PROVING CORRECTNESS OF ACTOR SYSTEMS USING FIFO COMMUNICATION

By

Ian W. Dunn

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Major Subject: COMPUTER SCIENCE

Examining Committee:

____

Carlos Varela, Thesis Adviser

____

Charles Stewart, Member

____

Ana Milanova, Member

Rensselaer Polytechnic Institute
Troy, New York

April 2014
(For Graduation May 2014)

CONTENTS

LIST OF FIGURES

ABSTRACT

We are developing a hierarchy of theories to reason about actor systems, with the ability to reuse proofs formalized at an abstract level in reasoning about concrete actor programs. Several actor languages, e.g., the ABCL family of languages, implement First-In First-Out (FIFO) communication between actors. Furthermore, many practical systems require FIFO communication for correctness. In previous work, Musser and Varela formalized properties including monotonicity of actor local states, guaranteed message delivery, and general con- sequences of fairness. While the actor model requires fairness, it does not require FIFO communication.

In this thesis, we extend the actor reasoning framework to enable proving correctness of systems which require FIFO communication. This is done by extending the actor framework within the Athena proof system, in which proofs are both human-readable and machine- checkable, taking advantage of its library of algebraic and relational theories.

We introduce three new theories into the actor model framework of Athena. All three of these theories are developed at the abstract level, enabling the use of them in many concrete programs. The first two of these theories introduce sequence numbers into the messages passed between actors, one for sending and one for receiving. We take advantage of the monotonicity of actor transitions to show that send sequence numbers and receive sequence numbers will only ever increase.

The third new theory begins to prove the ordering of messages given an order of the sequence numbers. We use results from the first two theories to show that if two messages are about to be sent or received, then the order in which the messages are sent or received is dictated by the sequence numbers. We then use that result to show that two messages must be received in the same order in which they were sent.

We continue on to show an example of an actor system, based on the computation of the Sieve of Eratosthenes, that requires FIFO communication in order to be able to prove correctness of its computation.

1. Introduction and Motivation

The actor model of computation [1], [2] can be useful as both a theoretical framework upon which it is possible to reason about concurrent computations [3], [4], as well as a practical means of building a distributed system [5], [6]. Actors concurrently encapsulate state, which makes them an intuitive choice for distributed computations [7]. Each actor must have a unique identifier and communicate with other actors through asynchronous message passing. Depending on its defined behavior, upon receiving a message an actor may change its state, send a message to another actor, and/or create a new actor.

Crucial to actor model computations is the ability to reason about them. These so called “actor theories” formalize the transitions by which each actor progresses from state to state, and the configurations that each point in the transition holds. The actor model imposes *fairness* on a progression of computations in order to be valid. Fairness means that if a transition (from an actor configuration) is infinitely often enabled, the transition must eventually happen [8].

FIFO communication further restricts valid computations to ensure that asynchronous messages between any two actors arrive in the order they were sent. While FIFO communication is not a requirement of the actor model of computation, several languages, e.g., the ABCL family of languages [9], [10] impose it to facilitate reasoning about and developing practical actor programs. Other actor languages, e.g., SALSA [6], require FIFO communication to be implemented at the application-level.

Actor languages can use different models for representing sequential computation within an actor. Agha, Mason, Smith, and Talcott use the untyped call-by-value lambda calculus to represent an actor's internal behavior [3]. Varela and Agha use an object's instance and class to represent an actor's state and its behavior [6]. This work follows on with Musser and Varela's approach, in which behavior within an actor is defined as axioms on certain functions and relations on their local states [8].

Our research goal is to develop a rich hierarchy of reusable theories modeling abstract actor computation, that is, following the actor model's operational semantics, to be able to derive general theorems that apply to all concrete actor programs that follow the model. In this thesis, we enrich the reasoning theory hierarchy to model systems which follow FIFO communication between actors.

The direction we take in this thesis is based around loose message ordering. In other words, we focus on a message arriving before another, but with the possibility of any number of messages arriving between the two.

1.1 Example: Sieve of Eratosthenes

As a motivating example of this thesis, we look at the computation given by the Sieve of Eratosthenes.

The Sieve of Eratosthenes is a basic computation developed to determine prime numbers. It begins by filtering out all multiples of two. Then, it takes the next value, three, and filters out all multiples of three. The computation continues this up to a given point, and returns all of the remaining numbers. Since no other numbers divide these, these must be prime numbers. It is intuitive that the Sieve computation is correct if every number that is returned is a prime number.

In an actor model representation of the Sieve of Eratosthenes, there would be three actors at the start: an *Input* actor, an *Output* actor, and an actor that represents the number two, called *S2*. The *Input* actor sends each value starting at three to the *S2* actor, which will filter out each value that it divides. When it receives a value that it does not divide, then it will pass it down to the next actor, creating a new one if it is at the end of the chain. Upon creation, a new actor will send the value that was used to create it to the *Output* actor. A graphical representation of the Sieve of Eratosthenes is given in Figure 1.1.

However, assume that the *Input* actor sends three, four, five, six, seven, eight, and nine to the *S2* actor. The *S2* actor will surely receive all of them, assuming guaranteed message delivery, but there is no guarantee that the *S2* actor will receive the number nine after it receives the number three. If this ends up being the case, then the *S2* actor will create the *S9* actor, which will then send the number nine to the *Output* actor. In the Athena proof system, this would be represented as follows:

```
(Initial then (send S2 Output (sm two))
         then (receive Output S2 (sm two))
         then (send Input S2 (sm three))
```

```
          then (send Input S2 (sm four))
          then (send Input S2 (sm five))
          then (send Input S2 (sm six))
          then (send Input S2 (sm seven))
          then (send Input S2 (sm eight))
          then (send Input S2 (sm nine))
          then (receive S2 Input (sm nine))
          then (create S2 (si nine))
          then (send (si nine) Output (sm nine))
          then (receive Output (si nine) (sm nine)))
```

After this happens, the *Output* actor would return the number nine as a prime number, which is incorrect. Therefore, in order for this computation to be correct, there must be some means of ensuring that the number three will reach the *S2* actor before the number nine reaches the *S2* actor.

In this thesis, we first discuss the current foundation of actor model proofs that are developed in the Athena proof system. We go on to extend this foundation to include a means of ensuring FIFO communication, and then prove that the foundation for FIFO communication ensures that messages are received in the correct order. We finally show an example, based upon the Sieve of Eratosthenes, of a computation that can be incorrect without FIFO communication, but will always be correct when

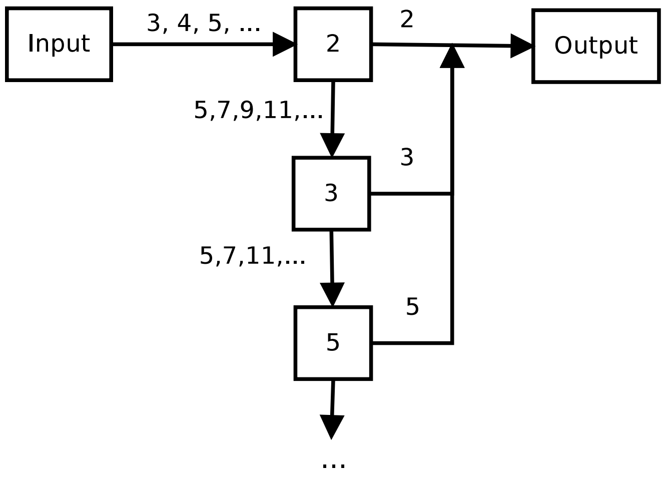

Figure 1.1: An Actor Model Representation of the Sieve of Eratosthenes. The Input actor continuously sends the numbers in increasing order, which are filtered out by the intermediate actors, which send their associated values to the Output actor.

FIFO communication is introduced.

2. Actor Proofs and Athena

The proofs regarding FIFO communications are built upon an existing library of formal proofs using the program Athena. Athena uses a proof system that is both human-readable and machine checkable, in an effort to improve readability while still allowing it to be easily used. The fact that the proofs are human-readable allows them to be easily used for educational purposes.

In this section, we will explore the existing actor proof hierarchy. We will give some of the basic definitions of actors in the Athena proof language, and show some currently existing lemmas on which we will be building our framework. The current actor model reasoning framework is shown in Figure 2.1.

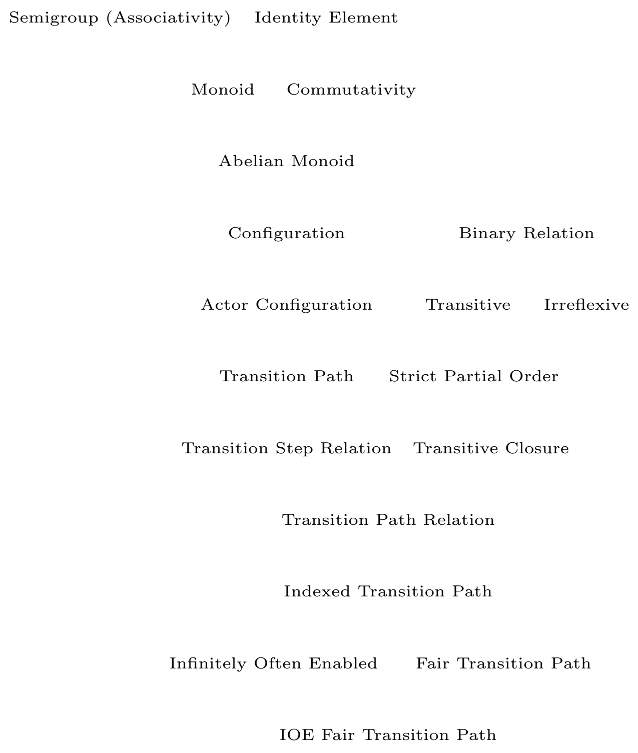

Figure 2.1: Algebraic, Relational, and Actor Theories. Arrows represent theory refinement.

2.1 Configurations

First, we formally define *configurations*. A configuration represents a multiset that holds specific information that pertains to the current point in a computation.

