Human-Readable Machine-Checkable Abstract Reasoning about Actor Systems

David Musser and Carlos A. Varela

Rensselaer Polytechnic Institute
Computer Science Department Technical Report
{musser,cvarela}@cs.rpi.edu

Abstract. The actor model of distributed computing imposes important restrictions on valid concurrent computations including fairness. We show that many properties of the model can be expressed and proved at an abstract level, independently of the details of a particular system of actors, in a logical framework in which proofs are both human-readable and machine-checkable. The framework, Athena, is briefly overviewed, with emphasis on its support for abstraction and specialization. A key contribution is the conceptual organization we develop: a richly structured hierarchy of formal theories that can be used to represent and reason about actor systems. Within this conceptual framework, we identify and prove a number of useful abstract-level theorems, including persistence of actors, preservation of unique actor identifiers, and general consequences of fairness. We also combine the general actor theory with a concrete ticker and clock actor system to prove several system-specific properties, including a progress theorem that depends on fairness.

1 Introduction and motivation

The actor model [9, 1] is useful both as a theoretical framework for reasoning about concurrent computation [2, 12] and as a practical paradigm for building distributed systems [5, 13]. 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 [7]. Actors have unique names 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, and/or send messages to other actors.

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 between actor configurations specify the possible ways in which an actor computation may evolve. The actor model imposes *fairness* on valid computation sequences. 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) would no longer hold when composed with another system (e.g., a denial of service attack), since the computation may evolve as a sequence of transitions that consistently ignores the web server actors.

Actor languages can use different models for representing sequential computation within an actor. Agha, Mason, Smith, and Talcott (AMST) use the untyped call-by-value lambda calculus to represent an actor's internal state and its behavior [2], whereas Varela and Agha use an object's instance and class to represent an actor's state and its behavior [13]. 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, 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 *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 demonstrate that abstract reasoning about actor systems, including issues of fairness and other key properties such as actor persistence and preservation of uniqueness of actor identities, can be carried out in a formalism that is both human-readable and machine-checkable. The logical framework that makes this possible, Athena [4, 3], is briefly described in Section 2, with emphasis on its support for abstraction and specialization. In Section 3 we begin the development

of the actor model at an abstract level, continuing in Section 4 with a general theory of fairness. 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 5, we return to the ticker-clock example and apply a combination of the general actor theory and the specifics of the example to prove several properties of the system, including a progress theorem that depends on fairness. Section 6 discusses related work and concludes with thoughts on future extensions of this work. Because of their length, most proofs are omitted from the main text but appear in full in appendices .

2 Athena

Athena is interactive and programmable, with separate but intertwined languages for conventional programming and "proof programming." 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, multisorted) 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.[1]

[1] Developing human-readable proofs has advantages and disadvantages in comparison with most approaches to mechanized theorem proving, which tend to be "black-box" to one degree or another. A resolution-based prover accepts just a set of axioms and the negation of the sentence to be proved and tries to find an inconsistency in a vast search space, which may take excessive time whether it succeeds or not. If it does succeed, even if the time required is minutes or hours, the human who posed the problem is relieved of having to work out the proof manually—a process which, after all, would probably take much longer, if it succeeded at all, and would not carry the same assurance that none of the thousands of details involved was mishandled. One the other hand, the proof found by a resolution prover 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 and as expressions of computation, as with Coq [10], HOL [8], Isabelle [11], 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, which does provide a much more readable user-level language, but without the same kinds of programming aids that Athena provides, and not entirely capable of suppressing the awkwardness, from a programmability standpoint, of the underlying sequent-based tactics and tacticals. Note that in Athena the two approaches can be freely mixed: the high-level skeleton of the proof (the important ideas) can be expressed in the language's

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 Appendix A. Athena facilities for *abstraction and specialization* are described next, since they are crucial to our abstract-level approach to actor system proofs.

In Athena, one can introduce axioms and theorems at an abstract level via *structured theories* [14], as explained below. Proofs are encapsulated in parameterized 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 in the development to be presented. Appendix B presents the relevant algebraic theories. The relational theories upon which we build actor theories are developed as successive *structured theory refinements*; see Figure 2.

In these definitions we use Athena's `theory` procedure. Specifically,

(`theory` *superiors axioms theory-name*)

defines a structured theory with name *theory-name*, (new) *axioms*, and *superiors*—theories of which the new theory is a direct refinement. The set of axioms of the theory so defined is the union of new axioms and, recursively, the sets of axioms of the superiors. In IRREFLEXIVE there is a single axiom, also named IRREFLEXIVE, and one superior, BINARY-RELATION`.Theory`, which has no axioms. STRICT-PARTIAL-ORDER refines IRREFLEXIVE`.Theory` and TRANSITIVE.`Theory`, and introduces no additional axiom.

The definition of TRANSITIVE-CLOSURE`.Theory` refines an *adapted* theory,

[STRICT-PARTIAL-ORDER`.Theory 'TC [R R+]]`,

which has the same axioms as STRICT-PARTIAL-ORDER but with `R` renamed as `R+`. These adapted axioms can be referred to by compound names of the form `['TC` *s*`]`, where *s* is a STRICT-PARTIAL-ORDER.`Theory` sentence. For example, from the TRANSITIVE axiom in STRICT-PARTIAL-ORDER.`Theory`,

readable proof format, while lower-level details may be outsourced to ATPs. In this work we have not made any use of external ATPs, although we have used some defined Athena methods (such as implicational chaining) that perform certain limited forms of proof search.

```
module Binary-Relation {
 declare R: (T) [T T] -> Boolean [100]
 define Theory := (theory [] [] 'Binary-Relation)}
module Irreflexive {
 open-module Binary-Relation
 define Irreflexive := (forall ?x . ~ ?x R ?x)
 define Theory := (theory [Binary-Relation.Theory]
                   [Irreflexive] 'Irreflexive)}
module Transitive {
 open-module 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-module Irreflexive
open-module Transitive
 define Theory :=
  (theory [Irreflexive.Theory Transitive.Theory]
    [] 'Strict-Partial-Order)}
module Transitive-Closure {
 open-module Irreflexive
 open-module Strict-Partial-Order
 declare R+, R*: (S) [S S] -> Boolean [100]
 declare R**: (S) [N S S] -> 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)
```

Fig. 1. Structured theory definitions. In symbol 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.

```
(forall ?x ?y ?z . ?x R ?y & ?y R ?z ==> ?x R ?z)),
```

we obtain ['TC TRANSITIVE] as the name of the axiom

```
(forall ?x ?y ?z . ?x R+ ?y & ?y R+ ?z ==> ?x R+ ?z).
```

With TRANSITIVE-CLOSURE.Theory's other ancestor theory, IRREFLEXIVE. Theory, there is no renaming, so its axiom with the original R symbol is included. Thus TRANSITIVE-CLOSURE.Theory is defining axioms that relate the three operators R, R+ and R∗.

Although we regard the sentences listed in a theory as axioms, we do not assert them into the assumption base. Instead, if we have *proved* that a homomorphic image of each of these sentences is a theorem, then we will be able to use proof methods associated with the theory to prove new theorems, rather than having to write their proofs for every structure that models the theory.

Figure 2 illustrates the refinement relation between the algebraic and relational theories and the actor theory development to be presented.

3 An abstract actor framework

3.1 Configuration theory

We will define actor configurations as a refinement of a more abstract notion of configuration. We first define a polymorphic configuration datatype, CFG:

```
datatype (CFG T) := Null | (One T) | (++ (CFG T) (CFG T))
assert (structure-axioms "Cfg")
```

The ***assert*** enters into the assumption base a small standard set of axioms for the constructors of the type, such as (forall ?x . Null =/= (One ?x)).

Next we define a configuration theory, CFG. (Module, datatype, and theory names need not be disjoint.)

```
module CFG {
open-module ABELIAN-MONOID
declare in: (T) [T (CFG T)] -> Boolean
define Empty := (forall ?a . ~ ?a in Null)
define Self := (forall ?a ?b . ?a in (One ?b) <==> ?a = ?b)
define Nonempty :=
  (forall ?a ?s1 ?s2 . ?a in (?s1 ++ ?s2) <==>
                       ?a in ?s1 | ?a in ?s2)
define Theory :=
  (theory [[ABELIAN-MONOID.Theory '++ [+ ++  <0> Null]]]
     [Empty Self Nonempty] 'Cfg)
```

Thus, the axioms of CFG theory include those of ABELIAN-MONOID,[2] but adapted to use the CFG datatype's ++ and Null constructors in place of + and <0>, respectively. Additionally, there are axioms in the new theory defining an in (i.e., membership) predicate.

[2] See Appendix B for our formulation of this theory by means of theory refinements.

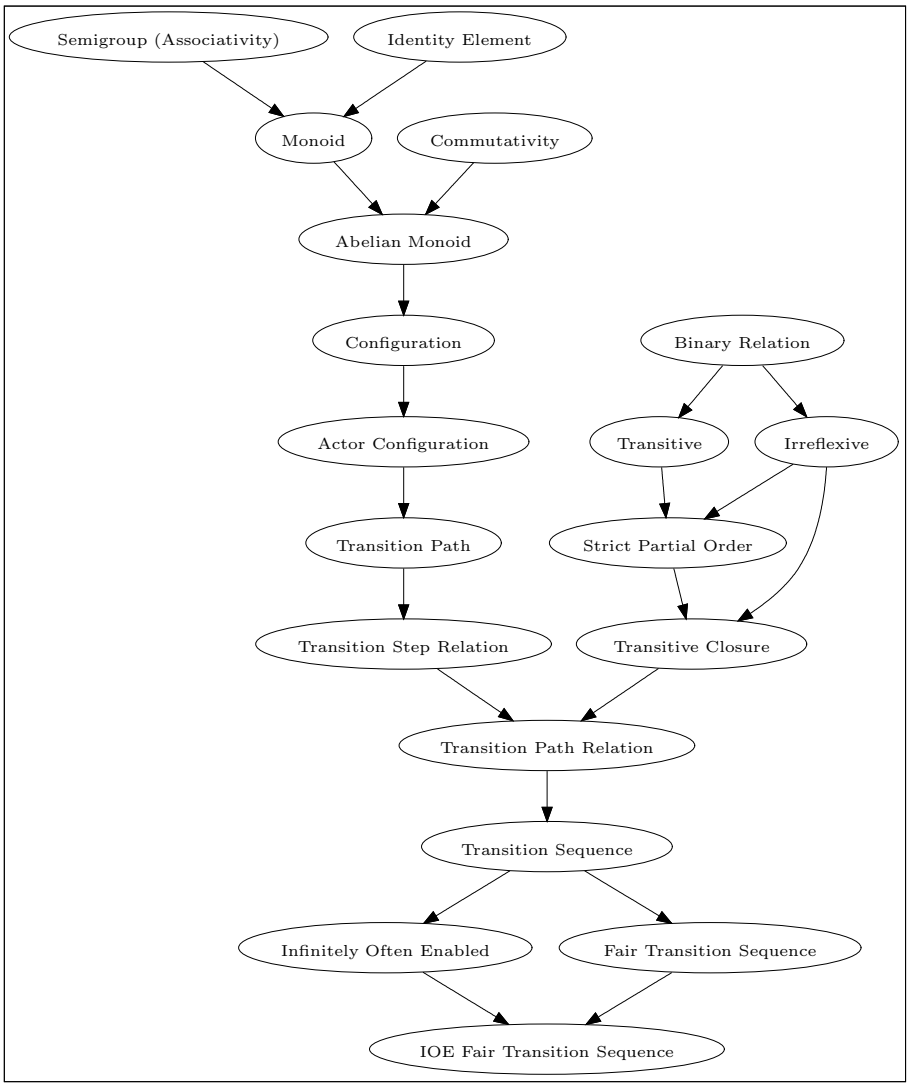

Fig. 2. Algebraic, Relational, and Actor Theories

Informally, we can think of a configuration as a sort of "soup" of components, with the ++ operator being the glue that holds the soup together, and the associative/commutative properties reflecting the fact that components can be arbitrarily reordered within the soup [6].

The following theorems relating in to the One and ++ configuration constructors are useful as lemmas in later proofs:

```
define Isolate1 := (forall ?s ?a . ?a in ?s ==>
                      (exists ?s1 . ?s = ?s1 ++ (One ?a)))
define Isolate2 :=
  (forall ?s ?a .
    (exists ?s1 . ?s = ?s1 ++ (One ?a)) ==> ?a in ?s)
define Together :=
  (forall ?s ?s1 ?s2 ?a ?b .
    ?s = ?s1 ++ (One ?a) &
    ?s = ?s2 ++ (One ?b) &
    ?a =/= ?b
    ==> (exists ?s3 . ?s = ?s3 ++ (One ?a) ++ (One ?b)))}
```

The proof of each of these lemmas is by induction on configurations, with one basis case (for the Null constructor) and two induction step cases (One and ++). For the ++ case, note that there are two induction hypotheses available, corresponding to its left and right arguments: see Figure 3.

3.2 An actor configuration theory

We now define Actor-Cfg as a refinement of Cfg theory. Additional axioms for this theory define a unique-ids predicate that can be used to require that no two actors in a configuration have the same identifier.

```
module Actor-Cfg {
open-module Cfg
datatype (Actor Id State) :=
  (actor' Id State) | (message' Id Id Ide)
assert (structure-axioms "Actor-Cfg.Actor")
define actor := lambda (id ls) (One (actor' id ls))
define message := lambda (fr to c) (One (message' fr to c))
declare unique-ids: (T) [(Cfg T)] -> Boolean
define uids-empty := (unique-ids Null)
define uids-nonempty :=
  (forall ?s ?id ?ls .
    (unique-ids ?s ++ (actor ?id ?ls)) <==>
    ~ (exists ?s' ?ls' . ?s = ?s' ++ (actor ?id ?ls')))
define uids-ignore-messages :=
  (forall ?s ?fr ?to ?c .
    (unique-ids ?s ++ (message ?fr ?to ?c)) <==>
    (unique-ids ?s))
define Theory :=
  (theory [Cfg.Theory]
    [uids-empty uids-nonempty uids-ignore-messages]
    'Actor-Cfg)
```

```
| (s' ++ s'') =>
  let {ind-hyp1 :=
        (forall ?a .
          ?a in s' ==> (exists ?s1 . s' = ?s1 ++ (One ?a)));
       ind-hyp2 :=
        (forall ?a .
          ?a in s'' ==> (exists ?s1 . s'' = ?s1 ++ (One ?a)))}
  pick-any a
    assume A := (a in (s' ++ s''))
      let {B := (!chain-last
                  [A ==> (a in s' | a in s'') [Nonempty]])}
      (!cases B
       assume (a in s')
         let {B1 := (!chain-last
                      [(a in s') ==>
                       (exists ?s1 . s' = ?s1 ++ (One a))
                         [ind-hyp1]])}
         pick-witness s1 for B1 B1-witnessed
           (!chain-last
            [(s' ++ s'')
             = ((s1 ++ (One a)) ++ s'') [B1-witnessed]
             = ((s1 ++ s'') ++ (One a)) [++A ++C]
             ==> (exists ?s1 .
                    s' ++ s'' =
                   ?s1 ++ (One a))      [existence]])
       assume (a in s'')
         let {B2 := (!chain-last
                      [(a in s'')
                       ==> (exists ?s2 .
                              s'' = ?s2 ++ (One a))
                                    [ind-hyp2]])}
         pick-witness s2 for B2 B2-witnessed
           (!chain-last
            [(s' ++ s'')
             = (s' ++ (s2 ++ (One a))) [B2-witnessed]
             = ((s' ++ s2) ++ (One a)) [++A]
             ==> (exists ?s1 .
                    s' ++ s'' = ?s1 ++ (One a))
                                    [existence]]))}
```

Fig. 3. A portion of the proof of `Isolate1`. `++A` and `++C` are defined as the associative and commutative axioms for `++`. This proof is representative of many of the actor systems proofs in its use of various inference methods, including induction, case analysis, and equality and implication chaining. The full proofs of the actor isolation lemmas are given in Appendix D.1.

In the `Actor` datatype, `Id` and `State` are type parameters, while `Ide` refers to a built-in identifier type. Appendix D.2 presents and proves some useful theorems about consequences of assuming the unique-ids predicate holds.

3.3 Transition path datatype and theory

We model dynamic changes to actor configurations in terms of *transition paths*. Syntactically, transition paths are defined by the following datatype.

```
datatype (TP Id State) := Initial
| (receive (TP Id State) Id State Id Ide)
| (send (TP Id State) Id Id Ide)
| (create (TP Id State) Id Id State)
| (compute (TP Id State) Id State State)
assert (datatype-axioms "TP")
```

For the semantics of transition paths, we define TRANSITION-PATH theory:

```
module TRANSITION-PATH {
open-module ACTOR-CFG
declare config: (Id, State)
  [(TP Id State)] -> (CFG (Actor Id State))
declare accept: (Id, State, Sender)
  [Id State Sender Ide] -> State
declare make-receptive: (State) [State] -> State
declare ready-to: (Id, State) [(TP Id State)] -> Boolean
define trans-receive :=
 (forall ?T ?s ?id ?ls ?fr ?to ?c .
   (config ?T) = ?s ++ (actor ?id ?ls) ++
                 (message ?fr ?id ?c) &
   (ready-to (receive ?T ?id ?ls ?fr ?c))
   ==> (config (receive ?T ?id ?ls ?fr ?c)) =
         ?s ++ (actor ?id (accept ?id ?ls ?fr ?c)))}
```

Additional axioms for the send, create, and compute transitions are given in Appendix D.3, as are declarations and axioms for an Enabled predicate on transition paths and actors states that characterizes when a transition can be taken. In terms of Enabled, we define a refinement of TRANSITION-PATH`.Theory` named TRANSITION-STEP-RELATION`.Theory`. The basic relation defined is `-->>`, with axioms such as

```
define directly-leads-to-receive :=
  (forall ?T0 ?T ?id ?ls ?fr ?c .
    ?T0 -->> (receive ?T ?id ?ls ?fr ?c)
    ==> ?T0 = ?T & (Enabled (receive ?T0 ?id ?ls ?fr ?c)))
```

The theory thus defined is then combined with TRANSITIVE-CLOSURE`.Theory` to derive a TRANSITION-PATH-RELATION theory that defines the irreflexive and reflexive transitive closures, `-->>+` and `-->>*`, of `-->>` (see Appendix D.4).

3.4 Actor persistence and unique-ids persistence

The following theorems show that as transition paths are traversed, existing actors persist, and so does uniqueness of actor id's.

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

The proof of these theorems (see Appendix D.5) is by induction on transition paths. For each of the basis case and the receive, send, create, and compute induction-step cases, the proof calls on many of the axioms and theorems of superior theories.

