)) [existence]])
                    } # datatype-cases
                  assume (focus step =/= id0)
                    let {OS1 := (!lemma other-step-1);
                         G1 := (!chain-> [(C-witnessed & D1 & focus step =/= id0)
                                          ==> (exists ?s2 . config (T then step) =
                                                            ?s2 ++ (actor id0 ls1)) [OS1]])}
                    pick-witness s2 for G1 G1-w
                      (!chain-> [G1-w ==> goal [existence]])))
   } # by-induction

define other-step-1-proof :=
  method (theorem adapt)
   let {given := lambda (P) (get-property P adapt Transition-Path.Theory);
        lemma := method (P) (!property P adapt Transition-Path.Theory);
        chain-> := method (L) (!chain-help given L 'last);
        config := (adapt config);
        [T-sort s-sort id-sort ls-sort step-sort] := (map sort-of (qvars-of (adapt theorem)))}
   pick-any T0:T-sort s0:s-sort id0:id-sort ls0:ls-sort step:step-sort
     let {A1 := (config T0 = s0 ++ (actor id0 ls0));
          A2 := (Enabled (T0 then step));
          A3 := (focus step =/= id0)}
     assume (A1 & A2 & A3)
       let {proof :=
             method (a-step id)
               let {B1 := (!chain-> [A3 ==> (focus a-step =/= id0) [(step = a-step)]
                                        ==> (id =/= id0)           [focus-definition]]);
                    L1 := (!lemma match a-step {
                                    (receive _ _ _) => other-receive-1
                                  | (send _ _ _) => other-send-1
                                  | (create _ _) => other-create-1
                                  })}
               (!chain->
                [A2 ==> (Enabled (T0 then a-step))           [(step = a-step)]
                    ==> (A1 & Enabled (T0 then a-step) & B1) [augment]
                    ==> (exists ?s .
                          config (T0 then a-step) = ?s ++ (actor id0 ls0)) [L1]])}
       datatype-cases (exists ?s . config (T0 then step) = ?s ++ (actor id0 ls0))
                      on step {
         (receive id fr c) => (!proof (receive id fr c) id)
       | (send id to c) => (!proof (send id to c) id)
       | (create id id') => (!proof (create id id') id)
       }
```