Configurations are defined as a polymorphic structure with three different constructors: *Null*, *One*, and *++*:

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

Null is the default constructor which represents an empty configuration. *One* is a constructor for a configuration containing a single object. Configurations are combined into a new configuration by means of the *++* constructor. The *++* constructor is defined to be associative and commutative, two properties that are used several times in the proofs concerning FIFO.

2.2 Actors

Now that configurations have been defined, the actors themselves must be defined as well.

Actor configurations will hold two types of objects: *actors* and *messages*. Like configurations, actors are formally defined as a polymorphic structure, this time allowing for an ID sort and a local state sort.

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

Note that *Ide* is a sort of quoted string in Athena, short for "Identifier". However, these constructors cannot be used directly in configurations, so there must be functions that use the *One* constructor from the configuration objects:

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

A commonly used predicate for the actors is the *unique-ids* predicate, which claims that the actor state relationship is a function, i.e. each identifier within a configuration has at most one associated local state. There are many proofs about *unique-ids*, however the most commonly used one in the FIFO proofs is:

```
 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
    ==> ?s1 = ?s2 & ?ls1 = ?ls2)
```

What this lemma is claiming is that if a configuration has two different representations, and the ids are the same of both actors in those configurations, then both the local states and the additional configurations must be the same.

2.3 Transitions

Next, actor *transitions* are defined. The three types of actor transitions are *send*, *receive*, and *create*. A *send* transition is made when an actor sends a message, a *receive* transition is made when and actor receives a message, and a *create* transition is made when an actor creates another actor:

```
datatype (Step Id) :=
  (receive Id Id Ide) | (send Id Id Ide) | (create Id Id)
```

After this, a means of reasoning about a series of transitions is required. This takes the form of *transition paths*. Transition paths are a sequence of transitions, beginning at *Initial*, and proceeding along through different steps:

```
datatype (TP Id LS) := Initial | (then (TP Id LS) (Step Id))
```

Another useful predicate that is given in the actor model framework is the "directly leads to" predicate. This is represented as "—>>" in Athena. This is a transitive relation, and built upon this are the predicates "—>>+" and "—>>*" predicates, which are the irreflexive and reflexive predicates for "leads to", respectively.

```
declare -->>: (Id, LS) [(TP Id LS) (TP Id LS)] -> Boolean
declare -->>+, -->>*:
 (Id, LS) [(TP Id LS) (TP Id LS)] -> Boolean
declare -->>**:
 (Id, LS) [N (TP Id LS) (TP Id LS)] -> Boolean
```

One of the last definitions gives a means of indexing transition paths. These *indexed transition paths* are defined through the function *itp*:

```
declare itp: (Id, State) [(TP Id State) N] -> (TP Id State)
```

2.4 Fairness and Progress

As we mentioned earlier, fairness is the property that if a transition is infinitely often enabled, then it must eventually occur. Athena builds upon this, and adds two lemmas, called *receive-happens* and *send-happens*.

```
define send-happens :=
(forall ?T ?n ?s ?ls ?c .
 (unique-ids config (itp ?T ?n)) &
 config (itp ?T ?n) = ?s ++ (actor sender ?ls) &
 ?ls ready-to (send sender receiver ?c)
 ==>
 exists ?k ?s' ?ls' .
  ?k >= ?n &
  config (itp ?T ?k) = ?s' ++ (actor sender ?ls') &
  ?ls' ready-to (send sender receiver ?c) &
  (itp ?T (S ?k)) = (itp ?T ?k) then (send sender receiver ?c))
```

Building on this, we add an axiom to be used with progress called *eventually-ready-to-receive*:

```
define eventually-ready-to-receive :=
(forall ?T ?n0 ?s0 ?ls0 ?c .
  config (itp ?T ?n0) =
  ?s0 ++ (actor receiver ?ls0)
      ++ (message sender receiver ?c) &
  (unique-ids config (itp ?T ?n0))
  ==>
  exists ?n ?s ?ls .
   ?n >= ?n0 &
   config (itp ?T ?n) =
     ?s ++ (actor receiver ?ls)
        ++ (message sender receiver ?c) &
     ?ls ready-to (receive receiver sender ?c))
```

A consequence of fairness and *eventually-ready-to-receive* is *guaranteed message delivery*, which is the property when a message is about to be sent, there must exist a point later on during which that message is received.

```
define guaranteed-message-delivery :=
(forall ?T ?n0 ?s0 ?ls0 ?ls1 ?c .
 config (itp ?T ?n0) =
   ?s0 ++ (actor sender ?ls0) ++ (actor receiver ?ls1) &
 ?ls0 ready-to (send sender receiver ?c) &
 (unique-ids config (itp ?T ?n0))
 ==>
 exists ?n ?s ?ls2 ?ls3 .
  ?n >= ?n0 &
  config (itp ?T ?n) =
   (?s ++ (actor sender ?ls2)) ++ (actor receiver ?ls3) ++
    (message sender receiver ?c) &
  ?ls3 ready-to (receive receiver sender ?c) &
  (itp ?T (S ?n)) = (itp ?T ?n) then (receive receiver sender ?c))
```

2.5 Monotonicity of Transitions

Another frequently used result is the *monotonicity of actor transitions*. This means that for any predicate R, if that predicate holds for a local state and its next local state, then if a transition leads to another, then R must hold for the local state in the first transition and the local state in the second transition, and any transition after that on:

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

3. Defining FIFO

Guaranteed message delivery means that messages reach their intended destination, but not necessarily that they reach it in the order in which they were sent. While for some computations this is not a problem, we have shown that for some others such as the Sieve of Eratosthenes, it can cause the computation to be incorrect.

To obtain FIFO communication between a sender and receiver, the sender can add a sequence number to each message it sends to the receiver, with the receiver maintaining an expected sequence number for any message it receives from that sender. Then the receiver only accepts a message from that sender if the sequence number in the message matches the expected one. While this is a well-known technique for achieving FIFO order, there are many details to be worked out to ensure that it operates correctly. By carrying out the specification and proof at an abstract level, as is done in the following sections, it is possible to reduce the task of ensuring correctness for a concrete application to checking that a few simple axioms are satisfied in that application.

All of the following development is placed within three new theories: *Send-Sequencing*, *Receive-Sequencing*, and *Fifo*. The relationship between these three theories and the rest of the actor model reasoning framework in Athena is shown in Figure 3.1.

3.1 Messages

To add sequence numbers to messages, the function *msg* is introduced that simply wraps up a sequence number and a message together so they can be treated as a message in the actor model. This allows actors to send sequence numbers in addition to the actual payload of a message.

For sequence numbers the natural numbers are used. Recall that *Ide*, a predefined quoted string type in Athena, is the type that messages are assumed to have in the actor model.

```
declare msg: [N Ide] -> Ide
```

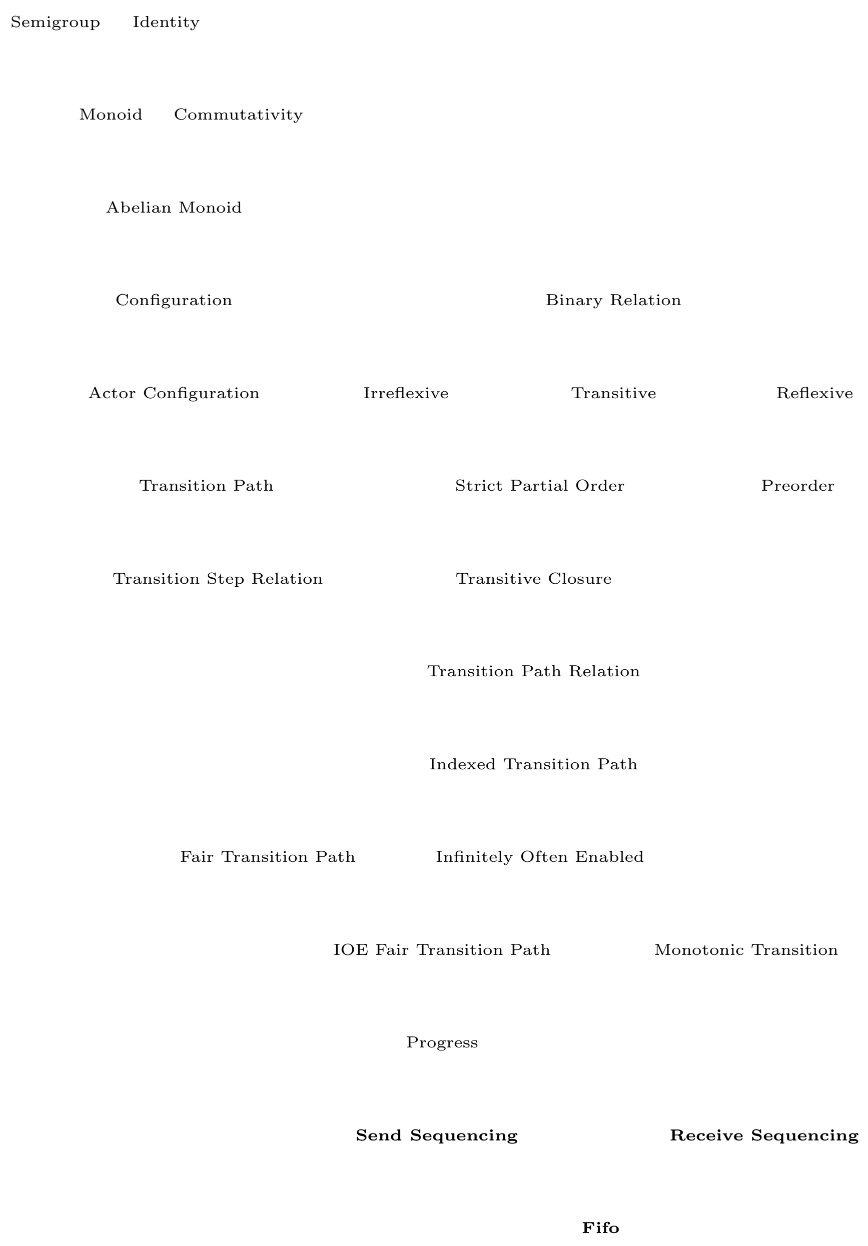

Figure 3.1: Algebraic, Relational, and Actor Theories with FIFO Theories. Rectangular nodes are algebraic theories, diamonds are relational theories, and ovals are actor theories. Bold nodes represent the theories defined in this thesis.

```
assert msg-injective :=
  (forall ?x0 ?y0 ?x1 ?y1 .
     (msg ?x0 ?y0) = (msg ?x1 ?y1) ==> ?x0 = ?x1 & ?y0 = ?y1)
```

We assume no other properties of the *msg* function other than its type and that it is injective, as is required in some of the later proofs.