4 Actor system fairness

We express fairness properties of actor systems in terms of *transition sequences*, which use natural numbers to label subpaths from one point in a transition path to another in the same path. This labeling is effected by a *transition sequence function*, ts.

```
module Transition-Sequence {
open-module Transition-Path-Relation
open-module N
declare ts: (Id, State) [(TP Id State) N] -> (TP Id State)
define ts-initial := (forall ?T . (ts ?T zero) = ?T)
define ts-directly-connected :=
  (forall ?T ?n . (ts ?T ?n) -->> (ts ?T (S ?n)))
define Theory :=
  (theory [Transition-Path-Relation.Theory]
    [ts-initial ts-directly-connected]
    'Transition-Sequence)}
```

We first axiomatize *infinitely-often-enabled* transitions and *fairness* in separate refinements of Transition-Sequence theory.

```
module Infinitely-Often-Enabled {
open-module Transition-Sequence
declare receiver, sender, creator, computer: (Id) [] -> Id
define ioe-receive :=
 (forall ?T ?n ?s ?ls ?fr ?to ?c .
  (config (ts ?T ?n)) = ?s ++ (actor receiver ?ls) ++
                        (message ?fr receiver ?c) &
  (ready-to (receive (ts ?T ?n) receiver ?ls ?fr ?c))
```

```
  ==>
  (ts ?T (S ?n)) = (receive (ts ?T ?n) receiver ?ls ?fr ?c)
  | (exists ?k ?s' ?ls' .
      ?k > ?n & (config (ts ?T ?k)) =
                  ?s' ++ (actor receiver ?ls') ++
                  (message ?fr receiver ?c) &
      (ready-to (receive (ts ?T ?k) receiver ?ls' ?fr ?c))))}
```

`ioe-receive` states that if a `receive` transition is enabled at transition n in an actor transition sequence, then either the `receive` is the next transition in the sequence, or there is a future point in the sequence where `receive` is enabled again. This property is specific to a concrete actor system and must be proved for each concrete system (see Appendix E.1 for the proof for the ticker-clock system). For the remaining IOE axioms, see Appendix D.6.

```
module Fair-Transition-Sequence {
open-module Transition-Sequence
define fair-receive :=
  (forall ?id ?T ?n ?s ?ls ?fr ?to ?c .
    (config (ts ?T ?n)) = ?s ++ (actor ?id ?ls) ++
                          (message ?fr ?id ?c) &
    (ready-to (receive (ts ?T ?n) ?id ?ls ?fr ?c))
    ==>
    (exists ?k ?s' ?ls' .
      ?k >= ?n &
      (config (ts ?T ?k)) = ?s' ++ (actor ?id ?ls') ++
                            (message ?fr ?to ?c) &
      (ready-to (receive (ts ?T ?k) ?id ?ls' ?fr ?c)) &
      (ts ?T (S ?k)) = (receive (ts ?T ?k) ?id ?ls' ?fr ?c))
    | ~ (exists ?k ?s' ?ls' .
          ?k > ?n &
          (config (ts ?T ?k)) =
            ?s' ++ (actor ?id ?ls') ++ (message ?fr ?id ?c) &
          (ready-to (receive (ts ?T ?k) ?id ?ls' ?fr ?c))))}
```

`fair-receive` states that if a `receive` transition is enabled at transition n in an actor transition sequence, then either the `receive` eventually happens or it becomes permanently disabled. We have used this definition of fairness from Agha et al. [2]. Since fairness is a requirement of the actor model, *i.e.*, valid actor implementations must be fair, we regard `fair-receive` as an axiom. Again, the remaining axioms are given in Appendix D.6.

Next, we define:

```
module IOE-Fair-Transition-Sequence {
open-module Infinitely-Often-Enabled
open-module Fair-Transition-Sequence
define Theory := (theory [Infinitely-Often-Enabled.Theory
                          Fair-Transition-Sequence.Theory]
                   [] 'IOE-Fair-Transition-Sequence)}
```

In this combined theory we can state and prove fairness theorems that are more convenient to use in progress theorem proofs than working from the axioms:

```
extend-module IOE-Fair-Transition-Sequence {
define fair-receive-theorem :=
 (forall ?T ?n ?s ?ls ?fr ?c .
  (config (ts ?T ?n)) = ?s ++ (actor receiver ?ls) ++
                        (message ?fr receiver ?c) &
  (ready-to (receive (ts ?T ?n) receiver ?ls ?fr ?c))
  ==>
  (exists ?m ?s' ?ls' .
   ?m >= ?n &
   (config (ts ?T ?m)) = ?s' ++ (actor receiver ?ls') ++
                         (message ?fr receiver ?c) &
   (ready-to (receive (ts ?T ?m) receiver ?ls' ?fr ?c)) &
   (ts ?T (S ?m)) =
     (receive (ts ?T ?m) receiver ?ls' ?fr ?c)))}
```

For corresponding theorems for send, create, and compute transitions, see Appendix D.7.

5 A concrete actor system example: ticker-clock

For the example of ticker and clock actors introduced in Section 1, we define:

```
module Clock-Actors {
open-module Transition-Path
domain Actor-Name
declare Ticker, Clock1: Actor-Name;
assert (Ticker =/= Clock1)
datatype CLS := empty | (clocal Actor-Name N)}
```

The empty and clocal constructors represent the local state of the Ticker and Clock1, respectively. These actors' behavior is defined by axioms, such as

```
assert accept :=
  (forall ?t .
    (accept Clock1 (clocal Clock1 ?t) Ticker 'tick) =
                   (clocal Clock1 (S ?t)))
```

which defines how Clock1 responds to a tick message. For the other axioms, see Appendix E.

We now define a refinement of IOE-Fair-Transition-Sequence that specializes the transition sequence function and sender and receiver constants to those of the clock system.

```
module Fair-Clock-System {
open-module IOE-Fair-Transition-Sequence
open-module Clock-Actors
declare cts: [(TP Actor-Name CLS) N] -> (TP Actor-Name CLS)
define CA := [ts cts  sender Ticker  receiver Clock1]
define Theory :=
  (theory [[IOE-Fair-Transition-Sequence.Theory 'Clock CA]]
    [] 'Fair-Clock-System)}
```

For fair transition sequences as defined by FAIR-CLOCK-SYSTEM, we have the following theorem:

```
define Clock1-progress :=
  (forall ?t ?T:(TP Actor-Name CLS) ?n0 ?s0 ?ls0 ?t0 .
    (config (cts ?T ?n0) =
       ?s0 ++ (actor Ticker ?ls0) ++
       (actor Clock1 (clocal Clock1 ?t0)) &
    (unique-ids (config (cts ?T ?n0)))
    ==> (exists ?n ?s ?ls ?u .
          ?n >= ?n0 &
          (config (cts ?T ?n)) =
            ?s ++ (actor Ticker ?ls) ++
            (actor Clock1 (clocal Clock1 ?u)) &
          ?u >= ?t)
```

This theorem states that for any arbitrarily large time t the clock will eventually hold a value $u \geq t$. It is proved by induction on t. In the inductive step, one has to show that eventually Ticker emits a `tick` message, and eventually Clock1 receives it and increments its internal counter. The details (see Appendix E.2) depend crucially on the `fair-send` and `fair-receive` theorems, the `actor-persistence` and `unique-ids-persistence` theorems, and, of course, the axiomatized behavior of Ticker and Clock1.

6 Related work and conclusions

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 [6]. Maude's high level language and powerful AC rewriting are the foundation of its system specification and model checking capabilities, but Athena's proof capabilities are more suitable for formulating the axioms, theorems, and proofs we have developed. Exploring fairness, for example, was done in [6] 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 testing specifications before attempting full proofs. To date, we have only explored such testing in a limited way; see Appendix F for a sample sequence of transitions exercising the clock system, beginning with the clock creating the ticker and continuing through several send and receive transitions.

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. Human-readability of proofs makes it possible to recognize repetitive patterns and develop methods that encapsulate them, so that existing proofs can be shortened and future proof development simplified. These methods can enable further formal reasoning about distributed system properties

such as information flow and fault tolerance.

Acknowledgment: The authors wish to thank Konstantine Arkoudas 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://people.csail.mit.edu/kostas/dpls/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. Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang, Second Edition*. Prentice-Hall, 1996.
6. 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.
7. T. Desell, K. E. Maghraoui, and C. A. Varela. Malleable applications for scalable high performance computing. *Cluster Computing*, pages 323–337, June 2007.
8. M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.
9. Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323–364, 1977.
10. The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.
11. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer, 1994. LNCS 828.
12. C. L. Talcott. Composable semantic models for actor theories. *Higher-Order and Symbolic Computation*, 11(3):281–343, 1998.
13. 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.
14. Aytekin Vargun and David Musser. Code carrying theory. In *SAC '08 Proceedings of the 2008 ACM symposium on applied computing*, pages 376–383.

Appendices

A Athena proof methods

Assumption bases (introduced in Section 2) play a fundamental role in Athena proofs. Anyone acquainted with the basics of how conventional programs are interpreted or compiled, manipulation of the assumption base during proofs will seem familiar, because it is similar to the way that the *run-time stack* is used to hold results of intermediate computations. As with the stack, sentences may be added to the assumption base temporarily and later removed. An important case is the construct

assume A
  D

during whose evaluation A is added to the assumption base and remains present while evaluating deduction D. If D succeeds in proving a theorem C then the new sentence added to the assumption base is $(A \Rightarrow C)$, and A is removed.[3] (A is also removed if D raises an error condition.) This is logically sound because A was added to the assumption base only for the purpose of proving $(A \Rightarrow C)$, and validity of that sentence itself does not depend on A.

A.1 Equality and implication chaining

One of the most ubiquitous proof methods is proving equations by chaining together a sequence of terms connected by equalities. In Athena, we can express such proofs with the `chain` method:

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

proves the equation $t_0 = t_n$, where each J_i is a *justification* for the preceding equality $t_{i-1} = t_i$. Each justification J_i must be a sentence—in this case a previously assumed or proved universally quantified equality or conditional equality—or a method capable of proving $t_{i-1} = t_i$.[4] Thus, each justification that is a sentence must already be in the assumption base, and each one that is a method must work with assumption base entries to prove the equality step.

One can also express *implication chains* with the `chain` method. To prove the implication $S_0 \Rightarrow S_n$, one can write a `chain` call

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

where the S_i are sentences and the justification J_i proves $(S_{i-1} \Rightarrow S_i)$. If S_0 has already been proved, the variant

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

proves S_n.

[3] For simplicity, we speak here of sentences being "added" or "removed" as if assumption bases were stateful objects subject to destructive modification. In reality, assumption bases are *functional*. Thus, for instance, the body of the `assume` deduction above is evaluated in a *new* assumption base, obtained by augmenting the outer assumption base with the hypothesis A.

[4] Actually, the justification can be a list of such sentences or methods which together justify the equality. Also, in place of a sentence or method one can write an arbitrary Athena expression that evaluates to it.

A.2 Induction

For natural numbers, ordinary mathematical induction takes the form of dividing a proof of $(\forall n.\ (P\ n))$ into two cases: (i) $(P\ 0)$ and (ii) $(\forall n.\ (P\ n) \Longrightarrow (P\ n+1))$. Case (i) is called the *Basis Case*, case (ii) is called the *Induction Step*, and within it $(P\ n)$ is called the *Induction Hypothesis*. Proof of the Basis Case and the Induction Step suffices, basically because every natural number is either 0 or can be constructed from 0 by a finite number of increments by 1. This property of natural numbers can be stated in Athena by defining natural numbers as the following datatype:

```
datatype N := zero | (S N)
```

The symbols `zero` and `S` ("successor") are called the *constructors* of `N`. Given this declaration, the only ground terms allowed by Athena's type system for `N` values are `zero`, `(S zero)`, `(S (S zero))`, etc. Furthermore, from this ***datatype*** declaration, Athena derives the usual natural number induction principle and makes it available via its ***by-induction*** form. Schematically,

```
by-induction (forall ?n . P ?n) {
  zero =>
    conclude (P zero)
      ...
| (S n) =>
    let {ind-hyp := (P n)}
      conclude (P (S n))
        ...
}
```

Here we have given a name `ind-hyp` to the induction hypothesis `(P n)`, but whether we name it or not, it is available in the induction-step case (i.e., ***by-induction*** places it in the assumption base) for use in the proof of $(P$ `(S n)`$)$.

More generally, Athena can extend ***by-induction*** to provide an induction principle based on the form of constructors as given in any ***datatype*** declaration. An example only slightly more complex than `N` is a *list* type defined by

```
datatype (List T) := nil | (:: T (List T))
```

This defines a *polymorphic* datatype: `T` is a sort parameter, and `(List T)` is the sort of homogeneous lists of sort `T` elements; e.g., `(List Boolean)`, `(List N)`, `(List (List N))`, etc., are separate sorts that share the `nil` and `::` constructors. A use of the corresponding induction principle might take the form:

```
by-induction (forall ?L ?x . (Q ?L ?x) {
  nil =>
    conclude (forall ?x . (Q nil ?x))
      ...
| (y :: L) =>
  let {ind-hyp := (forall ?x . (Q L ?x))}
    conclude (forall ?x . (Q (y :: L) ?x))
      ...
}
```

Note that the induction cases retain the universal quantification of variables other than the induction variable.

Proofs of many of the properties of actor systems proceed by induction on transitions that take an actor configuration to a new configuration. We obtain the appropriate induction principle by defining transitions as a ***datatype*** and using the corresponding form of ***by-induction*** for the proofs.

A.3 Case analysis

An induction proof is one important kind of case analysis, breaking a proof into one or more basis cases (corresponding to ***datatype*** constructors that take no arguments of the type) and one or more induction step cases (corresponding to constructors that take at least one argument of the type). More generally, case analysis can be applied independently of a datatype, when we have the same proof goal under each of several different assumptions whose disjunction is known to cover all possibilities. For example, assuming the disjunction

`(`P_1 `|` P_2 `| ... |` P_n `)`

is in the assumption base, if we write

```
(!cases (P1 | P2 | ... | Pn)
  assume P1
    D1
  assume P2
    D2
  ...
  assume Pn
    Dn)
```

and fill in each of the D_i deductions to conclude the same goal sentence Q, then Q is the result of the entire `cases` deduction.

An important special case is just two disjuncts with one the negation of the other. We can write

```
(!two-cases
  assume A
    D1
  assume (~ A)
    D2)
```

since it is equivalent to

```
(!cases (!ex-middle A)
  assume A
    D1
  assume (~ A)
    D2)
```

where `(!ex-middle` A`)` deduces `(`A `| ~` A`)` for any sentence A. (Athena's logic is classical.)

A.4 Proof by contradiction

In Athena, the main form of a proof by contradiction is

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

where D deduces `false`. Commonly, D obtains `false` with a call of the `absurd` method, of the form

```
(!absurd Q R)
```

where Q and R have been deduced from `(~ P)` and other members of the assumption base and R is the negation of Q.

Sometimes one does not need the negation of P to obtain the contradiction. Then one can deduce P more simply with the `from-complements` method:

```
(!from-complements P Q R)
```

where R is the negation of Q.

A.5 Proofs at an abstract level

To introduce the way that abstract-level proofs can be encapsulated in a parameterized method, let us state a couple of Transitive-Closure theorems, define a proof method that can prove any adapted version of the theorems, and show how the theorem statement and proof method can be incorporated into the Transitive-Closure theory.

```
extend-module TRANSITIVE-CLOSURE {
define RR+-inclusion :=  (forall ?x ?y . ?x R ?y ==> ?x R+ ?y)
define TC-Transitivity1 :=
  (forall ?x ?y ?z . ?x R+ ?y & ?y R ?z ==> ?x R+ ?z)
define theorems := [RR+-inclusion TC-Transitivity1]}
```

These are two of five Transitive-Closure theorems used in our actor theory development; see Appendix C for the full set of theorems.

```
extend-module TRANSITIVE-CLOSURE {
define proofs :=
  method (theorem adapt)
    let {given := lambda (P) (get-property P adapt Theory);
         lemma := method (P) (!property P adapt Theory);
         chain := method (L) (!chain-help given L 'none);
         chain-last := method (L) (!chain-help given L 'last);
         [R R+ R*] := (adapt [R R+ R*])}
    match theorem {
      (val-of RR+-inclusion) =>
      pick-any x y
        (!chain
         [(x R y)
          ==> (x R y | (exists ?z . x R+ ?z & ?y' R y))
```

```
                                    [alternate]
          ==> (x R+ y)              [R+-definition]])
    | (val-of TC-Transitivity1) =>
      pick-any x y z
        assume A := (x R+ y & y R z)
          (!chain-last
           [A ==> (exists ?y . x R+ ?y & ?y R z) [existence]
              ==> (x R z | (exists ?y . x R+ ?y & ?y R z))
                                                  [alternate]
              ==> (x R+ z)                  [R+-definition]])
    }
(evolve Theory [theorems proofs])}
```

The `evolve` procedure call extends the list of sentences associated with Transitive-Closure theory to include these theorems and also records the proof method containing the corresponding proofs. Having done so, we can at any time retrieve an instance one of the Transitive-Closure axioms or theorems with Athena's `get-property` procedure. In general, (`get-property` *P adapt theory*) searches the theory refinement hierarchy for *P*, using *theory* as the starting point for the search. The top-level axioms of *theory* are searched, followed by a recursive search of each of its superiors. If `get-property` finds *P* it applies the *adapt* mapping to it and returns the result. It is an error if the search ends without finding *P*, or if applying *adapt* results in an ill-typed sentence.

While `get-property` is just a procedure, there is a corresponding *method*—`property`, which takes the same arguments and conducts the same search and adaptation—that tries to prove the resulting sentence using the associated proof method.

The proof method introduces local, adapted versions of methods `lemma`, `chain` and `chain-last`, and a procedure, `given`, which are handy tools for simplifying the way that external properties can be cited in proof steps that need them. The `lemma` method simply calls the `property` method, passing it `P` (`lemma`'s argument), `adapt` (one of the enclosing proof method's arguments), and the theory to be used as the starting point for searches for `P`. Thus it finds the cited property in the theory structure and proves the instance produced by applying `adapt` to it, using the proof method that accompanies `P`.

The `given` procedure is similar but only retrieves the cited property; it is used in an abstract-level proof when the property is known either to be an axiom of the theory or to have already been proved using `lemma` (or `property` directly). Thus, in either case, the property is already in the assumption base.

Lastly, the local definition of `chain` (and, if necessary, `chain-last` and `chain-first`) is defined in terms of the predefined method `chain-help`, which tries to apply `given` to sentences in chain justifications (if that fails, it uses the sentence itself). This is what allows one to write chain-step justifications like `[R+-definition]` instead of the more verbose `[(given R+-definition)]` or `[(get-property R+-definition adapt` TRANSITIVE-CLOSURE`.Theory)]`.

B Algebraic theories

The algebraic theories we build on for defining actor configurations are the following:

```
module SEMIGROUP {
declare +: (T) [T T] -> T [200]
```

```
define Associative := (forall ?x ?y ?z .
                                (?x + ?y) + ?z = ?x + (?y + ?z))
define Theory := (theory [] [Associative] 'Semigroup)}
```

```
module IDENTITY {
open-module SEMIGROUP
declare <0>: (T) [] -> T
define Left-Identity := (forall ?x . <0> + ?x = ?x)
define Right-Identity := (forall ?x . ?x + <0> = ?x)
define Theory :=
  (theory [] [Left-Identity Right-Identity] 'Identity)}
```

```
module MONOID {
open-module IDENTITY
define Theory :=
  (theory [SEMIGROUP.Theory IDENTITY.Theory] [] 'Monoid)}
```

```
module ABELIAN-MONOID {
open-module MONOID
define Commutative := (forall ?x ?y . ?x + ?y = ?y + ?x)
define Theory :=
  (theory [MONOID.Theory] [Commutative] 'Abelian-Monoid)}
```

C Transitive-Closure theory

The full definition of Transitive-Closure:

```
module TRANSITIVE-CLOSURE {
open-module IRREFLEXIVE
open-module STRICT-PARTIAL-ORDER
declare R+, R*: (S) [S S] -> Boolean [100]
declare R**: (S) [N S S] -> Boolean
declare S0: (S) [] -> S
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)
define R**-sum :=
  (forall ?n ?m ?x ?y ?z .
    (R** ?m ?x ?y) & (R** ?n ?y ?z) ==> (R** (?m + ?n) ?x ?z))
define RR+-inclusion :=  (forall ?x ?y . ?x R ?y ==> ?x R+ ?y)
define R+R*-inclusion := (forall ?x ?y . ?x R+ ?y ==> ?x R* ?y)
define R+-lemma :=
  (forall ?x ?y .
     ?x R+ ?y <==> ?x R ?y |
     (exists ?y' . ?x R+ ?y' & ?y' R ?y))
define R*-lemma :=
  (forall ?x ?y . ?x R* ?y <==> ?x = ?y | ?x R+ ?y)
define TC-Transitivity :=
  (forall ?x ?y ?z . ?x R+ ?y & ?y R+ ?z  ==> ?x R+ ?z)
define TC-Transitivity1 :=
  (forall ?x ?y ?z . ?x R+ ?y & ?y R ?z ==> ?x R+ ?z)
define TC-Transitivity2 :=
  (forall ?x ?y ?z . ?x R ?y & ?y R ?z  ==> ?x R+ ?z)
define TC-Transitivity3 :=
  (forall ?x ?y ?z . ?x R* ?y & ?y R* ?z  ==> ?x R* ?z)
define theorems := [R**-sum TC-Transitivity RR+-inclusion R+R*-inclusion
                    R+-lemma R*-lemma TC-Transitivity1
                    TC-Transitivity2 TC-Transitivity3]
define proofs :=
  method (theorem adapt)
    let {given := lambda (P) (get-property P adapt Theory);
         lemma := method (P) (!property P adapt Theory);
         chain := method (L) (!chain-help given L  'none);
         chain-last := method (L) (!chain-help given L 'last);
         [R R+ R* R** S0] := (adapt [R R+ R* R** S0])}
    match theorem {
      (val-of R**-sum) =>
      by-induction theorem {
        zero =>
        pick-any m x y z
          let {A1 := (R** m x y);
               A2 := (R** zero y z)}
          assume (A1 & A2)
            let {B := (!chain-last
                       [A2 ==> (y = z)  [R**-zero]])}
            (!chain-last
             [A1 ==> (R** m x z)           [B]
                 ==> (R** (m + zero) x z) [N.Plus.right-zero]])
      | (S n) =>
        let {ind-hyp := (forall ?m ?x ?y ?z .
                                (R** ?m ?x ?y) & (R** n ?y ?z) ==>
                                (R** (?m + n) ?x ?z))}
        pick-any m x y z
          let {A1 := (R** m x y);
               A2 := (R** (S n) y z)}
```

```
        assume (A1 & A2)
          let {B := (!chain-last
                     [A2 ==> (exists ?y' . (R** n y ?y') & ?y' R z)
                         [R**-nonzero]])}
          pick-witness y' for B
            let {B-w1 := (R** n y y');
                 B-w2 := (y' R z)}
            (!chain-last
             [(A1 & B-w1)
              ==> (R** (m + n) x y')              [ind-hyp]
              ==> ((R** (m + n) x y') & B-w2)     [augment]
              ==> (exists ?y' . (R** (m + n) x ?y') & ?y' R z)
                                                  [existence]
              ==> (R** (S (m + n)) x z)           [R**-nonzero]
              ==> (R** (m + (S n)) x z)           [N.Plus.right-nonzero]])
    }
  | (val-of TC-Transitivity) =>
    pick-any x y z
      let {A1 := (x R+ y);
           A2 := (y R+ z)}
      assume (A1 & A2)
        let {B1 := (!chain-last
                    [A1 ==> (exists ?m . (R** (S ?m) x y))
                        [R+-definition]]);
             B2 := (!chain-last
                    [A2 ==> (exists ?n . (R** (S ?n) y z))
                        [R+-definition]]);
             _ := (!lemma R**-sum)}
        pick-witness m for B1 B1-w
        pick-witness n for B2 B2-w
          (!chain-last
           [(B1-w & B2-w)
            ==> (R** ((S m) + (S n)) x z)       [R**-sum]
            ==> (R** (S (m + (S n))) x z)       [N.Plus.left-nonzero]
            ==> (exists ?k . (R** (S ?k) x z)) [existence]
            ==> (x R+ z)                        [R+-definition]])
  | (val-of RR+-inclusion) =>
    pick-any x y
      (!chain
       [(x R y)
        ==> (x = x & x R y)                   [augment]
        ==> ((R** zero x x) & x R y)          [R**-zero]
        ==> (exists ?x' . (R** zero x ?x') & ?x' R y) [existence]
        ==> (R** (S zero) x y)                [R**-nonzero]
        ==> (exists ?k . (R** (S ?k) x y))    [existence]
        ==> (x R+ y)                          [R+-definition]])
  | (val-of R+-lemma) =>
    pick-any x y
     (!equiv
      assume A := (x R+ y)
```

```
    let {B :=
         (!chain-last
          [A ==> (exists ?k . (R** (S ?k) x y))    [R+-definition]])}
    pick-witness k for B B-w
      let {C := (!chain-last
                 [B-w ==> (exists ?x' . (R** k x ?x') & ?x' R y)
                      [R**-nonzero]])}
      pick-witness x' for C C-w
        (!two-cases
         assume D := (k = zero)
           let {E :=
                (!chain-last
                 [C-w ==> ((R** zero x x') & x' R y)    [D]
                      ==> (R** zero x x')               [left-and]
                      ==> (x = x')                      [R**-zero]])}
           (!chain-last
            [C-w ==> (x' R y)                            [right-and]
                 ==> (x R y)                             [(x = x')]
                 ==> (x R y | (exists ?y' . x R+ ?y' & ?y' R y))
                 [alternate]])
         assume D := (k =/= zero)
           let {E :=
                (!chain-last
                 [D ==> (exists ?k' . k = (S ?k'))  [N.nonzero-S]])}
           pick-witness k' for E E-w
             let {F :=
                  (!chain-last
                   [C-w ==> (x' R y)  [right-and]])}
             (!chain-last
              [C-w ==> ((R** (S k') x x') & x' R y) [E-w]
                   ==> (R** (S k') x x')            [left-and]
                   ==> (exists ?k' . (R** (S ?k') x x'))
                   [existence]
                   ==> (x R+ x')                    [R+-definition]
                   ==> (x R+ x' & F)                [augment]
                   ==> (exists ?x' . x R+ ?x' & ?x' R y) [existence]
                   ==> (x R y | (exists ?x' . x R+ ?x' & ?x' R y))
                   [alternate]]))
assume A := (x R y | (exists ?y' . x R+ ?y' & ?y' R y))
  let {RRI := (!lemma RR+-inclusion)}
  (!cases A
   (!chain [(x R y) ==> (x R+ y)  [RRI]])
   assume B := (exists ?y' . x R+ ?y' & ?y' R y)
     pick-witness y' for B B-w
       let {C :=
        (!chain-last
         [B-w ==> (x R+ y')       [left-and]
              ==> (exists ?k . (R** (S ?k) x y'))  [R+-definition]])}
       pick-witness k for C C-w
         (!chain-last
```

x

```
              [C-w ==> ((R** (S k) x y') & y' R y) [augment]
                   ==> (exists ?y' . (R** (S k) x ?y') & ?y' R y)
                   [existence]
                   ==> (R** (S (S k)) x y)          [R**-nonzero]
                   ==> (exists ?k . (R** (S ?k) x y))  [existence]
                   ==> (x R+ y)                     [R+-definition]])))
  | (val-of R*-lemma) =>
    pick-any x:(sort-of S0) y:(sort-of S0)
      (!equiv
        assume A := (x R* y)
          let {B := (!chain-last
                      [A ==> (exists ?n . (R** ?n x y))  [R*-definition]])}
          pick-witness n for B B-w
            (!two-cases
             assume C1 := (n = zero)
               (!chain-last
                [B-w ==> (R** zero x y)    [C1]
                     ==> (x = y)           [R**-zero]
                     ==> (x = y | x R+ y)  [alternate]])
             assume C2 := (n =/= zero)
               let {D := (!chain-last [C2 ==> (exists ?m . n = S ?m)
                                      [N.nonzero-S]])}
               pick-witness m for D D-w
                 (!chain-last
                  [B-w ==> (R** (S m) x y) [D-w]
                       ==> (exists ?m . (R** (S ?m) x y)) [existence]
                       ==> (x R+ y)         [R+-definition]
                       ==> (x = y | x R+ y) [alternate]]))
         assume A := (x = y | x R+ y)
           (!cases A
             assume A1 := (x = y)
               (!chain-last
                [A1 ==> (R** zero x y)              [R**-zero]
                    ==> (exists ?n . (R** ?n x y))  [existence]
                    ==> (x R* y)                    [R*-definition]])
             assume A2 := (x R+ y)
               let {B :=
                     (!chain-last
                      [A2 ==> (exists ?n . (R** (S ?n) x y))
                                                    [R+-definition]])}
               pick-witness n for B B-w
                 (!chain-last
                  [B-w ==> (exists ?k . (R** ?k x y)) [existence]
                       ==> (x R* y)                    [R*-definition]])))
   | (val-of R+R*-inclusion) =>
     let {R*L := (!lemma R*-lemma)}
     pick-any x y
       (!chain
        [(x R+ y)
         ==> (x = y | x R+ y)      [alternate]
```

```
      ==> (x R* y)                [R*L]])
| (val-of TC-Transitivity1) =>
  pick-any x y z
    let {A1 := (x R+ y);
         A2 := (y R z);
         R+L := (!lemma R+-lemma)}
    assume (A1 & A2)
      (!chain-last
       [(A1 & A2)
        ==> (exists ?y . x R+ ?y & ?y R z) [existence]
        ==> (x R z | (exists ?y . x R+ ?y & ?y R z))
                                           [alternate]
        ==> (x R+ z)                       [R+L]])
| (val-of TC-Transitivity2) =>
  pick-any x y z
    let {A1 := (x R y);
         A2 := (y R z);
         RR+I := (!lemma RR+-inclusion)}
    assume (A1 & A2)
      (!chain-last
       [A1 ==> (x R+ y)                [RR+I]
           ==> (x R+ y & A2)           [augment]
           ==> (x R+ y & y R+ z)       [RR+I]
           ==> (x R+ z)    [(given ['TC TRANSITIVE])]])
| (val-of TC-Transitivity3) =>
  pick-any x:(sort-of S0) y:(sort-of S0) z:(sort-of S0)
    let {A1 := (x R* y);
         A2 := (y R* z);
         RRI := (!lemma R+R*-inclusion);
         R*L := (!lemma R*-lemma)}
    assume (A1 & A2)
      let {B1 := (!chain-last
                  [A1 ==> (x = y | x R+ y) [R*L]]);
           B2 := (!chain-last
                  [A2 ==> (y = z | y R+ z) [R*L]])}
      (!cases B1
        assume C1 := (x = y)
          (!cases B2
           assume D1 := (y = z)
             (!chain-last
              [x = y [C1] = z [D1]
                 ==> (x = z | x R+ z) [alternate]
                 ==> (x R* z)         [R*L]])
           assume D2 := (y R+ z)
             (!chain-last
              [D2 ==> (x R+ z)     [C1]
                  ==> (x R* z)     [RRI]]))
        assume C2 := (x R+ y)
          (!cases B2
            assume D1 := (y = z)
```

```
                    (!chain-last
                     [C2 ==> (x R+ z)      [D1]
                         ==> (x R* z)      [RRI]])
                  assume D2 := (y R+ z)
                    (!chain-last
                     [D2 ==> (C2 & D2)     [augment]
                         ==> (x R+ z)      [TC-Transitivity]
                         ==> (x = z | x R+ z)   [alternate]
                         ==> (x R* z)      [R*L]])))

   }
(evolve Theory [theorems proofs])
}
```

D Additional theory and proofs

D.1 Proofs of configuration lemmas

In addition to the isolation lemmas and `Together`, we also have:

```
extend-module Cfg {
define More-Together :=
  (forall ?s ?s1 ?s2 ?a ?b ?c .
    ?s = ?s1 ++ (One ?a) &
    ?s = ?s2 ++ (One ?b) ++ (One ?c) &
    ?a =/= ?b & ?a =/= ?c
    ==> (exists ?s3 .
          ?s = ?s3 ++ (One ?a) ++ (One ?b) ++ (One ?c)))}
```

```
extend-module Cfg {
define proofs :=
 method (theorem adapt)
  let {given := lambda (P) (get-property P adapt Theory);
       lemma := method (P) (!property P adapt Theory);
       chain := method (L) (!chain-help given L 'none);
       chain-last := method (L) (!chain-help given L 'last);
       ++A := (given ['++ Associative]);
       ++C := (given ['++ Commutative]);
       left-Null := (given ['++ Left-Identity])}
   match theorem {
     (val-of Isolate1) =>
      by-induction theorem {
        Null =>
        pick-any a
         assume (a in Null)
          (!from-complements
           (exists ?s1 . Null = ?s1 ++ (One a))
           (a in Null)
```

```
      (!chain-last [true ==> (~ a in Null) [Empty]]))
| (One b) =>
  pick-any a
   assume (a in (One b))
    let {B := (!chain-last
               [(a in (One b))
                ==> (a = b)            [Self]])}
     (!chain-last
      [(One b)
       = (Null ++ (One b))          [left-Null]
       = (Null ++ (One a))          [B]
       ==> (exists ?s1 . (One b) = ?s1 ++ (One a))
                                    [existence]])
| (s' ++ s'') =>
  let {ind-hyp1 :=
        (forall ?a .
          ?a in s' ==> (exists ?s1 .
                          s' = ?s1 ++ (One ?a)));
       ind-hyp2 :=
        (forall ?a .
          ?a in s'' ==> (exists ?s1 .
                           s'' = ?s1 ++ (One ?a)))}
  pick-any a
    assume A := (a in (s' ++ s''))
     let {B := (!chain-last
                [A ==> (a in s' | a in s'') [Nonempty]])}
       (!cases B
         assume (a in s')
           let {B1 :=
                 (!chain-last
                  [(a in s') ==>
                     (exists ?s1 . s' = ?s1 ++ (One a))
                   [ind-hyp1]])}
           pick-witness s1 for B1 B1-witnessed
             (!chain-last
              [(s' ++ s'')
               = ((s1 ++ (One a)) ++ s'') [B1-witnessed]
               = ((s1 ++ s'') ++ (One a)) [++A ++C]
               ==> (exists ?s1 .
                      s' ++ s'' =
                      ?s1 ++ (One a))      [existence]])
         assume (a in s'')
           let {B2 :=
                 (!chain-last
                  [(a in s'')
                   ==> (exists ?s2 .
                          s'' = ?s2 ++ (One a))
                                           [ind-hyp2]])}
           pick-witness s2 for B2 B2-witnessed
             (!chain-last
```

```
                  [(s' ++ s'')
                   = (s' ++ (s2 ++ (One a))) [B2-witnessed]
                   = ((s' ++ s2) ++ (One a)) [++A ++C]
                   ==> (exists ?s1 .
                         s' ++ s'' = ?s1 ++ (One a))
                                          [existence]]))}
| (val-of Isolate2) =>
  pick-any s a
   assume A := (exists ?s1 . s = ?s1 ++ (One a))
    pick-witness s1 for A A-witnessed
     (!chain-last
      [(a = a)
       ==> (a in (One a))           [Self]
       ==> (a in s1 | a in (One a)) [alternate]
       ==> (a in (s1 ++ (One a)))   [Nonempty]
       ==> (a in s)                    [A-witnessed]])
| (val-of Together) =>
  pick-any s:(Cfg 'T) s1 s2 a b
   let {A1 := (s = s1 ++ (One a));
        A2 := (s = s2 ++ (One b));
        A3 := (a =/= b)}
    assume (A1 & A2 & A3)
     let {I2 := (!lemma Isolate2);
          B :=
            (!chain-last
             [A1 ==> (exists ?s1 . s = ?s1 ++ (One a))  [existence]
                 ==> (a in s)                   [I2]
                 ==> (a in (s2 ++ (One b)))     [A2]
                 ==> (a in s2 | a in (One b))  [Nonempty]]);
          goal := (exists ?s3 . s = ?s3 ++ (One a) ++ (One b))}
       (!cases B
         assume (a in s2)
           let {I1 := (!lemma Isolate1);
                C :=
                  (!chain-last
                   [(a in s2)
                    ==> (exists ?s3 . s2 = ?s3 ++ (One a)) [I1]])}
             pick-witness s3 for C C-witnessed
               (!chain-last
                [s = (s2 ++ (One b))                   [A2]
                   = ((s3 ++ (One a)) ++ (One b))  [C-witnessed]
                   = (s3 ++ (One a) ++ (One b))    [++A]
                   ==> goal  [existence]])
         assume D := (a in (One b))
           (!from-complements goal
            (!chain-last [D ==> (a = b)               [Self]])
            A3))
| (val-of More-Together) =>
  pick-any s:(Cfg 'T) s1 s2 a b c
   let {A1 := (s = s1 ++ (One a));
```

```
          A2 := (s = s2 ++ (One b) ++ (One c));
          A3 := (a =/= b);
          A4 := (a =/= c)}
       assume (A1 & A2 & A3 & A4)
        let {I2 := (!lemma Isolate2);
             B :=
               (!chain-last
                [A1 ==> (exists ?s1 . s = ?s1 ++ (One a)) [existence]
                    ==> (a in s)                          [I2]
                    ==> (a in (s2 ++ (One b) ++ (One c))) [A2]
                    ==> (a in s2 | a in ((One b) ++ (One c)))
                                                  [Nonempty]]);
             goal := (exists ?s3 .
                       s = ?s3 ++ (One a) ++ (One b) ++ (One c))}
          (!cases B
            assume (a in s2)
              let {I1 := (!lemma Isolate1);
                   C :=
                     (!chain-last
                      [(a in s2)
                       ==> (exists ?s3 . s2 = ?s3 ++ (One a)) [I1]])}
                pick-witness s3 for C C-witnessed
                  (!chain-last
                   [s = (s2 ++ (One b) ++ (One c))    [A2]
                      = ((s3 ++ (One a)) ++ (One b) ++ (One c))
                                                  [C-witnessed]
                      = (s3 ++ (One a) ++ (One b) ++ (One c)) [++A]
                      ==> goal  [existence]])
            assume D := (a in ((One b) ++ (One c)))
              let {E := (!chain-last
                         [D ==> (a in (One b) | a in (One c))
                                                [Nonempty]])}
                (!cases E
                   assume E1 := (a in (One b))
                     (!from-complements goal
                        (!chain-last [E1 ==> (a = b)  [Self]])
                        A3)
                   assume E2 := (a in (One c))
                     (!from-complements goal
                        (!chain-last [E2 ==> (a = c)  [Self]])
                        A4)))}}
```