In the same spirit, we specify how actors can keep track of sequence numbers via functions *send-seq* and *recv-seq*.

```
declare send-seq, recv-seq: (Id, LS) [(Cfg (Actor Id LS)) Id] -> N
```

with the intended meaning that *(send-seq (actor a ls) b)* is the sequence number that actor *a* uses for sending a message to actor *b*. This is also referred to as the send sequence number of actor *a* with respect to *b*. We formally specify the meaning of *send-seq* with axioms that together provide an inductive definition. Any way of implementing the function that satisfies the axioms is acceptable. For each *(a, b)* pair, only the local state of the actor *a* varies; i.e., the sequence number (or some way of computing it) must be embedded in *a*'s local state.

The base case of the inductive definition is when an actor is created, or when the computation begins; in either case we specify the sequence number as zero:

```
define send-seq-base :=
  (forall ?a ?b ?lsb .
    (send-seq (actor ?a (new-ls ?lsb)) ?b) = zero)
define send-seq-init :=
  (forall ?a ?b ?lsa .
    (exists ?s . config Initial = ?s ++ (actor ?a ?lsa))
    ==> (send-seq (actor ?a ?lsa) ?b) = zero)
```

For the inductive step, the *send-seq* value is incremented when an actor sends a message to another actor, but only then:

```
define send-seq-rec :=
  (forall ?a ?b ?lsa ?op .
    (exists ?c . ?op = (send ?a ?b ?c)) ==>
    (send-seq (actor ?a (next ?lsa ?op)) ?b)
      = (S (send-seq (actor ?a ?lsa) ?b)))
define send-seq-non-send :=
  (forall ?a ?b ?lsa ?op .
    ~ (exists ?c . ?op = (send ?a ?b ?c)) ==>
    (send-seq (actor ?a (next ?lsa ?op)) ?b)
        = (send-seq (actor ?a ?lsa) ?b))
```

The definitions pertaining to the receive sequence numbers are the same as for the send sequence numbers, but using *receive* instead of *send*:

```
define recv-seq-base :=
 (forall ?a ?b ?lsb .
   (recv-seq (actor ?a (new-ls ?lsb)) ?b) =
    zero)
define recv-seq-init :=
 (forall ?a ?b ?lsa .
  (exists ?s .
    (config Initial = ?s ++ (actor ?a ?lsa)))
    ==> (recv-seq (actor ?a ?lsa) ?b) = zero)
define recv-seq-rec :=
 (forall ?a ?b ?lsa ?op .
   (exists ?c .
     ?op = (receive ?a ?b ?c)) ==>
   (recv-seq (actor ?a (next ?lsa ?op)) ?b) =
   (S (recv-seq (actor ?a ?lsa) ?b)))
define recv-seq-non-recv :=
 (forall ?a ?b ?lsa ?op .
   ~(exists ?c .
      ?op = (receive ?a ?b ?c)) ==>
   (recv-seq (actor ?a (next ?lsa ?op)) ?b)  =
   (recv-seq (actor ?a ?lsa) ?b))
```

3.2 Specifying *Ready-to*

Each actor should only be ready to make a send or receive transition if the sequence number matches up correctly. For example, an actor a is only ever ready to receive a message from sender b if the receive sequence number actor a holds for sender b matches the sequence number of the message.

Since an actor implementer may have originally specified a *ready-to* predicate without concern for FIFO communication, how can we layer on this extra requirement of matching sequence numbers? The solution we use is to rename the given *ready-to* predicate as *ready-to-pred*, and then define *ready-to* in terms of *ready-to-pred* and the new sequence number requirements.

```
declare ready-to-pred: (Id,LS) [LS (Step Id)] -> Boolean

define ready-to-send-def :=
 (forall ?id ?ls ?to ?c .
  ?ls ready-to (send ?id ?to ?c)
  <==>
  (exists ?ct .
   ?c = (msg (send-seq (actor ?id ?ls) ?to) ?ct) &
   (?ls ready-to-pred (send ?id ?to ?ct))))

define ready-to-recv-def :=
 (forall ?id ?ls ?fr ?c .
  ?ls ready-to (receive ?id ?fr ?c)
  <==>
  (exists ?ct .
   ?c = (msg (recv-seq (actor ?id ?ls) ?fr) ?ct) &
   (?ls ready-to-pred (receive ?id ?fr ?ct))))

define ready-to-create-def :=
 (forall ?id ?ls ?new .
  ?ls ready-to (create ?id ?new)
  <==>
  (?ls ready-to-pred (create ?id ?new)))
```

3.3 Message Ordering

Next, a concept of a message being sent or received before another is required. This builds upon the *leads-to* predicates, but is set up around the messages instead of the transitions themselves.

First, we define a predicate that determines whether or not an actor is "about to" send or receive a message. In terms of the actor model, this refers to a point in an indexed transition path that has a configuration with an actor being ready to send or receive a message, and the next point in the transition path is that actual transition.

```
declare about-to-send, about-to-receive: (Id,LS)
```

```
 [(TP Id LS) N (Cfg (Actor Id LS)) Id Id LS LS N Ide] -> Boolean

define about-to-send-def :=
 (forall ?T ?n ?s ?a ?b ?lsa ?lsb ?x ?c .
  (about-to-send ?T ?n ?s ?a ?b ?lsa ?lsb ?x ?c)
  <==>
  config (itp ?T ?n) =
    ?s ++ (actor ?a ?lsa) ++ (actor ?b ?lsb) &
  ?lsa ready-to (send ?a ?b (msg ?x ?c)) &
  (itp ?T (S ?n)) = (itp ?T ?n) then (send ?a ?b (msg ?x ?c)))

define about-to-receive-def :=
 (forall ?t ?n ?s ?a ?b ?lsa ?lsb ?x ?c .
   (about-to-receive ?t ?n ?s ?a ?b ?lsa ?lsb ?x ?c)
   <==>
   (config (itp ?t ?n) =
     ?s ++ (actor ?a ?lsa)
        ++ (actor ?b ?lsb)
        ++ (message ?b ?a (msg ?x ?c)) &
    ?lsa ready-to (receive ?a ?b (msg ?x ?c)) &
    (itp ?t (S ?n)) =
    (itp ?t ?n) then (receive ?a ?b (msg ?x ?c))))
```

Thus, within indexed transition path T, this says that actor a is about to send *(msg x c)* to actor b at point n if and only if actor a has local state *lsa*, actor b has local state *lsb*, actor a is ready to send *(msg x c)* to actor b, and the next transition in the path is the actual sending of *(msg x c)* to b. The same follows for the *receive* definition as well.

With these definitions, it follows simply that an actor sends (receives) one message before another in a transition path if there are points in the path at which each message is about to be sent (received), and the first point is before the second. This is specified in terms of *sends-before* and *receives-before* predicates:

```
declare sends-before, receives-before:
 (Id,LS) [(TP Id LS) Id Id Ide Ide] -> Boolean

define sends-before-def :=
 (forall ?T ?a ?b ?c0 ?c1 ?sq0 ?sq1 .
   (sends-before ?T ?a ?b (msg ?sq0 ?c0) (msg ?sq1 ?c1))
   <==>
   exists ?n0 ?n1 ?s0 ?lsa0 ?lsb0 ?s1 ?lsa1 ?lsb1 .
     (about-to-send ?T ?n0 ?s0 ?a ?b ?lsa0 ?lsb0 ?sq0 ?c0) &
     (about-to-send ?T ?n1 ?s1 ?a ?b ?lsa1 ?lsb1 ?sq1 ?c1) &
     ?n0 < ?n1)

define receives-before-def :=
 (forall ?t ?a ?b ?c0 ?c1 ?sq0 ?sq1 .
   ((receives-before ?t ?a ?b (msg ?sq0 ?c0) (msg ?sq1 ?c1))
    <==>
    (exists ?n0 ?n1 ?s0 ?lsa0 ?lsb0 ?s1 ?lsa1 ?lsb1 .
      (about-to-receive ?t ?n0 ?s0 ?a ?b ?lsa0 ?lsb0 ?sq0 ?c0) &
      (about-to-receive ?t ?n1 ?s1 ?a ?b ?lsa1 ?lsb1 ?sq1 ?c1) &
      ?n0 < ?n1)))
```

These definitions and axioms restrict the actor model and are intended to establish FIFO communication. In the next section, we show that they do in fact impose FIFO communication on the actor model.

4. Proving FIFO

The main theorem to be shown is that messages must be received in the order in which they were sent. In other words, if a message was sent before another, then it must also be received before the other. In terms of the *sends-before* and *receives-before* predicates, and arbitrary actor ids *sender* and *receiver*, we have

```
define send-receive-order :=
(forall ?T ?c0 ?c1 ?sq0 ?sq1 .
 (unique-ids config ?T) &
 (sends-before ?T sender receiver (msg ?sq0 ?c0) (msg ?sq1 ?c1))
 ==>
 (receives-before ?T receiver sender (msg ?sq0 ?c0) (msg ?sq1 ?c1)))
```

To carry out the proof, we introduce several lemmas, discussing them in turn in the next sections.

4.1 Monotonicity of Sequence Numbers

It must be shown that when any transition occurs, the sequence numbers that actors hold never decrease. The proof takes advantage of the *actor-monotonicity* lemma, which was described in Section 2.5.

First, predicates are required that state that a local state's send (receive) sequence number is less than another local state's send (receive) sequence number. These relations are the *sslt* and *rslt* predicates, which stand for "send sequence less than" and "receive sequence less than," resp.

```
declare sslt, rslt: (Id,LS) [LS LS] -> Boolean

define sslt-def :=
 (forall ?ls0 ?ls1 .
   ?ls0 sslt ?ls1 <==>
    (send-seq (actor sender ?ls0) receiver) <=
     (send-seq (actor sender ?ls1) receiver))

define rslt-def :=
 (forall ?ls0 ?ls1 .
  ?ls0 rslt ?ls1 <==>
   (recv-seq (actor receiver ?ls0) sender) <=
    (recv-seq (actor receiver ?ls1) sender))
```

First it must be shown that the *rslt* and *sslt* predicates themselves are monotonic, i.e.,

```
define sslt-mono :=
  (forall ?ls0 ?step . ?ls0 sslt (next ?ls0 ?step))

define rslt-mono :=
  (forall ?ls0 ?step . ?ls0 rslt (next ?ls0 ?step))
```

The proofs for these two lemmas are straight-forward, and take advantage of Athena's *datatype-cases* proof mechanism to show that no matter what kind of transition the *step* variable is, it must follow that the *rslt* predicate will hold:

```
define rslt-mono-proof :=
method (theorem adapt)
 let {given := lambda (P) (get-property P adapt Theory);
      chain-last := method (L) (!chain-help given L 'last);
      [LS0 next ready-to new-ls unique-ids before after config
       sender receiver] :=
       (adapt [LS0 next ready-to new-ls unique-ids before
               after config sender receiver]);
      [ls-sort step-sort] :=
       (map sort-of (qvars-of (adapt theorem)))}
 conclude (adapt theorem)
  pick-any ls:ls-sort step:step-sort
   let {goal := (ls rslt (next ls step))}
   (!two-cases
    assume T1 :=
     (exists ?c . step = (receive receiver sender ?c))
     let {A :=
          (!chain-last
           [T1
            ==> ((recv-seq (actor receiver (next ls step))
                              sender) =
                  (S (recv-seq (actor receiver ls) sender)))
                 [(given recv-seq-rec)]])}
     (!chain-last
      [true
       ==> ((recv-seq (actor receiver ls) sender) <=
            (S (recv-seq (actor receiver ls) sender)))
           [Less=.S3]
       ==> ((recv-seq (actor receiver ls) sender) <=
            (recv-seq (actor receiver (next ls step)) sender))
           [A]
       ==> (ls rslt (next ls step))
           [(given rslt-def)]])
    assume T2 :=
     (~(exists ?c . step = (receive receiver sender ?c)))
     let {A :=
          (!chain-last
           [T2
            ==> ((recv-seq (actor receiver (next ls step))
                              sender) =
                  (recv-seq (actor receiver ls) sender))
                 [(given recv-seq-non-recv)]])}
     (!chain-last
      [(!reflex (recv-seq (actor receiver ls) sender))
       ==> ((recv-seq (actor receiver ls) sender) <=
            (recv-seq (actor receiver ls) sender))
           [Less=.Implied-by-equal]
       ==> ((recv-seq (actor receiver ls) sender) <=
            (recv-seq (actor receiver (next ls step)) sender))
           [A]
       ==> (rslt ls (next ls step))
           [(given rslt-def)]]))
```

We are then able to prove that if an actor is in a local state at one transition, and another in a second transition, and the first transition leads to the second transition, then the receive sequence number of the first local state with respect to any actor must be less than or equal to the receive sequence number of the second local state with respect to the same actor.

```
define rslt-act-mono :=
  (forall ?T ?T0 ?s0 ?ls0 ?s ?ls .
    (unique-ids config ?T0) &
    config ?T0 = ?s0 ++ (actor receiver ?ls0) &
    config ?T = ?s ++ (actor receiver ?ls) &
    ?T0 -->>* ?T
    ==> ?ls0 rslt ?ls)
```

The proof of this lemma is an application of *actor-monotonicity* theorem.

From these results we derive a lemma pertaining to equality of receive sequence numbers, that if two transition points have an actor in two local states, but the receive sequence numbers with respect to another actor are the same, then there cannot be a transition between them that involves that actor receiving a message from the respective actor.

```
define recv-seq-eq :=
  (forall ?T ?T' ?s ?s' ?ls ?ls' .
    config ?T = ?s ++ (actor receiver ?ls) &
    config ?T' = ?s' ++ (actor receiver ?ls') &
    (unique-ids config ?T) & (unique-ids config ?T') &
    (recv-seq (actor receiver ?ls) sender) =
       (recv-seq (actor receiver ?ls') sender)
    ==> ~ exists ?T0 ?T0' ?c .
           ?T -->>+ ?T0 & ?T0 -->>* ?T' &
           ?T0 = (?T0' then (receive receiver sender ?c)))
```

The corresponding lemmas for the *sslt* predicate are as follows:

```
define sslt-act-mono :=
 (forall ?t ?t0 ?s0 ?ls0 ?s ?ls .
   (unique-ids config ?t0) &
   config ?t0 = ?s0 ++ (actor sender ?ls0) &
   config ?t = ?s ++ (actor sender ?ls) &
   ?t0 -->>* ?t
   ==> ?ls0 sslt ?ls)

define send-seq-eq :=
  (forall ?t ?t' ?s ?s' ?ls ?ls' .
    config ?t = ?s ++ (actor sender ?ls) &
    config ?t' = ?s' ++ (actor sender ?ls') &
    (unique-ids config ?t) & (unique-ids config ?t') &
    (send-seq (actor sender ?ls) receiver) =
      (send-seq (actor sender ?ls') receiver)
    ==> ~ exists ?t0 ?t0' ?c .
           ?t -->>+ ?t0 & ?t0 -->>* ?t' &
           ?t0 = (?t0' then (send sender receiver ?c)))
```

4.2 Sequence Ordering

We then show a small lemma that claims that if an actor is ready to send a message, then the sequence number of that message must correspond to the send sequence number of the sender with respect to the receiver.

```
define enabled-send-seq :=
 (forall ?t ?a ?b ?c ?x ?s ?ls .
   config ?t = ?s ++ (actor ?a ?ls) &
   ?ls ready-to (send ?a ?b (msg ?x ?c))
   ==> ?x = (send-seq (actor ?a ?ls) ?b))
```

This lemma follows quickly from the *ready-to* axioms given earlier.

The final lemma claims that if an actor is about to send two messages, then the sequence numbers of the messages dictate the order of the messages.

```
define send-seq-ordered :=
 (forall ?t ?c0 ?c1 ?n0 ?n1
        ?x ?y ?s0 ?lsa0 ?lsb0 ?s1 ?lsa1 ?lsb1 .
   (about-to-send ?t ?n0 ?s0 sender receiver ?lsa0 ?lsb0 ?x ?c0) &
   (about-to-send ?t ?n1 ?s1 sender receiver ?lsa1 ?lsb1 ?y ?c1) &
   (unique-ids config ?t) ==>
   (?n0 < ?n1 <==> ?x < ?y))
```

The full proof is in Appendix A, but we give an informal version here.

To prove this lemma, first we assume that at point *n0* in transition *T*, *sender* is about to send *(msg x c0)* to *receiver*, and at point *n1* in transition *T*, *sender* is about to send *(msg y c1)* to *receiver*. We then prove the conclusion by using a standard biconditional introduction.

For the forward direction, we assume $n0 < n1$. Therefore, *(itp T n0)* leads to *(itp T n1)*. From the *sslt-act-mono* lemma, it follows that $x \leq y$. If $x = y$, then from *send-seq-eq* it is known that there can not be a transition between them in which *sender* sends a message to *receiver*. However, *(itp T n0)* is such a transition, which means that $x \neq y$, and therefore $x < y$.

For the opposite direction, we assume $x < y$. For a contradiction we assume that $n1 \leq n0$. Therefore, *(itp T n1)* leads to *(itp T n0)*, and from *send-seq-eq* it follows that the send sequence number of *sender* with respect to *receiver* at *n1* must be less than or equal to the send sequence number of *sender* with respect to *receiver* at *n0*. This is a contradiction, which means that $n0 < n1$.

With all of these lemmas defined, we are now ready to prove that the all of the axioms given above ensure message ordering. The full, formal version of this proof is in Appendix B

First, we let *m0 = (msg x0 c0)*, and *m1 = (msg x1 c1)*. We then assume that *sender* sends *m0* to *receiver* before *m1*. Therefore, *sender* will be ready to send *m0* at *(itp t n0)*, which *enabled-send-seq* gives that *x0* will be the sending sequence number of *sender* with respect to *receiver* at *(itp t n0)*, and *x1* will be the sending sequence number of *sender* with respect to *receiver* at *(itp t n1)*. Therefore, from *send-seq-ordered* it is known that $x0 < x1$. From *guaranteed-message-delivery*, it is known that *receiver* will eventually receive both messages. Let *n0'* be the point at which *receiver* receives message *m0* and *n1'* be the point at which *receiver* receives the message *m1*. Since it will be ready to receive both messages, *x0* must be the receiving sequence number of *receiver* with respect to *sender* at *(itp t n0')*, and the same for *x1* and *(itp t n1')*. From *receive-seq-ordered*, it is known that $n0' < n1'$, which means that *receiver* must receive *m0* before *m1*.