D.2 Theorems about actor configurations

```
extend-module Cfg {
define unique-ids1 :=
  (forall ?s ?s1 ?s2 ?id1 ?ls1 ?id2 ?ls2 .
    (unique-ids ?s) &
```

```
    ?s = ?s1 ++ (actor ?id1 ?ls1) &
    ?s = ?s2 ++ (actor ?id2 ?ls2) &
    ?id1 = ?id2
    ==> ?ls1 = ?ls2)

define unique-ids2 :=
  (forall ?id1 ?ls1 ?id2 ?ls2 .
    ?id1 =/= ?id2 ==> (actor' ?id1 ?ls1) =/= (actor' ?id2 ?ls2))

define proofs :=
  method (theorem adapt)
    let {given := lambda (P)
                    (get-property P adapt Theory);
         lemma := method (P)
                    (!property P adapt Theory);
         chain := method (L) (!chain-help given L 'none);
         chain-last := method (L) (!chain-help given L 'last);
         ++A := (given ['++ Associative]);
         ++C := (given ['++ Commutative])}
      match theorem {
        (val-of unique-ids1) =>
         pick-any s:(Cfg (Actor 'Id 'LS)) s1 s2 id1 ls1 id2 ls2
           let {A1 := (unique-ids s);
                A2 := (s = s1 ++ (actor id1 ls1));
                A3 := (s = s2 ++ (actor id2 ls2));
                A4 := (id1 = id2)}
             assume (A1 & A2 & A3 & A4)
              (!by-contradiction (ls1 = ls2)
               assume (ls1 =/= ls2)
                let {CI := (!constructor-injectivity actor');
                     B1 :=
                      (!by-contradiction
                           ((actor' id1 ls1) =/= (actor' id2 ls2))
                        assume C1 := ((actor' id1 ls1) =
                                      (actor' id2 ls2))
                          (!absurd
                           (!chain-last
                            [C1 ==> (id1 = id2 & ls1 = ls2) [CI]
                                ==> (ls1 = ls2)    [right-and]])
                           (ls1 =/= ls2)));
                     T := (!lemma Together);
                     B2 := (!chain-last
                             [(A2 & A3 & B1)
                              ==> (exists ?s3 .
                                     s = ?s3 ++ (actor id1 ls1) ++
                                         (actor id2 ls2)) [T]])}
               pick-witness s3 for B2 B2-w
                 let {s4 := (s3 ++ (actor id1 ls1));
                      C1 := (!chain-last
                              [A1 ==>
```

```
                                    (unique-ids s3 ++ (actor id1 ls1) ++
                                                  (actor id1 ls2))
                                    [B2-w A4]
                                    ==> (unique-ids s4 ++ (actor id1 ls2))
                                    [++A]
                                    ==> (~ (exists ?s' ?ls' .
                                             s4 = ?s' ++ (actor id1 ?ls')))
                                   [uids-nonempty]]);
                        C2 := (!chain-last
                               [(s4 = s4)
                                ==> (exists ?s' ?ls' .
                                       s4 = ?s' ++ (actor id1 ?ls'))
                                [existence]])}
                    (!absurd C2 C1))

          | (val-of unique-ids2) =>
            pick-any id1 ls1 id2 ls2
              assume A := (id1 =/= id2)
                (!by-contradiction
                   ((actor' id1 ls1) =/= (actor' id2 ls2))
                 assume B := ((actor' id1 ls1) = (actor' id2 ls2))
                  let {CI := (!constructor-injectivity actor')}
                  (!absurd
                    (!chain-last [B ==> (id1 = id2 & ls1 = ls2)  [CI]
                                    ==> (id1 = id2)        [left-and]])
                     A))
  } # match theorem

(evolve Theory [[unique-ids1 unique-ids2] proofs])
} # module Actor-Cfg
```

D.3 Transition-Path axioms

Continuing the definition of Transition-Path theory, begun in Section 3.3:

```
extend-module Transition-Path {
define trans-send :=
     (forall ?T ?s ?id ?ls ?to ?c .
        (config ?T) = ?s ++ (actor ?id ?ls) &
        (ready-to (send ?T ?id ?to ?c))
        ==>
        (config (send ?T ?id ?to ?c)) =
        ?s ++ (actor ?id (make-receptive ?ls))
           ++ (message ?id ?to ?c))
define trans-create :=
     (forall ?T ?s ?id ?ls ?new-id ?new-ls .
        (config ?T) = ?s ++ (actor ?id ?ls) &
        (ready-to (create ?T ?id ?new-id ?new-ls)) &
        (unique-ids (config ?T) ++ (actor ?new-id ?new-ls))
```

```
        ==>
        (config (create ?T ?id ?new-id ?new-ls)) =
        ?s ++ (actor ?id (make-receptive ?ls))
           ++ (actor ?new-id ?new-ls))
define trans-compute :=
     (forall ?T ?s ?id ?ls ?new-ls .
        (config ?T) = ?s ++ (actor ?id ?ls) &
        (ready-to (compute ?T ?id ?ls ?new-ls))
        ==>
        (config (compute ?T ?id ?ls ?new-ls)) =
        ?s ++ (actor ?id ?new-ls))
define Theory :=
  (theory [ACTOR-CFG.Theory]
     [trans-receive trans-send trans-create trans-compute]
     'Transition-Path)}
```

The preconditions under which each kind of transition is enabled are also expressed with the following Enabled predicate.

```
extend-module TRANSITION-PATH {
declare Enabled: (Id, State) [(TP Id State)] -> Boolean
define enabled-Initial := (Enabled Initial)
define enabled-receive :=
  (forall ?T ?id ?ls ?fr ?c .
    (Enabled (receive ?T ?id ?ls ?fr ?c)) <==>
    (Enabled ?T) &
    (exists ?s .
    (config ?T) = ?s ++ (actor ?id ?ls) ++
                  (message ?fr ?id ?c) &
    (ready-to (receive ?T ?id ?ls ?fr ?c))))
define enabled-send :=
  (forall ?T ?fr ?to ?c .
    (Enabled (send ?T ?fr ?to ?c)) <==>
    (Enabled ?T) &
    (exists ?s ?ls .
      (config ?T) = ?s ++ (actor ?fr ?ls) &
      (ready-to (send ?T ?fr ?to ?c))))
define enabled-create :=
  (forall ?T ?id ?new-id ?new-ls .
    (Enabled (create ?T ?id ?new-id ?new-ls)) <==>
    (Enabled ?T) &
    (exists ?s ?ls .
      (config ?T) = ?s ++ (actor ?id ?ls) &
      (ready-to (create ?T ?id ?new-id ?new-ls)) &
      (unique-ids (config ?T) ++ (actor ?new-id ?new-ls))))
define enabled-compute :=
  (forall ?T ?id ?ls ?new-ls .
    (Enabled (compute ?T ?id ?ls ?new-ls)) <==>
    (Enabled ?T) &
    (exists ?s .
```

```
      (config ?T) = ?s ++ (actor ?id ?ls) &
      (ready-to (compute ?T ?id ?ls ?new-ls))))
(evolve Theory
   [[enabled-Initial enabled-receive enabled-send
     enabled-create enabled-compute] Axiom])}
```

D.4 Transition Relations

For defining fairness in actor systems, it is useful to introduce a binary predicate, `-->>`, on transition paths, as axiomatized in the following TRANSITION-STEP-RELATION theory:

```
module TRANSITION-STEP-RELATION {
declare -->>: (Id, State)
  [(TP Id State) (TP Id State)] -> Boolean
define directly-leads-to-receive :=
  (forall ?T0 ?T ?id ?ls ?fr ?c .
    ?T0 -->> (receive ?T ?id ?ls ?fr ?c)
    ==> ?T0 = ?T & (Enabled (receive ?T0 ?id ?ls ?fr ?c)))
define directly-leads-to-send :=
  (forall ?T0 ?T ?fr ?to ?c .
    ?T0 -->> (send ?T ?fr ?to ?c)
    ==> ?T0 = ?T & (Enabled (send ?T0 ?fr ?to ?c)))
define directly-leads-to-create :=
  (forall ?T0 ?T ?id ?new-id ?new-ls .
    ?T0 -->> (create ?T ?id ?new-id ?new-ls)
    ==> ?T0 = ?T & (Enabled (create ?T0 ?id ?new-id ?new-ls)))
define directly-leads-to-compute :=
  (forall ?T0 ?T ?id ?ls ?new-ls .
    ?T0 -->> (compute ?T ?id ?ls ?new-ls)
    ==> ?T0 = ?T & (Enabled (compute ?T0 ?id ?ls ?new-ls)))
define Theory :=
  (theory [TRANSITION-PATH.Theory]
    [directly-leads-to-receive directly-leads-to-send
     directly-leads-to-create directly-leads-to-compute]
    'Transition-Step-Relation)}
```

Given the relation `-->>`, it is natural to consider its irreflexive and reflexive transitive closures, `-->>+` and `-->>*`, respectively:

```
module TRANSITION-PATH-RELATION {
open-module TRANSITION-STEP-RELATION
open-module TRANSITIVE-CLOSURE
declare -->>+, -->>*: (Id, State)
  [(TP Id State) (TP Id State)] -> Boolean
define nothing-leads-to-Initial :=
  (forall ?T . ~ (?T -->>+ Initial))
define Theory :=
  (theory [TRANSITION-STEP-RELATION.Theory
           [TRANSITIVE-CLOSURE.Theory
```

```
            '-->> [R -->>  R+ -->>+  R* -->>*]]]
          [nothing-leads-to-Initial]
          'Transition-Path-Relation)
```

The following theorems relating `-->>+` and `-->>*` are used in the proof of the actor and unique-ids persistence theorems in Section 3.4.

```
define leads-to-receive :=
  (forall ?T0 ?T ?id ?ls ?fr ?c .
    ?T0 -->>+ (receive ?T ?id ?ls ?fr ?c)
    ==>
    (?T0 -->>* ?T & (Enabled (receive ?T ?id ?ls ?fr ?c))))
define leads-to-send :=
  (forall ?T0 ?T ?fr ?to ?c .
    ?T0 -->>+ (send ?T ?fr ?to ?c)
    ==>
    ?T0 -->>* ?T & (Enabled (send ?T ?fr ?to ?c)))
define leads-to-create :=
  (forall ?T0 ?T ?id ?new-id ?new-ls .
    ?T0 -->>+ (create ?T ?id ?new-id ?new-ls)
    ==>
    ?T0 -->>* ?T & (Enabled (create ?T ?id ?new-id ?new-ls)))
define leads-to-compute :=
  (forall ?T0 ?T ?id ?ls ?new-ls .
    ?T0 -->>+ (compute ?T ?id ?ls ?new-ls)
    ==>
    ?T0 -->>* ?T & (Enabled (compute ?T ?id ?ls ?new-ls)))
define theorems := [leads-to-receive leads-to-send
                    leads-to-create leads-to-compute]
```

The proofs of these theorems are similar enough that we can take further advantage of Athena's programmability to define a single parameterized method that can be called with appropriate arguments to prove each theorem.

```
define proofs :=
  method (theorem adapt)
    let {given := lambda (P) (get-property P adapt Theory);
         lemma := method (P) (!property P adapt Theory);
         chain-last := method (L) (!chain-help given L 'last);
         RR*I := (!lemma ['-->> R+R*-inclusion]);
         R+D := (given ['-->> R+-definition]);
         R*D := (given ['-->> R*-definition]);
         proof :=
         method (step T0 T directly)
          assume A := (T0 -->>+ (step T))
            let {B1 := (!chain-last
                        [A ==>
                           (T0 -->> (step T) |
                            (exists ?T1 .
                              T0 -->>+ ?T1 &
```

```
                          ?T1 -->> (step T)))  [R+D]])}
        (!cases B1
          assume C1 := (T0 -->> (step T))
            let {(and D1 D2) :=
                   (!chain-last
                    [C1 ==> (T0 = T &
                             (Enabled (step T0)))
                                              [directly]])}
            (!chain-last
             [D1 ==> (D1 | T0 -->>+ T)        [alternate]
                 ==> (T0 -->>* T)             [R*D]
                 ==> (T0 -->>* T & D2)        [augment]
                 ==> (T0 -->>* T &
                          (Enabled (step T))) [D1]])
          assume C2 := (exists ?T1 .
                          T0 -->>+ ?T1 &
                          ?T1 -->> (step T))
            pick-witness T1 for C2 C2-w
              let {C2-w1 := (T0 -->>+ T1);
                   C2-w2 := (T1 -->> (step T));
                   (and D1 D2) :=
                     (!chain-last
                      [C2-w2
                       ==> (T1 = T &
                            (Enabled (step T1)))
                                              [directly]]);
                   RR*I := (!lemma
                              ['-->> R+R*-inclusion])}
              (!chain-last
               [C2-w1 ==> (T0 -->>+ T)        [D1]
                      ==> (T0 -->>* T)        [RR*I]
                      ==> (T0 -->>* T & D2)   [augment]
                      ==> (T0 -->>* T &
                           (Enabled (step T))) [D1]]))}
match theorem {
  (val-of leads-to-receive) =>
  pick-any T0 T id ls fr c
    (!proof lambda (T) (receive T id ls fr c)
            T0 T (given directly-leads-to-receive))
| (val-of leads-to-send) =>
  pick-any T0 T fr to c
    (!proof lambda (T) (send T fr to c)
            T0 T (given directly-leads-to-send))
| (val-of leads-to-create) =>
  pick-any T0 T id new-id new-ls
    (!proof lambda (T) (create T id new-id new-ls)
            T0 T (given directly-leads-to-create))
| (val-of leads-to-compute) =>
  pick-any T0 T id ls new-ls
    (!proof lambda (T) (compute T id ls new-ls)
```

```
                T0 T (given directly-leads-to-compute))
    }
(evolve Theory [theorems proofs])}
```