5. FIFO Application

5.1 The Sieve of Eratosthenes

Earlier, we discussed an actor model for the computation of the Sieve of Eratosthenes. We also showed that without FIFO communication, the Sieve of Eratosthenes may return an invalid answer. This makes the Sieve of Eratosthenes a good example to emphasize the importance of FIFO communication.

```
(Initial then (send S2 Output (sm two))
         then (receive Output S2 (sm two))
         then (send Input S2 (sm three))
         then (send Input S2 (sm four))
         then (send Input S2 (sm five))
         then (send Input S2 (sm six))
         then (send Input S2 (sm seven))
         then (send Input S2 (sm eight))
         then (send Input S2 (sm nine))
         then (receive S2 Input (sm nine))
         then (create S2 (si nine))
         then (send (si nine) Output (sm nine))
         then (receive Output (si nine) (sm nine)))
```

However, the full version of the Sieve of Eratosthenes uses dynamic actor creation and requires the ability to iterate through a pipeline of actors, concepts which are outside the scope of this thesis. Therefore, a smaller version of the Sieve of Eratosthenes is used instead. This version will only use two actors, the *Input* actor and the *Output* actor. The *Input* actor will begin with the number three, and send each value in turn to the *Output* actor as it does in the original version of the Sieve of Eratosthenes. The *Output* actor will begin with the number two in its list of numbers, and filter out any value which is divisible by any of its held values. This model is referred to later in this thesis as the "Small Sieve Model". Figure 5.1 shows the graphical representation of the Small Sieve Model.

The Small Sieve Model creates a smaller version of the Sieve of Eratosthenes, but still requires FIFO communication in order to ensure correctness. For example,

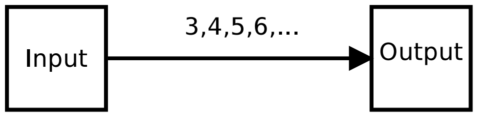

Figure 5.1: The Small Sieve Model. The Input actor sends each number to the Output actor, which directly filters the numbers

without FIFO communication it would still be possible for the *Output* actor to receive the number nine before it receives the number three.

In the following sections we will formally define the Small Sieve Model in the Athena proof system, and proceed to show that FIFO communication ensures that any transition path within the Small Sieve Model will be correct.

5.2 Defining the Small Sieve Model

We now formally define the datatypes for the Small Sieve Model. As we previously stated, there are only two actors, the *Input* actor and the *Output* actor. The *Input* actor only stores an integer in its local state, while the *Output* actor only stores a list of integers that it has received.

```
datatype Name := Input | Output

datatype ILS := (inn N)
datatype OLS := (out (List N))
datatype SLS := (input ILS) | (output OLS)
```

Since many of the functions for actor model proofs were formally defined as polymorphic, this allows us to redefine functions that work specifically with the Small Sieve model.

```
declare config: [(TP Name SLS)] -> (Cfg (Actor Name SLS))
declare next: [SLS (Step Name)] -> SLS
declare new-ls: [SLS] -> SLS
declare ready-to, ready-to-pred: [SLS (Step Name)] -> Boolean
declare sls0: SLS
declare sitp: [(TP Name SLS) N] -> (TP Name SLS)
declare unique-ids: [(Cfg (Actor Name SLS))] -> Boolean
declare output-subset: [SLS SLS] -> Boolean
declare input-le: [SLS SLS] -> Boolean

declare sm: [N] -> Ide
assert sm-injective :=
  (forall ?x ?y . (sm ?x) = (sm ?y) ==> ?x = ?y)
```

Next, readiness and transitions must be defined for the *Input* and *Output* actors. Since the Small Sieve Model will be using the FIFO communication theories, we define readiness in terms of the *ready-to-pred* predicate instead of the *ready-to* predicate.

For the *Input* actor, it is always ready to send its held integer, and after sending it, the *Input* actor increments its held value.

```
module Input {
assert ready-to-def :=
[(forall ?ls ?to ?c .
  ?ls ready-to-pred (send Input ?to ?c) <==>
  (?to = Output &
    (exists ?n . ?c = (sm ?n) & ?ls = (input inn ?n))))
 (forall ?ls ?fr ?c .
  ?ls ready-to-pred (receive Input ?fr ?c) <==> false)
 (forall ?ls ?new .
  ?ls ready-to-pred (create Input ?new) <==> false)]

assert next-def :=
(fun-def
 [(next (input inn ?i) (receive ?id ?fr ?c)) --> (input inn ?i)
  (next (input inn ?i) (send ?id ?to ?c)) -->
 [(?id = Input & ?to = Output & ?c = (msg ?sq (sm ?i))) -->
  (input inn (S ?i))
   _ --> (input inn ?i)]
 (next (input inn ?i) (create ?id ?new)) --> (input inn ?i)])
}
```

For the *Output* actor, it is always ready to receive a value, no matter what that value is. However, after receiving it, the *Output* actor will only add the integer to its list if that value is not divided by any of its currently held values.

In addition, the axioms for the *output-subset* predicate are given here. Since the *output-subset* predicate is defined across all local states, axioms for both actors must be given.

```
module Output {
assert ready-to-def :=
[(forall ?ls ?to ?c .
 ?ls ready-to-pred (send Output ?to ?c) <==> false)
 (forall ?ls ?fr ?c .
    ?ls ready-to-pred (receive Output ?fr ?c) <==>
     (?fr = Input & (exists ?n ?l . ?c = (sm ?n) &
      ?ls = (output out ?l))))
 (forall ?ls ?new .
  ?ls ready-to-pred (create Output ?new) <==> false)]

assert next-def :=
(fun-def
 [(next (output out ?ls) (receive ?id ?fr ?c)) -->
  [(?id = Output & ?fr = Input & ?c = (msg ?sq (sm ?n)) &
    (forall ?m . ?m In ?ls ==> ~(?m divides ?n))) -->
    (output out (?n :: ?ls))
   _ --> (output out ?ls)]
  (next (output out ?l) (send ?id ?to ?c)) --> (output out ?l)
  (next (output out ?l) (create ?id ?new)) --> (output out ?l)])

assert next-invalid-recv :=
(forall ?ls ?id ?fr ?c .
 ~(?id = Output & ?fr = Input &
   (exists ?sq ?n . ?c = (msg ?sq (sm ?n))))
 ==> (next (output out ?ls) (receive ?id ?fr ?c))
     = (output out ?ls))

 assert next-def-some-divide :=
  (forall ?ls ?n ?sq .
    (exists ?m . ?m In ?ls & ?m divides ?n)
    ==> (next (output out ?ls)
              (receive Output Input (msg ?sq (sm ?n))))
         = (output out ?ls))

 assert output-subset-def :=
  (forall ?ls ?ls' .
   (output out ?ls) output-subset ?ls' <==>
```

```
    (exists ?l . ?ls' = (output out ?l) &
     (forall ?x . ?x In ?ls ==> ?x In ?l)))

 assert output-subset-input :=
  (forall ?ls ?ls' .
   (input inn ?ls) output-subset ?ls' <==> true)
}
```

5.3 Correctness

We then show that our setup for FIFO communication ensures that the Small Sieve Model will be correct. We start by given the definition of correctness for the Small Sieve Model.

In terms of the Sieve of Eratosthenes, and therefore the Small Sieve Model, a transition path is correct iff every number held by the *Output* actor is a prime number.

```
(forall ?t .
 (Correct ?t) <==>
  (forall ?s ?ls .
   config ?t =
     ?s ++ (actor Output (output out ?ls))
   ==> (forall ?n .
        ?n In ?ls ==> Prime ?n)))
```

Before we begin the proofs about correctness, a few convenience functions are defined in order to improve the readability of the proofs themselves:

```
define IC := lambda (T)
 (config T =
   (actor Input (input inn three))
++ (actor Output (output out (two :: nil))))

define sieve-send := lambda (M)
 (send Input Output M)
define sieve-recv := lambda (M)
 (receive Output Input M)
define SMSG := lambda (S C)
 (msg S (sm C))
define IOSEQ := lambda (X)
```

```
 (send-seq (actor Input (input inn X)) Output)
define OSEQ := lambda (LS)
 (recv-seq (actor Output LS) Input)
define OISEQ := lambda (L)
 (OSEQ (output out L))
define IOMSG := lambda (X)
 (msg (send-seq (actor Input (input inn X)) Output) (sm X))
declare input-about-to-send, output-about-to-receive:
 [(TP Name SLS) N] -> Boolean
```

The *IC* macro simply defines the initial conditions of the Small Sieve Model. It claims that the configuration of transition *T* will only consist of the *Input* and *Output* actors, with the *Input* actor holding the number three, and the *Output* actor holding a list with just the number two.

The *sieve-send* and *sieve-recv* macros use the fact that only the *Input* actor ever sends a message, and only the *Output* actor ever receives a message to simplify the expression of *send* and *receive* transitions.

The *SMSG*, *IOSEQ*, *OSEQ*, *OISEQ*, and *IOMSG* macros wrap up the *msg*, *send-seq*, and *recv-seq* functions to further simplify the writing of the proofs about the Small Sieve Model.

The definitions of *input-about-to-send* and *output-about-to-receive* simplify the definitions of the *about-to-send* and *about-to-receive* predicates:

```
assert input-about-to-send-def :=
(forall ?t ?n ?x .
 (input-about-to-send (sitp ?t ?n) ?x) <==>
  (exists ?ls ?sq ?s .
   config (sitp ?t ?n) = ?s ++ (actor Input ?ls) &
   ?ls ready-to (sieve-send (SMSG ?sq ?x)) &
   (sitp ?t (S ?n)) =
    (sitp ?t ?n) then (sieve-send (SMSG ?sq ?x))))


assert output-about-to-receive-def :=
(forall ?t ?n ?x .
 (output-about-to-receive (sitp ?t ?n) ?x) <==>
  (exists ?ls ?s .
   (config (sitp ?t ?n) =
    ?s ++ (actor Output ?ls) ++ (MSG (IOSEQ ?x) ?x) &
    ?ls ready-to (sieve-recv (IOMSG ?x)) &
    (sitp ?t (S ?n)) =
     ((sitp ?t ?n) then (sieve-recv (IOMSG ?x))))))
```

Finally, the goal of this section is to prove that given initial conditions, the Small Sieve Model computation will always be correct:

```
define always-correct :=
(forall ?t .
 ((IC ?t) & (unique-ids config ?t))
 ==> (forall ?n . (Correct (sitp ?t ?n))))
```

To this end, a few lemmas are given. Many of these follow from the definition of the Small Sieve Model:

```
define input-sends-before :=
(forall ?t ?x ?y .
 ?x < ?y ==>
 (sends-before ?t Input Output (IOMSG ?x) (IOMSG ?y)))

define output-monotonic :=
(forall ?t ?t0 ?ls0 ?ls ?s0 ?s .
  (unique-ids config ?t0) &
  (unique-ids config ?t) &
  config ?t0 = ?s0 ++ (actor Output ?ls0) &
  config ?t = ?s ++ (actor Output ?ls) &
  ?t0 -->>* ?t
  ==> ?ls0 output-subset ?ls)

define output-receives-once :=
(forall ?t ?n ?m ?x .
  (unique-ids config (sitp ?t ?n)) &
  (unique-ids config (sitp ?t ?m)) &
  (output-about-to-receive (sitp ?t ?n) ?x) &
  (output-about-to-receive (sitp ?t ?m) ?x)
  ==> ?n = ?m)

define must-have-received :=
(forall ?t .
 ((IC ?t) & (unique-ids config ?t)) ==>
  (forall ?n ?s ?l ?x .
   (config (sitp ?t ?n) = ?s ++ (actor Output (output out ?l)) &
    ?x In ?l & ?x > two)
   ==>
   (exists ?n' .
    ?n' < ?n &
    (output-about-to-receive (sitp ?t ?n') ?x))))

define output-filtered :=
(forall ?t ?n ?x ?s ?l .
  (IC ?t) &
  (unique-ids config ?t) &
  (output-about-to-receive (sitp ?t ?n) ?x) &
  (config (sitp ?t (S ?n)) = ?s ++ (actor Output (output out ?l))) &
  (~(?x In ?l))
  ==>
  (forall ?n' ?s' ?l' .
    ?n' > ?n &
    (config (sitp ?t ?n') = ?s' ++ (actor Output (output out ?l')))
    ==>
    (~(?x In ?l'))))

define one-isnt-in-output :=
(forall ?t .
 ((IC ?t) & (unique-ids config ?t)) ==>
  (forall ?n ?l .
    (actor' Output (output out ?l)) in (config (sitp ?t ?n))
    ==> (~(one In ?l))))
```

The *input-sends-before* lemma claims that if a value $x < y$, then the *Input* actor must send x before it sends y. This is due to the fact that the *Input* actor increments its state after each time it sends a message, which means that it must send the smaller value first.

The *output-monotonic* lemma builds on the *output-subset* function and actor monotonicity from section 2.5. Since the list of numbers that the *Output* actor holds will always be a subset of its next transition, it follows that for any two transitions, if one leads to another then the list held by the *Output* actor in the first transition must be a subset of the list held by the *Output* actor in the second transition.

The *output-receives-once* lemma claims that the *Output* actor can only be about to receive any given value once. This is due to the fact that the *Input* actor only ever sends a value once, which means that the message can only ever be in the system once, and can therefore only be received once. In addition, the *Output* actor can only receive messages from the *Input* actor, due to the definition of the *Output* actor's readiness to receive.

The *must-have-received* lemma claims that for any value value greater than *two*, if the *Output* actor's list contains that value, then it must have received it at some point prior to the current point.

The *output-filtered* lemma claims that if the *Output* actor is about to receive a value, but it is not in the list held by the *Output* actor in the very next step, then the *Output* actor must have filtered it, which means that it will not be in any future configuration.

The final lemma is the basic lemma that simply claims that the number *one* will never be in the list held by the *Output* actor. This follows quickly from the other lemmas. If the value *one* is in the list held by the *Output* actor at some point, then since it is not in the *Output* actor's list of numbers in the initial configuration, *must-have-received* claims that it must have received it at an earlier point. However, the first sent message must be the value *three*, which means that *three* will be sent before any other value. However, *input-sends-before* claims that *one* must have been sent before *three*. Therefore, this is a contradiction, and *one* cannot be in any list held by the *Output* actor.

With these lemmas, we are now able to prove that the Small Sieve Model always gives prime numbers.

First we assume that at transition t, the *Output* actor holds the list ls. It then must be shown that all elements of this list are prime. This is done by contradiction, and it is therefore assumed that some element, $n \in ls$ is not prime. Therefore, there must be some prime p that divides n, which means that $p < n$. This means that the *Input* actor will send p before it sends n, and from the *send-receive-order* theorem from the *FIFO Theory*, it follows that the *Output* actor will receive p before it receives *n*.

From *one-isnt-in-output*, *one* cannot be in the *Output* actor's list of numbers. Since *p* is prime, the *Output* actor must add *p* to its list of integers at this point. Therefore, when the *Output* actor receives *n*, it will already be holding *p*, which means that *n* will be filtered out. From *output-receives-once* we know that the *Output* actor can only receive any value once, which means that it will never receive *n* again which means that *n* can not be in *ls*. This contradicts the original statement, which means that *n* must be prime.

6. Related Work

6.1 Formal Proof Systems

A lot of work by other authors has been done on proof systems that create proofs in a machine-checkable way. However, Athena aims to do this in a way that allows the proofs to be easily read.

A few such systems are Isabelle [11], Coq [12], and HOL [13]. Unfortunately, even though these systems can all create machine-checkable proofs, these systems become difficult to follow without prior knowledge.This means that it is not as easy to learn from any of these systems as it is from the proofs made in Athena. These systems tend to work in the background, preventing the user from seeing the steps that were taken to prove a specific theorem without running the theorem through the program itself.

The only possible exception is Isabelle-ISAR [14] which provides a far more readable language than any of the above languages. However, even ISAR is unable to replicate the ability to define recursive proof methods within a proof, as is possible in Athena. Since this technique is used in some of the actor model theories, it is not apparent how ISAR would be able to replicate the entire actor model reasoning framework developed in Athena.

6.2 Actor Model Proofs

Athena is not the only proof system to explore actor model proofs. The Rebeca Modeling Language [15] is dedicated solely to this task. In Rebeca, actor model computations are modeled by means of different objects that communicate by means of message passing. These messages are stored in each object's message queue, and processed in the order in which they arrive. However, Rebeca uses model checking which does not allow to reason about unbounded nondeterminism as Athena does, making it difficult to reason about some computations that require this kind of nondeterminism.

6.3 FIFO

While the actor model framework that is created within Athena does not initially assume FIFO communication, there are other systems that do. The ABCL/1 is an object-oriented concurrent computation model that does assume FIFO communications, or a *transmission ordering law* [10]. Specifically, in ABCL/1 each actor, or object, sends messages in an order dictated by its local clock. The receiver of these messages must receive them in the same order as dictated by its local clock. This means of communication thus ensures that communications are done in a FIFO manner. However, this requirement is built into ABCL/1. If an actor system does not guarantee FIFO communication, then ABCL/1 may find that certain properties hold, while in reality they do not. For example, if an implementation of the actor representation of the Sieve of Eratosthenes did not guarantee FIFO communication, the ABCL/1 system would claim that the system would always be correct, while in reality it would not be.

7. Discussion

7.1 Conclusion

In this thesis, we were able to extend the actor model reasoning framework in Athena to include the ability to prove correctness about actor systems that require FIFO communication. We accomplished this by extending messages to include sequence numbers, and set up a means by which each actor in the system could store its send and receive sequence numbers. We then only allowed each actor to receive a message if the sequence number of the message corresponded to the receiving sequence number of the receiving actor. Building upon previous results, we showed that this forces messages to be received in the same order in which they were sent. This framework then allowed us to prove that a smaller version of the Sieve of Eratosthenes would always return prime numbers, and therefore would always be correct.

The main contribution of this work is improving upon the ability to reason about concurrent computations. It is always important to know that a certain computation gives the answer that one expects, and it is extremely useful to be able to prove this fact. It is even more difficult to reason through a concurrent computation due to the increased complications brought on by concurrency. It is therefore very helpful to have the ability to confirm that a concurrent computation, and by extension an actor model computation, is in fact working as expected. It is this ability that has been improved upon in this thesis.

In addition, being developed in Athena means that the proofs are human-readable, which means that it is possible to learn from the proofs. Since many of the proofs regarding FIFO communications were developed at the abstract level, it is possible for them to be reused for later proofs about actor systems that require FIFO communication to be correct.

7.2 Future Work

Since the proofs of correctness only pertained to the Small Sieve Model, proofs relating to the full Sieve of Eratosthenes is the direction in which this work could continue. This would require creating a means of reasoning about dynamic actor creation. Specifically, the Sieve of Eratosthenes requires that ability to recurse over a pipeline of actors. A pipeline of actors in this case is a set of actors such that each one only sends to one actor, and only receives from one actor. In the case of the Sieve of Eratosthenes, the pipeline would start at the *Input* actor, and continue along the chain created by the Sieve actors. Proving the correctness of the complete Sieve of Eratosthenes model would require proving that each of these actors holds a prime number as its value, and thus would require the ability to recurse along this pipeline.

The Sieve of Eratosthenes also requires reasoning about dynamic actor creation. While the current implementations of the actor model in Athena use a finite number of actors, the full Sieve of Eratosthenes can create an arbitrarily large amount of actors that create the pipeline. This introduces complexities that are not present when only a finite number of actors are used. For example, if one of the Sieve actors needs to send a message to another Sieve actor, many of the existing theories require knowledge of both actors' local state. While this should be guaranteed, it would be one additional lemma to prove when reasoning about the Sieve of Eratosthenes.

LITERATURE CITED

[1] C. Hewitt, "Viewing control structures as patterns of passing messages," *Artificial Intell.*, vol. 8, no. 3, pp. 323 –364, Jun. 1977.

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

[3] G. A. Agha, *et al.*, "A foundation for actor computation," *J. Functional Programming*, vol. 7, no. 1, pp. 1–72, Jan. 1997.

[4] C. L. Talcott, "Composable semantic models for actor theories," *Higher-Order and Symbolic Computing*, vol. 11, no. 3, pp. 281–343, Oct. 1998.