D.5 Persistence Proofs

```
extend-module TRANSITION-PATH-RELATION {
define actor-persistence-proof :=
 method (theorem adapt)
   let {given := lambda (P) (get-property P adapt Theory);
        lemma := method (P) (!property P adapt Theory);
        chain := method (L) (!chain-help given L 'none);
        chain-last := method (L) (!chain-help given L 'last);
        ++A := (given ['++ Associative]);
        ++C := (given ['++ Commutative])}
   by-induction theorem {
     Initial:(TP 'Id 'LS) =>
       pick-any T0:(TP 'Id 'LS)
                s0:(CFG (Actor 'Id 'LS))
                id0:'Id
                ls0:'LS
         let {A1 := ((config T0) = s0 ++ (actor id0 ls0));
              A2 := (T0 -->>* Initial);
              A3 := (unique-ids (config T0))}
         assume (A1 & A2 & A3)
           let {goal := (exists ?s ?ls .
                          (config Initial) = ?s ++ (actor id0 ?ls));
                B1 := (!chain-last
                        [A2 ==> (T0 = Initial | T0 -->>+ Initial)
                            [(given ['-->> R*-definition])]])}
           (!cases B1
             assume B1a := (T0 = Initial)
               (!chain-last [A1 ==> ((config Initial) =
                                     s0 ++ (actor id0 ls0)) [B1a]
                                     ==> goal             [existence]])
             assume B1b := (T0 -->>+ Initial)
               (!from-complements goal B1b
                  (!chain-last [true ==> (~ B1b)
                                     [nothing-leads-to-Initial]])))
   | (receive T:(TP 'Id 'LS) id ls fr c) =>
     let {ind-hyp := (forall ?T0 ?s0 ?id ?ls0 .
                       (config ?T0) = ?s0 ++ (actor ?id ?ls0) &
                       (?T0 -->>* T) &
                       (unique-ids (config ?T0))
                       ==> (exists ?s ?ls .
                             (config T) = ?s ++ (actor ?id ?ls)))}
     pick-any T0 s0 id0 ls0
       let {A1 := ((config T0) = s0 ++ (actor id0 ls0));
```

```
     A2 := (T0 -->>* (receive T id ls fr c));
     A3 := (unique-ids (config T0))}
assume (A1 & A2 & A3)
  let {B1 := (!chain-last
              [A2 ==> (T0 = (receive T id ls fr c) |
                          T0 -->>+  (receive T id ls fr c))
                            [(given ['-->> R*-definition])]])}
  (!cases B1
    assume B1a := (T0 = (receive T id ls fr c))
      (!chain-last [A1 ==> ((config (receive T id ls fr c)) =
                            s0 ++ (actor id0 ls0))  [B1a]
                       ==> (exists ?s ?ls .
                             (config (receive T id ls fr c)) =
                             ?s ++ (actor id0 ?ls))
                                   [existence]])
    assume B1b := (T0 -->>+ (receive T id ls fr c))
      let {goal := (exists ?s ?ls .
                     (config (receive T id ls fr c)) =
                     ?s ++ (actor id0 ?ls));
           LTR := (!lemma leads-to-receive);
           B := (!chain-last
                 [B1b
                  ==> (T0 -->>* T &
                       (Enabled (receive T id ls fr c)))  [LTR]
                  ==> (Enabled (receive T id ls fr c)) [right-and]
                  ==> ((Enabled T) &
                       (exists ?s .
                         (config T) = ?s ++ (actor id ls) ++
                                      (message fr id c) &
                         (ready-to (receive T id ls fr c))))
                                          [enabled-receive]])}
      pick-witness s for (!right-and B) w
        let {w1 := ((config T) = s ++ (actor id ls) ++
                    (message fr id c));
             w2 := (ready-to (receive T id ls fr c))}
        (!two-cases
           assume (id = id0)
             (!chain-last
              [w ==> ((config (receive T id ls fr c)) =
                      s ++ (actor id (accept id ls fr c)))
                     [trans-receive]
                 ==> ((config (receive T id ls fr c)) =
                      s ++ (actor id0 (accept id0 ls fr c)))
                     [(id = id0)]
                 ==> goal     [existence]])
           assume (id =/= id0)
             let {C := (!chain-last
                        [B1b
                         ==> (T0 -->>* T &
                                  (Enabled (receive T id ls fr c)))
```

```
                                [leads-to-receive]
                                ==> (T0 -->>* T)            [left-and]
                                ==> (A1 & T0 -->>* T & A3)  [augment]
                                ==> (exists ?s1 ?ls1 .
                                      (config T) = ?s1 ++ (actor id0 ?ls1))
                                    [ind-hyp]])}
                        pick-witnesses s1 ls1 for C C-w
                          let {D1 := (!chain-last
                                      [(id =/= id0)
                                       ==> (id0 =/= id)   [sym]
                                       ==> ((actor' id0 ls1) =/=
                                            (actor' id ls))
                                       [unique-ids2]]);
                               D2 := (!chain-last
                                      [true
                                       ==> ((actor' id0 ls1) =/=
                                            (message' fr id c))
                                      [(exclusive-constructors
                                        "Actor-Cfg.Actor")]]);
                               MT := (!lemma More-Together);
                               D3 := (!mp (!uspec* MT
                                                   [(config T) s1 s
                                                    (actor' id0 ls1)
                                                    (actor' id ls)
                                                    (message' fr id c)])
                                          (!both C-w (!both w1 (!both D1 D2))))}
                            pick-witness s2 for D3 D3-w
                              let {E1 :=
                                   (!chain
                                    [(config T)
                                     = (s2 ++ (actor id0 ls1) ++ (actor id ls)
                                           ++ (message fr id c))        [D3-w]
                                     = ((s2 ++ (actor id0 ls1)) ++
                                        (actor id ls) ++ (message fr id c))
                                                                      [++A]])}
                              (!chain-last
                               [(E1 & w2)
                                ==> ((config (receive T id ls fr c)) =
                                     ((s2 ++ (actor id0 ls1))
                                      ++ (actor id (accept id ls fr c))))
                                    [trans-receive]
                                ==> ((config (receive T id ls fr c)) =
                                     ((s2 ++ (actor id (accept id ls fr c)))
                                      ++ (actor id0 ls1)))           [++A ++C]
                                ==> goal [existence]])))
  | (send T:(TP 'Id 'LS) fr to c) =>
    let {ind-hyp := (forall ?T0 ?s0 ?id ?ls0 .
                      (config ?T0) = ?s0 ++ (actor ?id ?ls0) &
                      (?T0 -->>* T) &
                      (unique-ids (config ?T0))
```

```
                  ==> (exists ?s ?ls .
                         (config T) = ?s ++ (actor ?id ?ls)))}
pick-any T0:(TP 'Id 'LS) s0 id0 ls0
  let {A1 := ((config T0) = s0 ++ (actor id0 ls0));
       A2 := (T0 -->>* (send T fr to c));
       A3 := (unique-ids (config T0))}
  assume (A1 & A2 & A3)
    let {B1 := (!chain-last [A2 ==> (T0 = (send T fr to c) |
                                     T0 -->>+ (send T fr to c))
                                [(given ['-->> R*-definition])]])}
    (!cases B1
      assume B1a := (T0 = (send T fr to c))
      (!chain-last [A1 ==> ((config (send T fr to c)) =
                            s0 ++ (actor id0 ls0)) [B1a]
                       ==> (exists ?s ?ls .
                             (config (send T fr to c)) =
                             ?s ++ (actor id0 ?ls))
                       [existence]])
      assume B1b := (T0 -->>+ (send T fr to c))
        let {goal := (exists ?s ?ls .
                       (config (send T fr to c)) =
                       ?s ++ (actor id0 ?ls));
             LTS := (!lemma leads-to-send);
             B := (!chain-last
                    [B1b ==> (T0 -->>* T & (Enabled (send T fr to c)))
                         [LTS]
                         ==> (Enabled (send T fr to c)) [right-and]
                         ==> ((Enabled T) &
                              (exists ?s ?ls .
                                (config T) = ?s ++ (actor fr ?ls) &
                                (ready-to (send T fr to c))))
                         [enabled-send]])}
        pick-witnesses s ls for (!right-and B) w
          let {w1 := ((config T) = s ++ (actor fr ls));
               w2 := (ready-to (send T fr to c))}
          (!two-cases
           assume (fr = id0)
             (!chain-last
              [w ==> ((config (send T fr to c)) =
                      s ++ (actor fr (make-receptive ls)) ++
                      (message fr to c))
                 [trans-send]
                 ==> ((config (send T fr to c)) =
                      (s ++ (message fr to c)) ++
                      (actor fr (make-receptive ls)))
                 [++A ++C]
                 ==> ((config (send T fr to c)) =
                      (s ++ (message fr to c)) ++
                      (actor id0 (make-receptive ls)))
                 [(fr = id0)]
```

```
                    ==> goal       [existence]])
              assume (fr =/= id0)
                let {C := (!chain-last
                           [B1b
                            ==> (T0 -->>* T &
                                 (Enabled (send T fr to c))) [LTS]
                            ==> (T0 -->>* T)            [left-and]
                            ==> (A1 & T0 -->>* T & A3)  [augment]
                            ==> (exists ?s1 ?ls1 .
                                   (config T) =
                                   ?s1 ++ (actor id0 ?ls1))
                            [ind-hyp]])}
                pick-witnesses s1 ls1 for C C-w
                  let {IST := (!lemma Together);
                       D1 :=  (!chain-last
                               [(fr =/= id0)
                                ==> (id0 =/= fr)   [sym]
                                ==> ((actor' id0 ls1) =/=
                                     (actor' fr ls))
                                [unique-ids2]]);
                       D3 := (!chain-last
                              [(C-w & w1 & D1)
                               ==> (exists ?s2 .
                                     (config T) =
                                     ?s2 ++ (actor id0 ls1) ++
                                     (actor fr ls))   [IST]])}
                  pick-witness s2 for D3 D3-w
                    let {E1 := (!chain
                                [(config T)
                                 = (s2 ++ (actor id0 ls1)
                                       ++ (actor fr ls))   [D3-w]
                                 = ((s2 ++ (actor id0 ls1))
                                    ++ (actor fr ls))   [++A]])}
                    (!chain-last
                     [(E1 & w2) ==> ((config (send T fr to c)) =
                                     ((s2 ++ (actor id0 ls1)) ++
                                      (actor fr (make-receptive ls)) ++
                                      (message fr to c)))
                                [trans-send]
                                ==> ((config (send T fr to c)) =
                                     ((s2 ++
                                       (actor fr (make-receptive ls)) ++
                                       (message fr to c)) ++
                                       (actor id0 ls1)))    [++A ++C]
                                ==> goal [existence]])))
  | (create T:(TP 'Id 'LS) id' new-id new-ls) =>
    let {ind-hyp := (forall ?T0 ?s0 ?id ?ls0 .
                      (config ?T0) = ?s0 ++ (actor ?id ?ls0) &
                      (?T0 -->>* T) &
                      (unique-ids (config ?T0))
```

```
                  ==> (exists ?s ?ls .
                         (config T) = ?s ++ (actor ?id ?ls)))}
pick-any T0:(TP 'Id 'LS) s0 id0 ls0
  let {A1 := ((config T0) = s0 ++ (actor id0 ls0));
       A2 := (T0 -->>* (create T id' new-id new-ls));
       A3 := (unique-ids (config T0))}
  assume (A1 & A2 & A3)
    let {B1 := (!chain-last
                [A2 ==> (T0 = (create T id' new-id new-ls) |
                              T0 -->>+ (create T id' new-id new-ls))
                          [(given ['-->> R*-definition])]])}
    (!cases B1
      assume B1a := (T0 = (create T id' new-id new-ls))
        (!chain-last [A1 ==> ((config (create T id' new-id new-ls)) =
                              s0 ++ (actor id0 ls0))        [B1a]
                         ==> (exists ?s ?ls .
                               (config (create T id' new-id new-ls)) =
                               ?s ++ (actor id0 ?ls))
                                                    [existence]])
      assume B1b := (T0 -->>+ (create T id' new-id new-ls))
        let {goal := (exists ?s ?ls .
                       (config (create T id' new-id new-ls)) =
                       ?s ++ (actor id0 ?ls));
             LTC := (!lemma leads-to-create);
             B := (!chain-last
                   [B1b ==> (T0 -->>* T &
                             (Enabled (create T id' new-id new-ls)))
                                      [LTC]
                        ==> (Enabled (create T id' new-id new-ls))
                                      [right-and]
                        ==>
                        ((Enabled T) &
                         (exists ?s ?ls .
                           (config T) = ?s ++ (actor id' ?ls) &
                           (ready-to (create T id' new-id new-ls)) &
                           (unique-ids (config T) ++
                                       (actor new-id new-ls))))
                                      [enabled-create]])}
        pick-witnesses s ls for (!right-and B) w
          let {w1 := ((config T) = s ++ (actor id' ls));
               w2 := (ready-to (create T id' new-id new-ls));
               w3 := (unique-ids (config T) ++ (actor new-id new-ls))}
          (!two-cases
            assume (id' = id0)
              (!chain-last
               [w ==> ((config (create T id' new-id new-ls)) =
                       s ++ (actor id' (make-receptive ls)) ++
                       (actor new-id new-ls))
                  [trans-create]
                  ==> ((config (create T id' new-id new-ls)) =
```

```
                    (s ++ (actor new-id new-ls)) ++
                    (actor id' (make-receptive ls)))
               [++A ++C]
               ==> ((config (create T id' new-id new-ls)) =
                    (s ++ (actor new-id new-ls)) ++
                    (actor id0 (make-receptive ls)))
               [(id' = id0)]
               ==> goal       [existence]])
          assume (id' =/= id0)
            let {C := (!chain-last
                       [B1b
                        ==>(T0 -->>* T &
                            (Enabled (create T id' new-id new-ls)))
                        [LTC]
                        ==> (T0 -->>* T)            [left-and]
                        ==> (A1 & T0 -->>* T & A3)  [augment]
                        ==> (exists ?s1 ?ls1 .
                              (config T) = ?s1 ++ (actor id0 ?ls1))
                        [ind-hyp]])}
            pick-witnesses s1 ls1 for C C-w
              let {D1 := (!chain-last
                          [(id' =/= id0)
                           ==> (id0 =/= id')   [sym]
                           ==> ((actor' id0 ls1) =/=
                                (actor' id' ls))
                           [unique-ids2]]);
                   IST := (!lemma Together);
                   D3 := (!chain-last
                          [(C-w & w1 & D1)
                           ==> (exists ?s2 .
                                 (config T) =
                                 ?s2 ++ (actor id0 ls1) ++
                                 (actor id' ls))    [IST]])}
              pick-witness s2 for D3 D3-w
                let {E1 := (!chain
                            [(config T)
                             = (s2 ++ (actor id0 ls1) ++
                                (actor id' ls))   [D3-w]
                             = ((s2 ++ (actor id0 ls1)) ++
                                (actor id' ls))   [++A]])}
                (!chain-last
                 [(E1 & w2 & w3)
                  ==> ((config (create T id' new-id new-ls))
                       = (s2 ++ (actor id0 ls1)) ++
                       (actor id' (make-receptive ls)) ++
                       (actor new-id new-ls))
                  [trans-create]
                  ==> ((config (create T id' new-id new-ls))
                       = (s2 ++
                             (actor id' (make-receptive ls))
```

```
                                  ++ (actor new-id new-ls)) ++
                                  (actor id0 ls1))     [++A ++C]
                            ==> goal [existence]])))
| (compute T:(TP 'Id 'LS) id' ls new-ls) =>
  let {ind-hyp := (forall ?T0 ?s0 ?id0 ?ls0 .
                    (config ?T0) = ?s0 ++ (actor ?id0 ?ls0) &
                    (?T0 -->>* T) &
                    (unique-ids (config ?T0))
                    ==> (exists ?s ?ls .
                        (config T) = ?s ++ (actor ?id0 ?ls)))}
  pick-any T0:(TP 'Id 'LS) s0 id0 ls0
    let {A1 := ((config T0) = s0 ++ (actor id0 ls0));
         A2 := (T0 -->>* (compute T id' ls new-ls));
         A3 := (unique-ids (config T0))}
    assume (A1 & A2 & A3)
      let {B1 := (!chain-last
                  [A2 ==> (T0 = (compute T id' ls new-ls) |
                                  T0 -->>+ (compute T id' ls new-ls))
                                  [(given ['-->> R*-definition])]])}
      (!cases B1
        assume B1a := (T0 = (compute T id' ls new-ls))
          (!chain-last [A1 ==> ((config (compute T id' ls new-ls)) =
                                s0 ++ (actor id0 ls0))      [B1a]
                               ==> (exists ?s ?ls .
                                  (config (compute T id' ls new-ls)) =
                                  ?s ++ (actor id0 ?ls))  [existence]])
        assume B1b := (T0 -->>+ (compute T id' ls new-ls))
          let {goal := (exists ?s ?ls .
                         (config (compute T id' ls new-ls)) =
                         ?s ++ (actor id0 ?ls));
               LTC := (!lemma leads-to-compute);
               B := (!chain-last
                     [B1b ==> (T0 -->>* T &
                                  (Enabled (compute T id' ls new-ls)))
                              [LTC]
                              ==> (Enabled (compute T id' ls new-ls))
                              [right-and]
                              ==> ((Enabled T) &
                                   (exists ?s .
                                     (config T) = ?s ++ (actor id' ls) &
                                     (ready-to (compute T id' ls new-ls))))
                              [enabled-compute]])}
          pick-witness s for (!right-and B) w
            let {w1 := ((config T) = s ++ (actor id' ls));
                 w2 := (ready-to (compute T id' ls new-ls))}
            (!two-cases
              assume (id' = id0)
                (!chain-last
                 [w ==> ((config (compute T id' ls new-ls)) =
                         s ++ (actor id' new-ls))
```

xxx

```
                        [trans-compute]
                        ==> ((config (compute T id' ls new-ls)) =
                             s ++ (actor id0 new-ls))
                        [(id' = id0)]
                        ==> goal       [existence]])
                  assume (id' =/= id0)
                    let {C := (!chain-last
                                [B1b ==> (T0 -->>* T &
                                          (Enabled (compute T id' ls new-ls)))
                                 [leads-to-compute]
                                 ==> (T0 -->>* T)            [left-and]
                                 ==> (A1 & T0 -->>* T & A3)  [augment]
                                 ==> (exists ?s1 ?ls1 .
                                       (config T) =
                                       ?s1 ++ (actor id0 ?ls1))
                                 [ind-hyp]])}
                    pick-witnesses s1 ls1 for C C-w
                      let {D1 := (!chain-last
                                  [(id' =/= id0)
                                   ==> (id0 =/= id')   [sym]
                                   ==> ((actor' id0 ls1) =/=
                                        (actor' id' ls))
                                   [unique-ids2]]);
                           IST := (!lemma Together);
                           D3 := (!chain-last
                                  [(C-w & w1 & D1)
                                   ==> (exists ?s2 .
                                         (config T) =
                                         ?s2 ++ (actor id0 ls1) ++
                                         (actor id' ls))  [IST]])}
                      pick-witness s2 for D3 D3-w
                        let {E1 := (!chain
                                    [(config T)
                                     = (s2 ++ (actor id0 ls1) ++
                                        (actor id' ls))   [D3-w]
                                     = ((s2 ++ (actor id0 ls1))
                                           ++ (actor id' ls))  [++A]])}
                        (!chain-last
                         [(E1 & w2)
                          ==> ((config (compute T id' ls new-ls)) =
                               (s2 ++ (actor id0 ls1)) ++
                               (actor id' new-ls))
                          [trans-compute]
                          ==> ((config (compute T id' ls new-ls)) =
                               (s2 ++ (actor id' new-ls)) ++
                               (actor id0 ls1))
                          [++A ++C]
                          ==> goal [existence]])))
          } # by-induction
(evolve Theory [[actor-persistence] actor-persistence-proof])}
```

D.6 Fairness

First, the full definition of TRANSITION-SEQUENCE, which includes a theorem about connectivity of the reflexive transitive closure, `-->>*`, of `-->>`.

```
module TRANSITION-SEQUENCE {
open-module TRANSITION-PATH-RELATION
open-module N
declare ts: (Id, State) [(TP Id State) N] -> (TP Id State)
define ts-initial := (forall ?T . (ts ?T zero) = ?T)
define ts-directly-connected :=
       (forall ?T ?n . (ts ?T ?n) -->> (ts ?T (S ?n)))
define Theory :=
  (theory [TRANSITION-PATH-RELATION.Theory]
    [ts-initial ts-directly-connected]
    'Transition-Sequence)
define ts-connected :=
       (forall ?m ?n ?T . ?m >= ?n ==> (ts ?T ?n) -->>* (ts ?T ?m))
define proof :=
  method (theorem adapt)
    let {given := lambda (P) (get-property P adapt Theory);
         lemma := method (P) (!property P adapt Theory);
         chain := method (L) (!chain-help given L 'none);
         chain-last := method (L) (!chain-help given L 'last);
         L1 := (!lemma ['-->> R*-Reflexive]);
         L2 := (!lemma ['-->> RR+-inclusion]);
         L3 := (!lemma ['-->> R+R*-inclusion]);
         L4 := (!lemma ['-->> TC-Transitivity3])}
    match theorem {
      (val-of ts-connected) =>
      by-induction theorem {
        zero =>
        pick-any n T
          assume A := (zero >= n)
            let {B := (!chain-last [A ==> (n = zero)
                                    [N.Less=.zero2]])}
            (!chain-last
             [true
              ==> ((ts T zero) -->>* (ts T zero))  [L1]
              ==> ((ts T n) -->>* (ts T zero))     [B]])
      | (S m) =>
        let {ind-hyp :=
               (forall ?n ?T . m >= ?n ==> (ts ?T ?n) -->>* (ts ?T m))}
        pick-any n T
          assume A := ((S m) >= n)
            (!two-cases
              assume A1 := ((S m) = n)
```

```
                (!chain-last
                 [true ==> ((ts T n) -->>* (ts T n)) [L1]
                       ==> ((ts T n) -->>* (ts T (S m))) [A1]])
              assume A2 := ((S m) =/= n)
                let {B1 := (!chain-last
                            [A2 ==> (n =/= (S m))          [sym]
                                ==> (A & n =/= (S m))      [augment]
                                ==> (m >= n)               [N.Less=.S5]
                                ==> ((ts T n) -->>* (ts T m)) [ind-hyp]])}
                (!chain-last
                 [true
                  ==> ((ts T m) -->> (ts T (S m)))
                                     [ts-directly-connected]
                  ==> ((ts T m) -->>+ (ts T (S m)))      [L2]
                  ==> ((ts T m) -->>* (ts T (S m)))      [L3]
                  ==> (B1 & (ts T m) -->>* (ts T (S m))) [augment]
                  ==> ((ts T n) -->>* (ts T (S m)))      [L4]]))
      }
   }
(evolve Theory [[ts-connected] proof])
}
```

The receive axiom for infinitely-often-enabled sequences was given in Section 4.

```
extend-module INFINITELY-OFTEN-ENABLED {
define ioe-send :=
  (forall ?T ?n ?s ?ls ?to ?c .
    (config (ts ?T ?n)) = ?s ++ (actor <sender> ?ls) &
    (ready-to (send (ts ?T ?n) <sender> ?to ?c))
    ==>
    (ts ?T (S ?n)) = (send (ts ?T ?n) <sender> ?to ?c) |
      (exists ?k ?s' ?ls' .
        ?k > ?n &
        (config (ts ?T ?k)) = ?s' ++ (actor <sender> ?ls') &
        (ready-to (send (ts ?T ?k) <sender> ?to ?c))))
define ioe-create :=
  (forall ?T ?n ?s ?ls ?new-id ?new-ls .
    (config (ts ?T ?n)) = ?s ++ (actor creator ?ls) &
    (ready-to (create (ts ?T ?n) creator ?new-id ?new-ls)) &
    (unique-ids (config (ts ?T ?n)) ++ (actor ?new-id ?new-ls))
    ==>
    (ts ?T (S ?n)) =
    (create (ts ?T ?n) creator ?new-id ?new-ls) |
     (exists ?k ?s' ?ls' ?new-id' .
       ?k > ?n &
       (config (ts ?T ?k)) = ?s' ++ (actor creator ?ls') &
       (ready-to (create (ts ?T ?k) creator ?new-id' ?new-ls)) &
       (unique-ids (config (ts ?T ?k)) ++ (actor ?new-id' ?new-ls))))
define ioe-compute :=
  (forall ?T ?n ?s ?ls ?new-ls .
```

```
    (config (ts ?T ?n)) = ?s ++ (actor computer ?ls) &
    (ready-to (compute (ts ?T ?n) computer ?ls ?new-ls))
    ==>
    (ts ?T (S ?n)) = (compute (ts ?T ?n) computer ?ls ?new-ls) |
     (exists ?k ?s' ?ls' ?new-ls' .
       ?k > ?n &
       (config (ts ?T ?k)) = ?s' ++ (actor computer ?ls') &
       (ready-to (compute (ts ?T ?k) computer ?ls' ?new-ls'))))
define Theory :=
  (theory [TRANSITION-SEQUENCE.Theory]
    [ioe-receive ioe-send ioe-create ioe-compute]
    'Infinitely-Often-Enabled)}
```

The fair-receive axiom of Fair-Transition-Sequence theory was given in Section 4.

```
extend-module FAIR-TRANSITION-SEQUENCE {
define fair-send :=
  (forall ?id ?T ?n ?s ?ls ?to ?c .
    (config (ts ?T ?n)) = ?s ++ (actor ?id ?ls) &
    (ready-to (send (ts ?T ?n) ?id ?to ?c))
    ==>
    (exists ?k ?s' ?ls' .
      ?k >= ?n &
      (config (ts ?T ?k)) = ?s' ++ (actor ?id ?ls') &
      (ready-to (send (ts ?T ?k) ?id ?to ?c)) &
      (ts ?T (S ?k)) = (send (ts ?T ?k) ?id ?to ?c)) |
    ~ (exists ?k ?s' ?ls' .
        ?k > ?n &
        (config (ts ?T ?k)) = ?s' ++ (actor ?id ?ls') &
        (ready-to (send (ts ?T ?k) ?id ?to ?c))))

define fair-create :=
  (forall ?id ?T ?n ?s ?ls ?new-id ?new-ls .
    (config (ts ?T ?n)) = ?s ++ (actor ?id ?ls) &
    (ready-to (create (ts ?T ?n) ?id ?new-id ?new-ls)) &
    (unique-ids (config (ts ?T ?n)) ++ (actor ?new-id ?new-ls))
    ==>
    (exists ?k ?s' ?ls' ?new-id' .
      ?k >= ?n &
     (config (ts ?T ?k)) = ?s' ++ (actor ?id ?ls') &
     (ready-to (create (ts ?T ?k) ?id ?new-id' ?new-ls)) &
     (unique-ids (config (ts ?T ?k)) ++ (actor ?new-id' ?new-ls)) &
     (ts ?T (S ?k)) = (create (ts ?T ?k) ?id ?new-id' ?new-ls)) |
   ~ (exists ?k ?s' ?ls' ?new-id' .
       ?k > ?n &
       (config (ts ?T ?k)) = ?s' ++ (actor ?id ?ls') &
       (ready-to (create (ts ?T ?k) ?id ?new-id' ?new-ls)) &
       (unique-ids (config (ts ?T ?k)) ++ (actor ?new-id' ?new-ls))))

define fair-compute :=
```

```
  (forall ?id ?T ?n ?s ?ls ?new-ls .
    (config (ts ?T ?n)) = ?s ++ (actor ?id ?ls) &
    (ready-to (compute (ts ?T ?n) ?id ?ls ?new-ls))
    ==>
    (exists ?k ?s' ?ls' ?new-ls' .
      ?k >= ?n &
      (config (ts ?T ?k)) = ?s' ++ (actor ?id ?ls') &
      (ready-to (compute (ts ?T ?k) ?id ?ls' ?new-ls')) &
      (ts ?T (S ?k)) = (compute (ts ?T ?k) ?id ?ls' ?new-ls')) |
    ~ (exists ?k ?s' ?ls' ?new-ls' .
      ?k > ?n &
      (config (ts ?T ?k)) = ?s' ++ (actor ?id ?ls') &
      (ready-to (compute (ts ?T ?k) ?id ?ls' ?new-ls'))))

define Theory :=
  (theory [TRANSITION-SEQUENCE.Theory]
    [fair-receive fair-send fair-create fair-compute]
    'Fair-Transition-Sequence)}
```

The combination of these two theories gives us the basis for proving that progress occurs in an actor system, as illustrated in the clock example.

```
module IOE-FAIR-TRANSITION-SEQUENCE {
open-module INFINITELY-OFTEN-ENABLED
open-module FAIR-TRANSITION-SEQUENCE

define Theory :=
  (theory [INFINITELY-OFTEN-ENABLED.Theory FAIR-TRANSITION-SEQUENCE.Theory]
    [] 'IOE-Fair-Transition-Sequence)}
```

D.7 Fairness Theorems

In Section 4 we stated the fairness theorem for the receive transition. Here we state the corresponding theorems for the other transitions and give the proofs for all transitions.

```
extend-module IOE-FAIR-TRANSITION-SEQUENCE {
define fair-send-theorem :=
  (forall ?T ?n ?s ?ls ?to ?c .
    (config (ts ?T ?n)) = ?s ++ (actor <sender> ?ls) &
    (ready-to (send (ts ?T ?n) <sender> ?to ?c))
    ==>
    (exists ?m ?s' ?ls' .
      ?m >= ?n &
      (config (ts ?T ?m)) = ?s' ++ (actor <sender> ?ls') &
      (ready-to (send (ts ?T ?m) <sender> ?to ?c)) &
      (ts ?T (S ?m)) = (send (ts ?T ?m) <sender> ?to ?c)))

define fair-create-theorem :=
  (forall ?T ?n ?s ?ls ?new-id ?new-ls .
```

```
    (config (ts ?T ?n)) = ?s ++ (actor creator ?ls) &
    (ready-to (create (ts ?T ?n) creator ?new-id ?new-ls)) &
    (unique-ids (config (ts ?T ?n)) ++ (actor ?new-id ?new-ls))
    ==>
    (exists ?m ?s' ?ls' ?new-id' .
      ?m >= ?n &
      (config (ts ?T ?m)) = ?s' ++ (actor creator ?ls') &
      (ready-to (create (ts ?T ?m) creator ?new-id' ?new-ls)) &
      (unique-ids (config (ts ?T ?m)) ++ (actor ?new-id' ?new-ls)) &
      (ts ?T (S ?m)) = (create (ts ?T ?m) creator ?new-id' ?new-ls)))

define fair-compute-theorem :=
  (forall ?T ?n ?s ?ls ?new-ls .
  (config (ts ?T ?n)) = ?s ++ (actor computer ?ls) &
  (ready-to (compute (ts ?T ?n) computer ?ls ?new-ls))
  ==>
  (exists ?m ?s' ?ls' ?new-ls' .
    ?m >= ?n &
    (config (ts ?T ?m)) = ?s' ++ (actor computer ?ls') &
    (ready-to (compute (ts ?T ?m) computer ?ls' ?new-ls')) &
    (ts ?T (S ?m)) = (compute (ts ?T ?m) computer ?ls' ?new-ls')))

define theorems :=
  [fair-receive-theorem fair-send-theorem fair-create-theorem
   fair-compute-theorem]

define proofs :=
  method (theorem adapt)
    let {given := lambda (P) (get-property P adapt Theory);
         lemma := method (P) (!property P adapt Theory);
         chain := method (L) (!chain-help given L 'none);
         chain-last := method (L) (!chain-help given L 'last);
         [ts receiver sender creator computer] :=
           (adapt [ts receiver sender creator computer])}
      match theorem {
       (val-of fair-receive-theorem) =>
       pick-any T n s ls fr c
        let {A1 := ((config (ts T n)) =
                    s ++ (actor receiver ls) ++
                    (message fr receiver c));
             A2 := (ready-to (receive (ts T n) receiver ls fr c))}
        assume A := (A1 & A2)
          let {B1 := ((ts T (S n)) =
                      (receive (ts T n) receiver ls fr c));
               B2 := (exists ?k ?s' ?ls' .
                       ?k > n &
                       (config (ts T ?k)) =
                       ?s' ++ (actor receiver ?ls') ++
                       (message fr receiver c) &
                       (ready-to (receive (ts T ?k) receiver ?ls'
```

```
                                        fr c)));
             goal := (exists ?k ?s' ?ls' .
                       ?k >= n &
                       (config (ts T ?k)) =
                         ?s' ++ (actor receiver ?ls') ++
                         (message fr receiver c) &
                       (ready-to (receive (ts T ?k) receiver ?ls'
                                          fr c)) &
                       (ts T (S ?k)) =
                         (receive (ts T ?k) receiver ?ls' fr c));
              B3 := (!chain-last [A ==> (B1 | B2) [ioe-receive]])}
          (!cases B3
           assume B1
             (!chain-last
              [true ==> (n >= n)  [N.lessEqual.reflexive]
                    ==> (n >= n & A1 & A2 & B1)  [augment]
                    ==> goal                     [existence]])
           assume B2
             let {C := (!chain-last [A ==> (goal | ~ B2)
                                           [fair-receive]])}
             (!cases C
              assume goal
                (!claim goal)
              assume (~ B2)
                (!from-complements goal B2 (~ B2))))
    | (val-of fair-send-theorem) =>
        pick-any T n s ls to c
         let {A1 := ((config (ts T n)) = s ++ (actor sender ls));
              A2 := (ready-to (send (ts T n) sender to c))}
         assume A := (A1 & A2)
            let {B1 := ((ts T (S n)) =
                        (send (ts T n) sender to c));
                 B2 := (exists ?k ?s' ?ls' .
                          ?k > n &
                          (config (ts T ?k)) =
                            ?s' ++ (actor sender ?ls') &
                          (ready-to (send (ts T ?k) sender to c)));
                 goal := (exists ?k ?s' ?ls' .
                           ?k >= n &
                           (config (ts T ?k)) =
                             ?s' ++ (actor sender ?ls') &
                           (ready-to (send (ts T ?k) sender to c)) &
                           (ts T (S ?k)) =
                             (send (ts T ?k) sender to c));
                 B3 := (!chain-last [A ==> (B1 | B2) [ioe-send]])}
              (!cases B3
                assume B1
                  (!chain-last
                   [true
                    ==> (n >= n)  [N.lessEqual.reflexive]
```

```
                ==> (n >= n & A1 & A2 & B1)  [augment]
                ==> goal                     [existence]])
            assume B2
              let {C := (!chain-last [A ==> (goal | ~ B2)
                                        [fair-send]])}
                (!cases C
                  assume goal
                    (!claim goal)
                  assume (~ B2)
                      (!from-complements goal B2 (~ B2))))
| (val-of fair-create-theorem) =>
    pick-any T n s ls new-id new-ls
     let {A1 := ((config (ts T n)) = s ++ (actor creator ls));
          A2 := (ready-to (create (ts T n) creator
                                  new-id new-ls));
          A3 := (unique-ids (config (ts T n)) ++
                            (actor new-id new-ls))}
      assume A := (A1 & A2 & A3)
        let {B1 := ((ts T (S n)) =
                    (create (ts T n) creator new-id new-ls));
             B2 := (exists ?k ?s' ?ls' ?new-id' .
                     ?k > n &
                     (config (ts T ?k)) = ?s' ++
                                          (actor creator ?ls') &
                     (ready-to (create (ts T ?k) creator
                                       ?new-id' new-ls)) &
                     (unique-ids (config (ts T ?k)) ++
                                 (actor ?new-id' new-ls)));
             goal := (exists ?k ?s' ?ls' ?new-id' .
                       ?k >= n &
                       (config (ts T ?k)) = ?s' ++
                       (actor creator ?ls') &
                       (ready-to (create (ts T ?k) creator
                                         ?new-id' new-ls)) &
                       (unique-ids (config (ts T ?k)) ++
                                   (actor ?new-id' new-ls)) &
                       (ts T (S ?k)) =
                         (create (ts T ?k) creator
                                 ?new-id' new-ls));
             B3 := (!chain-last [A ==> (B1 | B2) [ioe-create]])}
          (!cases B3
            assume B1
              (!chain-last
               [true
                ==> (n >= n)  [N.lessEqual.reflexive]
                ==> (n >= n & A1 & A2 & A3 & B1)  [augment]
                ==> goal                          [existence]])
            assume B2
              let {C := (!chain-last [A ==> (goal | ~ B2)
                                        [fair-create]])}
```

```
                    (!cases C
                      assume goal
                        (!claim goal)
                      assume (~ B2)
                          (!from-complements goal B2 (~ B2))))
    | (val-of fair-compute-theorem) =>
        pick-any T n s ls new-ls
         let {A1 := ((config (ts T n)) = s ++ (actor computer ls));
              A2 := (ready-to (compute (ts T n) computer ls new-ls))}
          assume A := (A1 & A2)
             let {B1 := ((ts T (S n)) =
                         (compute (ts T n) computer ls new-ls));
                  B2 := (exists ?k ?s' ?ls' ?new-ls' .
                           ?k > n &
                           (config (ts T ?k)) =
                           ?s' ++ (actor computer ?ls') &
                           (ready-to (compute (ts T ?k) computer
                                              ?ls' ?new-ls')));
                  goal := (exists ?k ?s' ?ls' ?new-ls' .
                            ?k >= n &
                            (config (ts T ?k)) =
                            ?s' ++ (actor computer ?ls') &
                            (ready-to (compute (ts T ?k) computer
                                               ?ls' ?new-ls')) &
                            (ts T (S ?k)) =
                              (compute (ts T ?k) computer
                                       ?ls' ?new-ls'));
                  B3 := (!chain-last [A ==> (B1 | B2) [ioe-compute]])}
            (!cases B3
              assume B1
                (!chain-last
                 [true
                  ==> (n >= n)  [N.lessEqual.reflexive]
                  ==> (n >= n & A1 & A2 & B1)  [augment]
                  ==> goal                     [existence]])
              assume B2
                let {C := (!chain-last [A ==> (goal | ~ B2)
                                       [fair-compute]])}
                  (!cases C
                    assume goal
                      (!claim goal)
                    assume (~ B2)
                        (!from-complements goal B2 (~ B2))))
    }
(evolve Theory [theorems proofs])
} # module IOE-Fair-Transition-Sequence
```

E Clock system specification

Ticker can send tick messages to Clock1; it can do nothing else.

```
extend-module CLOCK-ACTORS {
module Ticker {
assert not-ready-to-receive :=
  (forall ?T ?ls ?fr ?c .
    ~ (ready-to (receive ?T Ticker ?ls ?fr ?c)))
assert ready-to-send :=
  (forall ?T ?to ?c . (ready-to (send ?T Ticker ?to ?c)))
assert not-ready-to-create :=
  (forall ?T ?ls ?new-id ?new-ls .
    ~ (ready-to (create ?T Ticker ?new-id ?new-ls)))
assert not-ready-to-compute :=
  (forall ?T ?ls ?new-ls .
    ~ (ready-to (compute ?T Ticker ?ls ?new-ls)))
assert make-receptive :=
  (forall ?ls . (make-receptive ?ls) = empty)}}
(!sym (Ticker =/= Clock1))
```

The Clock1 axioms are as follows:

```
extend-module CLOCK-ACTORS {
module Clock1 {
assert accept :=
  (forall ?t .
    (accept Clock1 (clocal Clock1 ?t) Ticker 'tick) =
                  (clocal Clock1 (S ?t)))
assert make-receptive :=
  (forall ?ls . (make-receptive ?ls) = ?ls)
assert ready-to-receive :=
  (forall ?T ?ls ?fr ?c .
    (ready-to (receive ?T Clock1 ?ls ?fr ?c)))
assert not-ready-to-send :=
  (forall ?T ?to ?c . ~ (ready-to (send ?T Clock1 ?to ?c)))
assert ready-to-create :=
  (forall ?T ?new-id ?new-ls .
    (ready-to (create ?T Clock1 ?new-id ?new-ls)))
assert not-ready-to-compute :=
  (forall ?T ?ls ?new-ls .
    ~ (ready-to (compute ?T Clock1 ?ls ?new-ls)))}}
declare Time: [Actor-Name (CFG (Actor Actor-Name CLS))] -> N
module Time {
assert read :=
  (forall ?id ?s ?t .
    (Time ?id (?s ++ (actor ?id (clocal ?id ?t)))) = ?t)}}
```