[5] R. Virding, *et al.*, *Concurrent Programming in ERLANG*, J. Armstrong, Ed., 2nd ed. Hertfordshire, UK: Prentice Hall Int. Ltd., 1996.

[6] C. Varela and G. Agha, "Programming dynamically reconfigurable open systems with salsa," *SIGPLAN Notices*, vol. 36, no. 12, pp. 20–34, Dec. 2001.

[7] T. Desell, *et al.*, "Malleable applications for scalable high performance computing," *Cluster Computing*, vol. 10, no. 3, pp. 323–337, Sep. 2007.

[8] D. R. Musser and C. A. Varela, "Structured reasoning about actor systems," in *Proc. 2013 Workshop Programming Based Actors, Agents, and Decentralized Control*, Indianapolis, Indiana: ACM, 2013, pp. 37–48.

[9] A. Yonezawa, *et al.*, "Object-oriented concurrent programming abcl/1," in *Conf. Proc. Object-Oriented Programming Systems, Languages and Applications*, Portland, Oregon: ACM, 1986, pp. 258–268.

[10] A. Yonezawa, Ed., *ABCL: An Object-Oriented Concurrent System.* Cambridge, MA: MIT Press, 1990.

[11] L. C. Paulson, *Isabelle: A Generic Theorem Prover*, ser. LNCS 828. Berlin, Germany: Springer, 1994.

[12] Coq Development team. (2004). The coq proof assistant reference manual. Version 8.0, LogiCal Project, [Online]. Available: http://coq.inria.fr/distrib/current/files/Reference-Manual.pdf.

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

[14] M. Wenzel, “Isabelle/Isar — A versatile environment for human-readable formal proof documents,” Ph.D. dissertation, Institut für Informatik, Technische Universität München, 2002.

[15] M. Sirjani and M. M. Jaghoori, “Ten years of analyzing actors: rebeca experience,” in *Formal Modeling: Actors, Open Systems, Biological Systems*, G. Agha, *et al.*, Eds. Berlin, Germany: Springer, 2011, pp. 20–56.

APPENDIX A
Proof of Sequence Ordering

```
define send-seq-ordered-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);
      [LS0 itp config before after next ready-to new-ls
       unique-ids ready-to-pred sender receiver] :=
        (adapt [LS0 itp config before after next ready-to
                new-ls unique-ids ready-to-pred sender receiver]);
      [t-sort _ _ _ _ _ _ _ s-sort ls-sort _ _ _ _] :=
        (map sort-of (qvars-of (adapt theorem)));
      ++A := (given ['++ Associative]);
      ++C := (given ['++ Commutative]);
      SSE := (!lemma send-seq.send-seq-eq);
      ESS := (!lemma enabled-send-seq);
      UIP := (!lemma unique-ids-persistence);
      ITC := (!lemma itp-connected);
      SAM := (!lemma send-seq.sslt-act-mono);
      RR+I := (!lemma ['-->> RR+-inclusion])}
 conclude (adapt theorem)
  pick-any t:t-sort c0:Ide c1:Ide n0:N n1:N
           x:N y:N s0:s-sort lsa0:ls-sort lsb0:ls-sort s1:s-sort
           lsa1:ls-sort lsb1:ls-sort
   let {a := sender; b := receiver}
   assume A := ((about-to-send t n0 s0 a b lsa0 lsb0 x c0) &
                (about-to-send t n1 s1 a b lsa1 lsb1 y c1) &
                (unique-ids config t))
    let {A1 := (!chain-last
                [(about-to-send t n0 s0 a b lsa0 lsb0 x c0)
                 ==> (config (itp t n0) =
                        s0 ++ (actor a lsa0)
                           ++ (actor b lsb0) &
                      lsa0 ready-to (send a b (msg x c0)) &
                      (itp t (S n0)) =
                        (itp t n0) then (send a b (msg x c0)))
                     [(given about-to-send-def)]]);
          A2 := (!chain-last
                  [(about-to-send t n1 s1 a b lsa1 lsb1 y c1)
                   ==> (config (itp t n1) =
                         s1 ++ (actor a lsa1)
                            ++ (actor b lsb1) &
                       lsa1 ready-to (send a b (msg y c1)) &
                       (itp t (S n1)) =
                       (itp t n1) then (send a b (msg y c1)))
                       [(given about-to-send-def)]]);
          A0 := (!chain-last
                  [(unique-ids config t)
                   ==> (unique-ids config (itp t zero))
                       [(given itp-initial)]]);
          A0-1 := (config (itp t n0) =
                    s0 ++ (actor a lsa0) ++ (actor b lsb0));
          A0-2 := (lsa0 ready-to (send a b (msg x c0)));
          A0-3 := ((itp t (S n0)) =
                    (itp t n0) then (send a b (msg x c0)));
          A1-1 := (config (itp t n1) =
                    s1 ++ (actor a lsa1) ++ (actor b lsb1));
          A1-2 := (lsa1 ready-to (send a b (msg y c1)));
          A1-3 := ((itp t (S n1)) =
                    (itp t n1) then (send a b (msg y c1)));
          B0-01 := (!chain-last
                     [A0-1
                      ==> (config (itp t n0) =
                            (s0 ++ (actor b lsb0)) ++ (actor a lsa0))
                          [++C ++A]]);
          B0-02 := (!chain-last
                     [(B0-01 & A0-2)
                       ==> (x = (send-seq (actor a lsa0) b))
                           [ESS]]);
          B0-11 := (!chain-last
                     [A1-1
                      ==> (config (itp t n1) =
                            (s1 ++ (actor b lsb1)) ++ (actor a lsa1))
                          [++C ++A]]);
          B0-12 := (!chain-last
                     [(B0-11 & A1-2)
                      ==> (y = (send-seq (actor a lsa1) b))
                          [ESS]]);
          B0-03a := (conclude (zero <= n0) (!uspec Less=.zero<= n0));
          B0-03 := (!chain-last
                     [(A0 & B0-03a)
                      ==> (unique-ids config (itp t n0))
                          [UIP]]);
          B0-13a := (conclude (zero <= n1) (!uspec Less=.zero<= n1));
          B0-13 := (!chain-last
                     [(A0 & B0-13a)
                      ==> (unique-ids config (itp t n1))
                          [UIP]])}
          (!equiv
           assume B1 := (n0 < n1)
            let {B1-21 := (!chain-last
                            [B1
                             ==> (n0 <= n1)
                                 [Less=.Implied-by-<]
                             ==> ((itp t n0) -->>* (itp t n1))
                                 [ITC]]);
                 B1-E1 := (!chain-last
                            [(B0-03 & B0-01 & B0-11 & B1-21)
                              ==> (lsa0 sslt lsa1)
                                  [SAM]
                              ==> ((send-seq (actor a lsa0) b) <=
                                   (send-seq (actor a lsa1) b))
                                  [(given send-seq.sslt-def)]
                              ==> (x <= y)
                                  [B0-02 B0-12]
                              ==> ( x < y | x = y)
                                  [Less=.definition]]);
                 B1-E2 := (x =/= y);
                 _ :=
                  (!by-contradiction B1-E2
                    assume B1-E2a := (x = y)
                     let {G1 := (!chain-last
                                  [B1-E2a
                                   ==> ((send-seq (actor a lsa0) b) =
                                        (send-seq (actor a lsa1) b))
                                       [B0-02 B0-12]]);
                         G2 := (!chain-last
                                  [(B0-01 & B0-11 & B0-03 & B0-13 & G1)
                                   ==> (~(exists ?t0 ?t0' ?c .
                                            (itp t n0) -->>+ ?t0 &
                                             ?t0 -->>* (itp t n1) &
                                             ?t0 =
                                             (?t0' then (send a b ?c))))
                                       [SSE]]);
                         G3 := (!chain-last
                                  [true ==> ((itp t n0) -->>
                                               (itp t (S n0)))
                                               [itp-directly-connected]
                                        ==> ((itp t n0) -->>+
                                               (itp t (S n0)))
                                              [RR+I]]);
                         G4 :=
                          (!chain-last
                           [B1
                            ==> ((S n0) <= n1)
                                [Less=.discrete]
                            ==> ((itp t (S n0)) -->>* (itp t n1))
                                [ITC]]);
                         _ := (!both G3 (!both G4 A0-3));
                         G5 := (!egen* (exists ?t0 ?t0' ?c .
                                         (itp t n0) -->>+ ?t0 &
                                         ?t0 -->>* (itp t n1) &
                                         ?t0 =
                                          (?t0' then (send a b ?c)))
                                       [(itp t (S n0)) (itp t n0)
                                        (msg x c0)])}
                     (!absurd G5 G2))}
            (!dsyl B1-E1 B1-E2)
            assume B2 := (x < y)
             let {B2-21 := (!chain-last
                              [B2 ==> ((send-seq (actor a lsa0) b) <
                                       (send-seq (actor a lsa1) b))
                                      [B0-02 B0-12]
                                 ==> (~((send-seq (actor a lsa1) b) <=
                                         (send-seq (actor a lsa0) b)))
                                      [Less=.trichotomy3]]);
                     _ :=
                     (!by-contradiction (~ (n1 <= n0))
                      assume (n1 <= n0)
                       let {G1 := (!chain-last
                                  [(n1 <= n0)
                                   ==> ((itp t n1) -->>*
                                        (itp t n0))
                                       [ITC]
                                   ==> (B0-13 &
                                        B0-11 &
                                        B0-01 &
                                        ((itp t n1) -->>*
                                         (itp t n0)))
                                       [augment]
                                   ==> (lsa1 sslt lsa0)
                                       [SAM]
                                   ==> ((send-seq (actor a lsa1) b) <=
                                        (send-seq (actor a lsa0) b))
                                       [(given send-seq.sslt-def)]])}
                       (!absurd B2-21 (!bdn G1)))}
                (!chain-last
                 [B2-22
                  ==> (n0 < n1)
                      [Less=.trichotomy1]]))
```