E.1 Proving infinitely-often-enabled

While we have presented fair transitions and infinitely-often-enabled (IOE) transitions in terms of axioms, our clock progress proofs ultimately should not rely on *assuming* these axioms, but rather we should *prove* that they hold based on the clock system implementation. We omit this exercise for the fair transition axioms, but we now show how to derive the needed IOE properties from the axioms given in the Ticker, Clock1, and Time modules. Rather than carry out these proofs just for this clock system, however, let us return to the abstract level to formulate a couple of small abstract theories, Simple-Sender and Simple-Receiver, in which (a) the axioms are satisfied by the Ticker and Clock1 implementations, and from which (b) the needed IOE properties can be proved. Here we show only axioms and theorems for Simple-Sender.

```
module Simple-Sender {
open-module INFINITELY-OFTEN-ENABLED

define never-ready-to-receive :=
  (forall ?T ?ls ?fr ?c .
    ~ (ready-to (receive ?T sender ?ls ?fr ?c)))

define ready-to-send :=
  (forall ?T ?to ?c .
    (ready-to (send ?T sender ?to ?c)))

define never-ready-to-create :=
  (forall ?T ?new-id ?new-ls .
    ~ (ready-to (create ?T sender ?new-id ?new-ls)))

define never-ready-to-compute :=
  (forall ?T ?ls ?new-ls .
    ~ (ready-to (compute ?T sender ?ls ?new-ls)))

define Theory :=
  (theory [TRANSITION-SEQUENCE.Theory]
    [never-ready-to-receive ready-to-send
     never-ready-to-create never-ready-to-compute]
  'Simple-Sender)

define ioe-send :=
  (forall ?T ?n ?s ?ls ?to ?c .
    (config (ts ?T ?n)) = ?s ++ (actor sender ?ls) &
      (ready-to (send (ts ?T ?n) sender ?to ?c))
      ==>
      (ts ?T (S ?n)) = (send (ts ?T ?n) sender ?to ?c) |
      (exists ?k ?s' ?ls' .
        ?k > ?n &
        (config (ts ?T ?k)) = ?s' ++ (actor sender ?ls') &
        (ready-to (send (ts ?T ?k) sender ?to ?c))))

define ioe-send-lemma :=
  (forall ?T':(TP 'Id 'LS) ?T ?n ?s ?ls:'LS ?to ?c .
```

```
    (ts ?T ?n) -->> ?T' & ?T' = (ts ?T (S ?n)) &
    (config (ts ?T ?n)) = ?s ++ (actor sender ?ls) &
    (ready-to (send (ts ?T ?n) sender ?to ?c))
    ==>
    (ts ?T (S ?n)) = (send (ts ?T ?n) sender ?to ?c) |
    (exists ?k ?s' ?ls' .
      ?k > ?n &
      (config (ts ?T ?k)) = ?s' ++ (actor sender ?ls') &
      (ready-to (send (ts ?T ?k) sender ?to ?c))))
```

```
define proofs :=
  method (theorem adapt)
   let {given := lambda (P) (get-property P adapt Theory);
        lemma := method (P) (!property P adapt Theory);
        chain := method (L) (!chain-help given L 'none);
        chain-last := method (L) (!chain-help given L 'last);
        ts := (adapt ts);
        sender := (adapt sender);
        ++A := (given ['++ Associative]);
        ++C := (given ['++ Commutative])}
   match theorem {
     (val-of ioe-send) =>
     pick-any T:(TP 'Id 'LS) n s ls:'LS to:'Id c
       let {A1 := ((config (ts T n)) = s ++ (actor sender ls));
            A2 := (ready-to (send (ts T n) sender to c))}
       assume (A1 & A2)
         let {B1 := (!chain-last
                     [true ==> ((ts T n) -->> (ts T S n))
                               [ts-directly-connected]]);
              B2 := (!reflex (ts T S n));
              ISL := (!lemma ioe-send-lemma)}
         (!chain-last
          [(B1 & B2 & A1 & A2) ==>
           ((ts T (S n)) =
            (send (ts T n) sender to c) |
            (exists ?k ?s' ?ls' .
               ?k > n &
               (config (ts T ?k)) =
               ?s' ++ (actor sender ?ls') &
               (ready-to (send (ts T ?k) sender to c)))) [ISL]])
   | (val-of ioe-send-lemma) =>
     datatype-cases (adapt theorem) {
       Initial =>
       pick-any T n s ls to c
         let {A1 := ((ts T n) -->> Initial);
              A2 := (Initial = (ts T S n));
              A3 := ((config (ts T n)) = s ++ (actor sender ls));
              A4 := (ready-to (send (ts T n) sender to c));
              RRI := (!lemma ['-->> RR+-inclusion])}
```

```
      assume (A1 & A2 & A3 & A4)
        (!from-complements
         ((ts T S n) = (send (ts T n) sender to c) |
          (exists ?k ?s' ?ls' .
            ?k > n &
            (config (ts T ?k)) =
              ?s' ++ (actor sender ?ls') &
            (ready-to (send (ts T ?k) sender to c))))
         (!chain-last [A1 ==> ((ts T n) -->>+ Initial)
                              [RRI]])
         (!chain-last [true ==> (~ ((ts T n) -->>+ Initial))
                              [nothing-leads-to-Initial]]))
    | (receive T' id' ls' fr' c') =>
      pick-any T n s ls to c
        let {A1 := ((ts T n) -->> (receive T' id' ls' fr' c'));
             A2 := ((receive T' id' ls' fr' c') = (ts T S n));
             A3 := ((config (ts T n)) = s ++ (actor sender ls));
             A4 := (ready-to (send (ts T n) sender to c))}
        assume (A1 & A2 & A3 & A4)
          let {
             goal' := (exists ?k ?s' ?ls' .
                        ?k > n &
                        (config (ts T ?k)) =
                        ?s' ++ (actor sender ?ls') &
                        (ready-to (send (ts T ?k) sender to c)));
             goal := ((ts T (S n)) =
                       (send (ts T n) sender to c) | goal');
             (and C1 C2) :=
             (!chain-last
              [A1 ==> ((ts T n) = T' &
                       (Enabled (receive (ts T n) id' ls' fr' c')))
                   [directly-leads-to-receive]]);
             C3 := (!chain-last
                     [C2 ==> ((Enabled (ts T n)) &
                              (exists ?s .
                                (config (ts T n)) =
                                ?s ++ (actor id' ls') ++
                                (message fr' id' c') &
                                (ready-to (receive (ts T n)
                                                   id' ls' fr' c'))))
                         [enabled-receive]])}
          pick-witness s1 for (!right-and C3) C3-w
            let {C3-w1 := ((config (ts T n)) =
                             s1 ++ (actor id' ls') ++
                             (message fr' id' c'));
                 C3-w2 := (ready-to (receive (ts T n)
                                             id' ls' fr' c'))}
            (!two-cases
             assume (sender = id')
               (!from-complements goal
```

```
    C3-w2
    (!chain-last
     [true ==>
           (~ (ready-to
               (receive (ts T n) sender
                                ls' fr' c')))
              [never-ready-to-receive]
           ==> (~ (ready-to
                   (receive (ts T n) id'
                            ls' fr' c')))
                   [(sender = id')]]))
assume (sender =/= id')
  let {UI2 := (!lemma unique-ids2);
       D0 := (!chain-last
              [(sender =/= id')
               ==> ((actor' sender ls) =/=
                    (actor' id' ls'))  [UI2]]);
       D1 := (!chain-last
              [true ==> ((actor' sender ls) =/=
                         (message' fr' id' c'))
              [(exclusive-constructors
                "Actor-Cfg.Actor")]]);
       MT := (!lemma More-Together);
       D2 := (!mp (!uspec* MT
                           [(config (ts T n)) s s1
                            (actor' sender ls)
                            (actor' id' ls')
                            (message' fr' id' c')])
                  (!both A3 (!both C3-w1
                                   (!both D0 D1))))}
        pick-witness s2 for D2 D2-w
         let {E1 :=
              (!chain-last
               [D2-w
                ==> ((config (ts T n))
                     = ((s2 ++ (actor sender ls))
                        ++ (actor id' ls') ++
                        (message fr' id' c')))
                                  [++A]]);
             E2 :=
              (!chain-last
               [(E1 & C3-w2)
                ==> ((config (receive (ts T n)
                                      id' ls' fr' c'))
                     = ((s2 ++ (actor sender ls)) ++
                        (actor
                         id' (accept id' ls' fr' c'))))
                [trans-receive]]);
             E3 := (!chain-last [true ==> ((S n) > n)
                                 [N.less.<S]]);
```

```
                                  E4 :=
                                   (!chain
                                    [(config (ts T (S n)))
                                     = (config (receive T' id' ls' fr' c'))
                                                                          [A2]
                                     = (config (receive (ts T n)
                                                        id' ls' fr' c')) [C1]
                                     = ((s2 ++ (actor sender ls)) ++
                                        (actor id' (accept id' ls' fr' c')))
                                     [E2]
                                     = ((s2 ++ (actor
                                                id' (accept id' ls' fr' c')))
                                        ++ (actor sender ls))
                                     [++A ++C]]);
                                  E5 := (!chain-last
                                         [true
                                          ==> (ready-to (send (ts T (S n))
                                                              sender to c))
                                              [ready-to-send]])}
                              (!chain-last
                               [(E3 & E4 & E5) ==> goal'   [existence]
                                               ==> goal    [augment]]))

      | (send T' fr' to' c') =>
        (!force (urep (rename (adapt theorem))
                      [(send T' fr' to' c')]))
      | (create T' id' new-id' new-ls') =>
        (!force (urep (rename (adapt theorem))
                      [(create T' id' new-id' new-ls')]))
      | (compute T' id' new-id' new-ls') =>
        (!force (urep (rename (adapt theorem))
                      [(compute T' id' new-id' new-ls')]))
      }
    } # match theorem
 (evolve Theory [[ioe-send ioe-send-lemma] proofs])
} # module Simple-Sender
```

We have omitted the proof of the send, create, and compute cases by using Athena's `force` method, which anoints its sentence argument as a theorem without requiring any proof.[5]

```
define Clock1-persistence :=
       (forall ?T ?T0 ?s0 ?t0 .
          (config ?T0) = ?s0 ++ (actor Clock1 (clocal Clock1 ?t0)) &
          ?T0 -->>* ?T &
          (unique-ids (config ?T0))
```

[5] Using `force` is obviously unsound, and all uses of `force` must be eliminated before proofs can be considered complete. On the other hand, using `force` during proof development enables one to proceed top-down, expressing first the gross structure of a proof, then filling in details later, by replacing uses of `force` with actual proofs.

```
          ==>
          (exists ?s ?t .
            (config ?T) = ?s ++ (actor Clock1 (clocal Clock1 ?t)) &
            ?t >= ?t0))

define Clock1-persistence-1 :=
       (forall ?T ?s ?T0 ?s0 ?t0 ?ls .
          (config ?T0) = ?s0 ++ (actor Clock1 (clocal Clock1 ?t0)) &
          ?T0 -->>* ?T &
          (config ?T) = ?s ++ (actor Clock1 ?ls) &
          (unique-ids (config ?T0))
          ==>
          (exists ?t . ?ls = (clocal Clock1 ?t) & ?t >= ?t0))
define Clock1-persistence-theorems :=
  [Clock1-persistence Clock1-persistence-1]
```

E.2 Proofs of the clock progress theorems

```
extend-module Fair-Clock-System {
define Clock1-progress-proof :=
  method (theorem adapt)
    let {given := lambda (P) (get-property P adapt Theory);
         lemma := method (P) (!property P adapt Theory);
         chain := method (L) (!chain-help given L 'none);
         chain-last := method (L) (!chain-help given L 'last);
         ++A := (given ['Clock ['++ Associative]]);
         ++C := (given ['Clock ['++ Commutative]])}
      by-induction theorem {
          zero =>
          pick-any T:(TP Actor-Name CLS) n0 s0 ls0 t0
            let {A1 := ((config (cts T n0)) =
                         s0 ++ (actor Ticker ls0) ++
                         (actor Clock1 (clocal Clock1 t0)));
                 A2 := (unique-ids (config (cts T n0)))}
              assume (A1 & A2)
                conclude goal :=
                  (exists ?n ?s ?ls:CLS ?u .
                    ?n >= n0 &
                    (config (cts T ?n)) =
                      ?s ++ (actor Ticker ?ls) ++
                      (actor Clock1 (clocal Clock1 ?u)) &
                    ?u >= zero)
                  let {C := (!chain-last
                              [true ==> (n0 >= n0)
                                    [N.lessEqual.reflexive]])}
                    (!chain-last
                     [true ==> (t0 >= zero)      [N.lessEqual.zero <=]
                           ==> (C & A1 & t0 >= zero) [augment]
```

```
                        ==> goal              [existence]])
    | (S t) =>
      let {ind-hyp :=
           (forall ?T ?n0 ?s0 ?ls0 ?t0 .
             (config (cts ?T ?n0)) =
                ?s0 ++ (actor Ticker ?ls0) ++
                (actor Clock1 (clocal Clock1 ?t0)) &
             (unique-ids (config (cts ?T ?n0)))
             ==>
             (exists ?n ?s ?ls ?u .
               ?n >= ?n0 &
               (config (cts ?T ?n)) =
                 ?s ++ (actor Ticker ?ls) ++
                 (actor Clock1 (clocal Clock1 ?u)) &
               ?u >= t))}
        pick-any T:(TP
                     Actor-Name CLS)
                     n0 s0 ls0:CLS t0
          let {A1 := ((config (cts T n0)) =
                       s0 ++ (actor Ticker ls0) ++
                       (actor Clock1 (clocal Clock1 t0)));
               A2 := (unique-ids (config (cts T n0)))}
            assume (A1 & A2)
              let {goal := (exists ?n ?s ?ls:CLS ?u .
                             ?n >= n0 &
                             (config (cts T ?n)) =
                               ?s ++ (actor Ticker ?ls) ++
                               (actor Clock1 (clocal Clock1 ?u)) &
                             ?u >= (S t));
                   B := (!chain-last
                          [(A1 & A2) ==>
                           (exists ?n1 ?s1 ?ls1 ?t1 .
                             ?n1 >= n0 &
                             (config (cts T ?n1)) =
                                ?s1 ++ (actor Ticker ?ls1) ++
                                (actor Clock1 (clocal Clock1 ?t1)) &
                             ?t1 >= t)                  [ind-hyp]])}
                pick-witnesses n1 s1 ls1 t1 for B
                  let {B-w0 := (n1 >= n0);
                       B-w1 :=
                         ((config (cts T n1)) =
                            s1 ++ (actor Ticker ls1) ++
                            (actor Clock1 (clocal Clock1 t1)));
                       B-w2 := (t1 >= t);
                       C1 := (!chain-last
                              [true
                               ==>
                               (ready-to (send (cts T n1) Ticker
                                               Clock1 'tick))
                               [Ticker.ready-to-send]]);
```

```
    C2 :=
      (!chain
       [(config (cts T n1))
        = (s1 ++ (actor Ticker ls1) ++
              (actor Clock1 (clocal Clock1 t1)))
        [B-w1]
        = ((s1 ++ (actor Clock1 (clocal Clock1 t1)))
           ++ (actor Ticker ls1))
        [++A ++C]]);
    FST := (!lemma ['Clock fair-send-theorem]);
    C3 := (!chain-last
           [(C2 & C1)
            ==>
     (exists ?n2 ?s2 ?ls2 .
        ?n2 >= n1 &
        (config (cts T ?n2)) =
        ?s2 ++ (actor Ticker ?ls2) &
        (ready-to (send (cts T ?n2) Ticker
                        Clock1 'tick)) &
        (cts T (S ?n2)) =
        (send (cts T ?n2) Ticker Clock1 'tick))
      [FST]])}
 pick-witnesses n2 s2 ls2 for C3
   let {
C3-w1 := (n2 >= n1);
C3-w2 :=
  ((config (cts T n2)) = s2 ++ (actor Ticker ls2));
C3-w3 := (ready-to (send (cts T n2) Ticker
                         Clock1 'tick));
C3-w4 := ((cts T (S n2)) =
                 (send (cts T n2) Ticker
                       Clock1 'tick));
D1 := (!chain-last
       [C3-w1 ==> ((cts T n1) -->>* (cts T n2))
                  [(given ['Clock ts-connected])]]);
D2 := (!chain-last
       [B-w0 ==> ((cts T n0) -->>* (cts T n1))
                 [(given ['Clock ts-connected])]]);
TCT3 := (!lemma ['Clock ['-->> TC-Transitivity3]]);
D3 := (!chain-last
       [(D2 & D1)  ==> ((cts T n0) -->>* (cts T n2))
        [TCT3]]);
D4 := (!chain-last
       [true ==> ((S n2) >= n2) [N.lessEqual.S3]
             ==> ((cts T n2) -->>* (cts T (S n2)))
             [(given ['Clock ts-connected])]]);
UIP := (!lemma ['Clock unique-ids-persistence]);
D5 := (!chain-last
       [(A2 & D2)
        ==> (unique-ids (config (cts T n1))) [UIP]]);
```

```
                    D6 := (!chain
                           [(config (cts T n1))
                            = (s1 ++ (actor Ticker ls1)
                                  ++ (actor Clock1 (clocal Clock1 t1)))
                            [B-w1]
                            = ((s1 ++ (actor Ticker ls1))
                               ++ (actor Clock1 (clocal Clock1 t1)))
                            [++A ++C]]);
                    CP := (!lemma Clock1-persistence);
                    D7 := (!chain-last
                           [(D6 & D1 & D5) ==>
                              (exists ?s3 ?t2 .
                                (config (cts T n2)) =
                                  ?s3 ++ (actor Clock1
                                                (clocal Clock1 ?t2)) &
                                ?t2 >= t1)         [CP]])}
                  pick-witnesses s3 t2 for D7
                    let {D7-w1 :=
                           ((config (cts T n2)) =
                            s3 ++ (actor Clock1 (clocal Clock1 t2)));
                         D7-w2 := (t2 >= t1);
                         IST := (!lemma ['Clock Together]);
                         UI2 := (!lemma ['Clock unique-ids2]);
                         E1 := (!chain-last
                                [(Clock1 =/= Ticker)
                                 ==> ((actor' Clock1 (clocal Clock1 t2))
                                      =/= (actor' Ticker ls2))       [UI2]
                                 ==> (D7-w1 & C3-w2 &
                                      (actor' Clock1 (clocal Clock1 t2))
                                      =/= (actor' Ticker ls2))
                                 [augment]
                                 ==> (exists ?s4 .
                                       (config (cts T n2)) =
                                       ?s4 ++ (actor Clock1
                                                     (clocal Clock1 t2))
                                       ++ (actor Ticker ls2))    [IST]])}
                      pick-witness s4 for E1 E1-w
                       let {F1 :=
                              (!chain-last
                               [(config (cts T n2))
                                = (s4 ++ (actor Clock1 (clocal Clock1 t2)) ++
                                      (actor Ticker ls2))
                                [E1-w]
                                = ((s4 ++ (actor Clock1 (clocal Clock1 t2))) ++
                                   (actor Ticker ls2))
                                [++A ++C]
                                ==> ((config (cts T n2)) =
                                     (s4 ++ (actor Clock1 (clocal Clock1 t2)))
                                     ++ (actor Ticker ls2) & C3-w3)
                                [augment]
```

```
      ==> ((config (send (cts T n2) Ticker
                         Clock1 'tick))
           = (s4 ++ (actor Clock1
                           (clocal Clock1 t2)))
           ++ (actor Ticker (make-receptive ls2))
           ++ (message Ticker Clock1 'tick))
      [(given ['Clock trans-send])]
      ==> ((config (cts T (S n2))) =
           (s4 ++ (actor Clock1 (clocal Clock1 t2)))
           ++ (actor Ticker (make-receptive ls2))
           ++ (message Ticker Clock1 'tick))
      [C3-w4]
      ==> ((config (cts T (S n2))) =
           (s4 ++ (actor Clock1 (clocal Clock1 t2)))
           ++ (actor Ticker empty)
           ++ (message Ticker Clock1 'tick))
      [Ticker.make-receptive]
      ==> ((config (cts T (S n2))) =
           ((s4 ++ (actor Ticker empty))
            ++ (actor  Clock1 (clocal Clock1 t2))
            ++ (message Ticker Clock1 'tick)))
      [++A ++C]]);
      F2 := (!chain-last
             [true
              ==>
              (ready-to (receive (cts T (S n2))
                                 Clock1
                                 (clocal Clock1 t2)
                                 Ticker 'tick))
              [Clock1.ready-to-receive]]);
      FRT := (!lemma ['Clock fair-receive-theorem]);
      F4 :=
      (!chain-last
       [(F1 & F2)
        ==>
        (exists ?n3 ?s5 ?ls5 .
          ?n3 >= (S n2) &
          (config (cts T ?n3)) =
           ?s5 ++ (actor Clock1 ?ls5) ++
           (message Ticker Clock1 'tick) &
          (ready-to (receive (cts T ?n3) Clock1
                             ?ls5 Ticker 'tick)) &
          (cts T (S ?n3)) =
          (receive (cts T ?n3) Clock1 ?ls5
                   Ticker 'tick))
        [FRT]])}
pick-witnesses n3 s5 ls5 for F4
 let {F4-w1 :=  (n3 >= (S n2));
      F4-w2a :=
        ((config (cts T n3)) =
```

```
      s5 ++ (actor Clock1 ls5)
         ++ (message Ticker Clock1 'tick));
F4-w2b := (ready-to (receive (cts T n3)
                             Clock1 ls5
                             Ticker 'tick));
F4-w3 :=
   ((cts T (S n3)) =
    (receive (cts T n3) Clock1 ls5
             Ticker 'tick));
G1 := (!chain-last
       [F4-w1 ==> ((S n3) >= (S n2))
                  [N.lessEqual.S2]]);
G2 := (!chain-last
       [G1 ==> ((cts T (S n2)) -->>*
                (cts T (S n3)))
           [(given ['Clock ts-connected])]]);
G3 := (!chain-last
       [(config (cts T n2))
        = (s4 ++ (actor Clock1
                        (clocal Clock1 t2))
              ++ (actor Ticker ls2))
        [E1-w]
        = ((s4 ++ (actor Ticker ls2))
           ++ (actor Clock1
                     (clocal Clock1 t2)))
        [++A ++C]]);
G4 := (!chain-last
       [true ==> ((S n2) >= n2)
                 [N.lessEqual.S3]]);
G5 := (!chain-last
       [true ==> (F4-w1 & (S n2) >= n2)
             [augment]
             ==> (n3 >= n2)
             [N.lessEqual.transitive]]);
G6 := (!chain-last
        [G5
         ==> ((cts T n2) -->>* (cts T n3))
         [(given ['Clock ts-connected])]]);
G7 := (!chain-last
       [(config (cts T n3))
        = (s5 ++ (actor Clock1 ls5)
              ++ (message Ticker
                          Clock1 'tick))
        [F4-w2a]
        = ((s5 ++ (message Ticker Clock1
                           'tick))
           ++ (actor Clock1 ls5))
        [++A ++C]]);
UIP := (!lemma ['Clock
                unique-ids-persistence]);
```

```
  G8 := (!chain-last
          [(A2 & D3) ==>
           (unique-ids (config (cts T n2)))
                                        [UIP]]);
  CP1 := (!lemma Clock1-persistence-1);
  G9 := (!chain-last
          [(G3 & G6 & G7 & G8)
          ==> (exists ?t .
                ls5 = (clocal Clock1 ?t) &
                ?t >= t2)
          [CP1]])}
pick-witness t3 for G9
  let
   {G9-w1 := (ls5 = (clocal Clock1 t3));
    G9-w2 := (t3 >= t2);
    H1 := (!chain-last
        [(F4-w2a & F4-w2b)
         ==> ((config (receive (cts T n3)
                               Clock1 ls5
                               Ticker 'tick))
              =
              s5 ++
              (actor Clock1
                     (accept Clock1 ls5
                             Ticker 'tick)))
        [(given ['Clock trans-receive])]
        ==> ((config (receive (cts T n3)
                              Clock1 ls5
                              Ticker 'tick))
             =
             s5 ++
             (actor Clock1
                    (clocal Clock1 (S t3))))
        [G9-w1 Clock1.accept]
        ==> ((config (cts T (S n3))) =
             s5 ++
             (actor Clock1
                    (clocal Clock1 (S t3))))
        [F4-w3]]);
     H2 := (!chain
           [(config (cts T (S n2)))
            = ((s4 ++ (actor Ticker empty))
               ++ (actor Clock1
                         (clocal Clock1 t2))
               ++ (message Ticker Clock1 'tick))
            [F1]
            = ((s4 ++ (actor Clock1
                             (clocal Clock1 t2))
                   ++ (message Ticker
                               Clock1 'tick))
```

```
                    ++ (actor Ticker empty))
                [++A ++C]]);
      H3 := (!chain-last
             [G8 ==> (G8 & D4)          [augment]
                  ==> (unique-ids
                       (config (cts T (S n2))))
                  [UIP]]);
      AP := (!lemma ['Clock actor-persistence]);
      H4 := (!chain-last
             [(H2 & G2 & H3) ==>
              (exists ?s6 ?ls6 .
                (config (cts T (S n3))) =
                ?s6 ++ (actor Ticker ?ls6))[AP]])}
   pick-witnesses s6 ls6 for H4 H4-w
     let {I1 :=
      (!chain-last
       [(Ticker =/= Clock1)
        ==> ((actor' Ticker ls6) =/=
             (actor' Clock1 (clocal Clock1
                                    (S t3))))
        [UI2]
        ==> (H4-w & H1 &
             (actor' Ticker ls6) =/=
             (actor' Clock1 (clocal Clock1
                                    (S t3))))
        [augment]
        ==> (exists ?s7 .
              (config (cts T (S n3)))
              = ?s7 ++ (actor Ticker ls6)
              ++ (actor Clock1
                        (clocal Clock1 (S t3))))
             [IST]])}
      pick-witness s7 for I1 I1-w
        let {J1 := (!chain-last
                    [true ==> ((S n2) >= n2)
                         [N.lessEqual.S3]]);
             J2 := (!chain-last
                    [true ==> ((S n3) >= n3)
                          [N.lessEqual.S3]]);
             J3 := (!combine-inequalities
                    [(n1 >= n0)        # B-w0
                     (n2 >= n1)        # C3-w1
                     ((S n2) >= n2)    # J1
                     (n3 >= (S n2))    # F4-w1
                     ((S n3) >= n3)]); # J2
             J4 := (!combine-inequalities
                    [(t1 >= t)         # B2-w
                     (t2 >= t1)        # D7-w2
                     (t3 >= t2)])}     # G9-w2
          (!chain-last
```

```
                                    [J4 ==> ((S t3) >= (S t))
                                                [N.lessEqual.injective]
                                        ==> (J3 & I1-w & (S t3) >= (S t))
                                                [augment]
                                        ==> goal  [existence]])
        } # by-induction

(evolve Theory
  [[Clock1-progress] Clock1-progress-proof])

define Clock1-Time-progress :=
  (forall ?t ?T:(TP Actor-Name CLS) ?n0 ?s0 ?ls0 ?t0 .
          (config (cts ?T ?n0) = ?s0 ++ (actor Ticker ?ls0)
            ++ (actor Clock1 (clocal Clock1 ?t0)) &
          (unique-ids (config (cts ?T ?n0)))
          ==>
          (exists ?n . ?n >= ?n0 &
                       (Time Clock1 (config (cts ?T ?n))) >= ?t)))

define Clock1-Time-progress-proof :=
  method (theorem adapt)
    let {given := lambda (P) (get-property P adapt Theory);
         lemma := method (P) (!property P adapt Theory);
         chain := method (L) (!chain-help given L 'none);
         chain-last := method (L) (!chain-help given L 'last);
         ++A := (given ['Clock ['++ Associative]]);
         ++C := (given ['Clock ['++ Commutative]])}
        pick-any t T:(TP Actor-Name CLS) n0 s0 ls0 t0
          assume A := ((config (cts T n0)) = s0 ++ (actor Ticker ls0) ++
                       (actor Clock1 (clocal Clock1 t0)) &
                       (unique-ids (config (cts T n0))))
            let {CPD := (!lemma Clock1-progress);
                 B := (!chain-last
                       [A ==> (exists ?n ?s ?ls ?u .
                                ?n >= n0 &
                                (config (cts T ?n)) =
                                ?s ++ (actor Ticker ?ls)
                                   ++ (actor Clock1 (clocal Clock1 ?u)) &
                                  ?u >= t)
                           [CPD]])}
             pick-witnesses n s ls u for B
               let {w0 := (n >= n0);
                    w1 := ((config (cts T n)) =
                           s ++ (actor Ticker ls) ++
                           (actor Clock1 (clocal Clock1 u)));
                    w2 := (u >= t);
                    C := (!chain
                          [(Time Clock1 (config (cts T n)))
                           = (Time Clock1
                                   (s ++ (actor Ticker ls)
```

```
                                        ++ (actor Clock1 (clocal Clock1 u))))
                          [w1]
                          = (Time Clock1
                                  ((s ++ (actor Ticker ls))
                                   ++ (actor Clock1 (clocal Clock1 u))))
                          [++A ++C]
                          = u   [Time.read]])}
                 (!chain-last
                  [w2
                   ==> ((Time Clock1 (config (cts T n))) >= t) [C]
                   ==> (w0 & (Time Clock1 (config (cts T n))) >= t)
                   [augment]
                   ==> (exists ?n .
                         ?n >= n0 &
                         (Time Clock1 (config (cts T ?n))) >= t)
                   [existence]])

(evolve Theory [[Clock1-Time-progress] Clock1-Time-progress-proof])
} # module FAIR-CLOCK-SYSTEM
```

E.3 Proofs of the clock persistence theorems

The proofs of the `Clock1-persistence-theorems` is by induction on transition paths. We have not done these proofs yet; we would like to first try to identify patterns in the proofs of the transition induction proofs we have previously done in order to develop methods encapsulating the patterns. A simple example of this approach was used in proving the four "leads-to" theorems in Appendix D.4. Using such methods will allow significantly shortening the existing proofs and easing the development of new ones.

E.4 Testing the proofs

To set up and test the proofs of the clock progress theorems, we first assert the FAIR-TRANSITION-SEQUENCE axioms, adapted for Clock-Actors:

```
open-module FAIR-CLOCK-SYSTEM
assert (theory-axioms [FAIR-TRANSITION-SEQUENCE.Theory 'Clock CA])
```

Next, we define a clock adaptation of the `Simple-Sender` theory:

```
define Ticker-Is-Simple := (theory [[Simple-Sender.Theory 'SS CA]]
                                   [] 'Ticker-Is-Simple)
```

But we don't need to assert the `Ticker-Is-Simple` axioms since they are consequences of the Ticker implementation.

Now we can prove the `ioe-send` property for Ticker:

```
(!property-test ['SS ioe-send] no-renaming Ticker-Is-Simple)
```

Here we just assert the `ioe-receive` property for Clock1 receiving; its proof would be similar to that of `ioe-send`.

```
assert (get-property ['Clock ioe-receive] no-renaming Theory)
```

We now have all we need to prove the Clock progress theorems:

```
(!property-test Clock1-progress no-renaming FAIR-CLOCK-SYSTEM)
(!property-test Clock1-Time-progress no-renaming FAIR-CLOCK-SYSTEM)
```

F Exercising the clock system: a sample sequence of transitions

Before attempting general proofs, it's useful to exercise a specification with symbolic execution to gain assurance that the system behaves as expected, in at least a few sample execution sequences. Here we show one such sample sequence of transitions of the Ticker-Clock system. The complexity of the specification does not permit automatic symbolic execution, but we can manually step through the transition sequence using simple proof steps.

We begin with setting up a sample sequence of transitions:

```
open-module CLOCK-ACTORS
open-module TRANSITION-PATH
assert (theory-axioms TRANSITION-PATH.Theory)
define T0 := Initial:(TP Actor-Name CLS)
define T1 := (create T0 Clock1 Ticker empty)
define T2 := (send T1 Ticker Clock1 'tick)
define T3 := (receive T2 Clock1 (clocal Clock1 zero) Ticker 'tick)
define T4 := (send T3 Ticker Clock1 'tick)
define T5 := (send T4 Ticker Clock1 'tick)
define T6 := (receive T5 Clock1 (clocal Clock1 (S zero)) Ticker 'tick)
```

Next we declare an arbitrary configuration `s0` and state as a theorem the implication we expect to hold about the relation between `(config T0)` and `(config T6)`:

```
declare s0: (CFG (Actor Actor-Name CLS));
define T0->T6 :=
    ((config T0) = s0 ++ (actor Clock1 (clocal Clock1 zero)) &
     (ready-to (create T0 Clock1 Ticker empty)) &
     (unique-ids (config T0) ++ (actor Ticker empty))
     ==> ((config T6) =
          s0 ++ (actor Clock1 (clocal Clock1 (S (S zero)))) ++
          (actor Ticker empty) ++
          (message Ticker Clock1 'tick)))
```

The following function takes a transition `T` and returns a method that can compute a justification in an implication chain (for applying a transition implication in a link in the chain).

```
define trans :=
  lambda (T)
    method (premise goal)
```

```
    let {given := lambda (P)
                   (get-property P no-renaming TRANSITION-PATH.Theory);
         chain-last := method (L) (!chain-help given L 'last);
         ++A := (given ['++ Associative]);
         ++C := (given ['++ Commutative])}
     match T {
       (send T0 fr to c) =>
          let {R := (!chain-last
                     [true ==> (ready-to (send T0 fr to c))
                           [Ticker.ready-to-send]])}
            (!chain-last
             [(premise & R) ==> goal [trans-send]])
     | (receive T0 id ls fr c) =>
         match premise {
           (= (config T0)
              (++ s (++ (Single (actor' to ls))
                        (Single (message' fr id c))))) =>
            let {R := (!chain-last
                       [true ==> (ready-to (receive T0 id ls fr c))
                             [Clock1.ready-to-receive]])}
              (!chain-last
               [(premise & R) ==> goal [trans-receive]])
         }
      }
```

Now we can step through the transition sequence in an implication chain:

```
conclude T0->T6
  let {given := lambda (P)
                  (get-property P no-renaming TRANSITION-PATH.Theory);
       chain := method (L) (!chain-help given L 'none);
       ++A := (given ['++ Associative]);
       ++C := (given ['++ Commutative]);
       A1 := ((config T0) = s0 ++ (actor Clock1 (clocal Clock1 zero)));
       A2 := (ready-to (create T0 Clock1 Ticker empty));
       A3 := (unique-ids (config T0) ++ (actor Ticker empty))}
    (!chain
     [(A1 & A2 & A3)
       ==> ((config T1) =
            s0 ++ (actor Clock1 (make-receptive (clocal Clock1 zero))) ++
            (actor Ticker empty))
       [trans-create]
       ==> ((config T1) =
            (s0 ++ (actor Clock1 (clocal Clock1 zero))) ++
            (actor Ticker empty))
       [Clock1.make-receptive ++A]
       ==> ((config T2) =
            (s0 ++ (actor Clock1 (clocal Clock1 zero))) ++
            (actor Ticker (make-receptive empty)) ++
            (message Ticker Clock1 'tick))
```

```
[(trans T2)]
==> ((config T2) =
     (s0 ++ (actor Ticker empty)) ++
     (actor Clock1 (clocal Clock1 zero)) ++
     (message Ticker Clock1 'tick))
[Ticker.make-receptive ++A ++C]
==> ((config T3) =
     (s0 ++ (actor Ticker empty)) ++
     (actor Clock1 (accept Clock1 (clocal Clock1 zero)
                           Ticker 'tick)))
[(trans T3)]
==> ((config T3) =
     (s0 ++ (actor Clock1 (clocal Clock1 (S zero)))) ++
     (actor Ticker empty))
[Clock1.accept ++A ++C]
==> ((config T4) =
     (s0 ++ (actor Clock1 (clocal Clock1 (S zero)))) ++
     (actor Ticker (make-receptive empty)) ++
     (message Ticker Clock1 'tick))
[(trans T4)]
==> ((config T4) =
     (s0 ++ (actor Clock1 (clocal Clock1 (S zero))) ++
     (message Ticker Clock1 'tick)) ++
     (actor Ticker empty))
[Ticker.make-receptive ++A ++C]
==> ((config T5) =
     (s0 ++ (actor Clock1 (clocal Clock1 (S zero))) ++
     (message Ticker Clock1 'tick)) ++
     (actor Ticker (make-receptive empty)) ++
     (message Ticker Clock1 'tick))
[(trans T5)]
==> ((config T5) =
     (s0 ++ (actor Ticker empty) ++
     (message Ticker Clock1 'tick)) ++
     (actor Clock1 (clocal Clock1 (S zero))) ++
     (message Ticker Clock1 'tick))
[Ticker.make-receptive ++A ++C]
==> ((config T6) =
     (s0 ++ (actor Ticker empty) ++
     (message Ticker Clock1 'tick)) ++
     (actor Clock1 (accept Clock1 (clocal Clock1 (S zero))
                           Ticker 'tick)))
[(trans T6)]
==> ((config T6) =
     s0 ++ (actor Clock1 (clocal Clock1 (S (S zero)))) ++
     (actor Ticker empty) ++ (message Ticker Clock1 'tick))
[Clock1.accept ++A ++C]])
```