APPENDIX B
Proof of FIFO Order

```
define send-receive-order-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);
      [LS0 itp config before after next ready-to new-ls
       unique-ids ready-to-pred sender receiver contents] :=
        (adapt [LS0 itp config before after next ready-to
               new-ls unique-ids ready-to-pred sender
               receiver contents]);
      [t-sort _ _ _ _] :=
        (map sort-of (qvars-of (adapt theorem)));
      SSO := (!lemma send-seq-ordered);
      RSO := (!lemma recv-seq-ordered);
      UIP := (!lemma unique-ids-persistence);
      ATR := (given about-to-receive-def);
      ATS := (given about-to-send-def);
      ++A := (given ['++ Associative]);
      ++C := (given ['++ Commutative])}
 conclude (adapt theorem)
  pick-any t:t-sort c0:Ide c1:Ide x0:N x1:N
   let {a := sender; b := receiver}
   assume T := (unique-ids config t)
    let {S0 := (!chain-last
               [T ==> (unique-ids config (itp t zero))
                      [itp-initial]])}
     assume A := (sends-before t a b (msg x0 c0) (msg x1 c1))
      let {A' := (!chain-last
                 [A
                  ==> (exists ?n0 ?n1 ?s0 ?lsa0 ?lsb0 ?s1
                             ?lsa1 ?lsb1 .
                        (about-to-send t ?n0 ?s0 a b
                                        ?lsa0 ?lsb0 x0 c0) &
                        (about-to-send t ?n1 ?s1 a b
                                       ?lsa1 ?lsb1 x1 c1) &
                        ?n0 < ?n1)
                      [sends-before-def]])}
    pick-witnesses n0 n1 s0 lsa0 lsb0 s1 lsa1 lsb1 for A' A''
     let {A0 := (about-to-send t n0 s0 a b lsa0 lsb0 x0 c0);
          A1 := (about-to-send t n1 s1 a b lsa1 lsb1 x1 c1);
          A2 := (n0 < n1);
          A3 := (!chain-last
                  [(A0 & A1 & T)
                    ==> ((n0 < n1) <==> (x0 < x1))
                        [SSO]]);
          A4 := (!mp (!left-iff A3) A2);
          A00 := (!chain-last
                   [A0
                    ==> (config (itp t n0)
                         = (s0 ++ (actor a lsa0)
                               ++ (actor b lsb0)) &
                         (lsa0 ready-to (send a b (msg x0 c0))) &
                         (itp t (S n0)) =
                          (itp t n0) then (send a b (msg x0 c0)))
                        [ATS]]);
          A01 := (config (itp t n0) =
                   s0 ++ (actor a lsa0) ++ (actor b lsb0));
          A02 := (lsa0 ready-to (send a b (msg x0 c0)));
          A03 := ((itp t (S n0)) =
                   (itp t n0) then (send a b (msg x0 c0)));
          A04a := (conclude (zero <= n0)
                         (!uspec Less=.zero<= n0));
          A04 := (!chain-last
                   [(S0 & A04a)
                    ==> (unique-ids config (itp t n0))
                        [UIP]]);
          A10 := (!chain-last
                   [A1
                    ==> (config (itp t n1)
                         = s1 ++ (actor a lsa1)
                              ++ (actor b lsb1) &
                         lsa1 ready-to (send a b (msg x1 c1)) &
                        (itp t (S n1)) =
                         (itp t n1) then (send a b (msg x1 c1)))
                       [ATS]]);
            A11 := (config (itp t n1) =
                    s1 ++ (actor a lsa1) ++ (actor b lsb1));
            A12 := (lsa1 ready-to (send a b (msg x1 c1)));
            A13 := ((itp t (S n1)) =
                    (itp t n1) then (send a b (msg x1 c1)));
            A14a := (conclude (zero <= n1)
                              (!uspec Less=.zero<= n1));
            A14 := (!chain-last
                    [(S0 & A14a)
                     ==> (unique-ids config (itp t n1))
                       [UIP]]);
            GMD := (!lemma guaranteed-message-delivery);
            B0 :=
             (!chain-last
              [(A01 & A02 & A04)
               ==>
               (exists ?n ?s ?ls2 ?ls3 .
                ?n >= n0 &
                config (itp t ?n) =
                (?s ++ (actor a ?ls2))
                    ++ (actor b ?ls3)
                    ++ (One (message' a LS0 b (msg x0 c0))) &
                    ?ls3 ready-to (receive b a (msg x0 c0)) &
                    (itp t (S ?n)) =
                     (itp t ?n) then (receive b a (msg x0 c0)))
               [GMD]]);
             B1 :=
              (!chain-last
               [(A11 & A12 & A14)
                ==>
                (exists ?n ?s ?ls2 ?ls3 .
                 ?n >= n1 &
                 config (itp t ?n) =
                  (?s ++ (actor a ?ls2))
                      ++ (actor b ?ls3)
                      ++ (One (message' a LS0 b (msg x1 c1))) &
                   ?ls3 ready-to (receive b a (msg x1 c1)) &
                  (itp t (S ?n)) =
                   (itp t ?n) then (receive b a (msg x1 c1)))
                [GMD]])}
        pick-witnesses n0' s0' lsa0' lsb0' for B0 C0
        pick-witnesses n1' s1' lsa1' lsb1' for B1 C1
         let {C0-0 := (n0' >= n0);
               C0-1 :=
                (config (itp t n0') =
                 (s0' ++ (actor a lsa0')) ++ (actor b lsb0')
                      ++ (One (message' a LS0 b (msg x0 c0))));
               C0-1a :=
                (!chain-last
                 [C0-1
                  ==> (config (itp t n0') =
                        (s0' ++ (actor b lsb0')
                             ++ (actor a lsa0')
                             ++ (One (message' a LS0
                                               b (msg x0 c0)))))
                      [++A ++A ++C ++A]]);
               C0-2 := (lsb0' ready-to (receive b a (msg x0 c0)));
               C0-3 := ((itp t (S n0')) =
                         (itp t n0') then (receive b a (msg x0 c0)));
               C0-4 := (!chain-last
                        [(C0-1a & C0-2 & C0-3)
                        ==> (about-to-receive t n0' s0' b
                                                  a lsb0' lsa0' x0 c0)
                             [about-to-receive-def]]);
               C1-0 := (n1' >= n1);
               C1-1 :=
                (config (itp t n1') =
                 (s1' ++ (actor a lsa1')) ++ (actor b lsb1')
                      ++ (One (message' a LS0 b (msg x1 c1))));
               C1-1a :=
                (!chain-last
                 [C1-1
                  ==> (config (itp t n1') =
                        (s1' ++ (actor b lsb1')
                             ++ (actor a lsa1')
```

```
                             ++ (One (message' a LS0
                                                b (msg x1 c1)))))
                       [++A ++A ++C ++A]]);
                C1-2 := (lsb1' ready-to (receive b a (msg x1 c1)));
                C1-3 :=
                 ((itp t (S n1')) =
                  (itp t n1') then (receive b a (msg x1 c1)));
                C1-4 := (!chain-last
                         [(C1-1a & C1-2 & C1-3)
                          ==> (about-to-receive t n1' s1' b a
                                                 lsb1' lsa1' x1 c1)
                              [about-to-receive-def]]);
                C2 := (!chain-last
                       [(C0-4 & C1-4 & T)
                        ==> ((n0' < n1') <==> (x0 < x1))
                            [RS0]
                        ==> ((x0 < x1) ==> (n0' < n1'))
                            [right-iff]]);
                C3 := (!mp C2 A4);
                C4 := (!both C0-4 (!both C1-4 C3));
                C5 :=
                 (!egen* (exists ?n0 ?n1 ?s0 ?lsa0 ?lsb0
                                 ?s1 ?lsa1 ?lsb1 .
                          (about-to-receive t ?n0 ?s0 b a
                                            ?lsa0 ?lsb0 x0 c0) &
                          (about-to-receive t ?n1 ?s1 b a
                                            ?lsa1 ?lsb1 x1 c1) &
                          ?n0 < ?n1)
                        [n0' n1' s0' lsb0' lsa0' s1' lsb1' lsa1'])}
          (!chain-last
           [C5
            ==> (receives-before t b a (msg x0 c0)
                                       (msg x1 c1))
                 [receives-before-def]])
```

APPENDIX C
Proof of Correctness

```
conclude always-correct
 let {given := lambda (P) (get-property P no-renaming Theory);
      chain-last := method (L) (!chain-help given L 'last);
      ++A := (given ['Sieve ['++ Associative]]);
      ++C := (given ['Sieve ['++ Commutative]]);
      lemma := method (P) (!property P no-renaming Theory);
      TR := (!lemma ['Sieve trans-receive]);
      ITC := (!lemma ['Sieve itp-connected]);
      SRO := (!lemma ['Sieve send-receive-order]);
      RTR := (!lemma ['Sieve ready-to-recv-def]);
      ISO2 := (!lemma ['Sieve Isolate2])}
 pick-any t:(TP Name SLS)
  assume I := ((IC t) & (unique-ids config t))
   let {goal := (forall ?n . (Correct (sitp t ?n)))}
    pick-any m:N
     let {goal' := (forall ?s ?ls .
                    (config (sitp t m) =
                     ?s ++ (actor Output (output out ?ls)))
                    ==>
                    (forall ?n .
                     ?n In ?ls ==> (Prime ?n)));
          Prem := (conclude goal'
     pick-any s:(Cfg (Actor Name SLS)) ls:(List N)
      assume A0 := (config (sitp t m) =
                    s ++ (actor Output (output out ls)))
       pick-any n:N
        assume A1 := (n In ls)
         let {ngoal := (~(Prime n))}
         (!by-contradiction (Prime n)
          assume ngoal
           let {A000 := (!chain-last
                         [(unique-ids config t)
                          ==> (forall ?n .
                               (unique-ids config (sitp t ?n)))
```

```
                              [always-unique-ids]]);
               A1-O := (!chain-last
                        [ngoal
                         ==> (exists ?y .
                               (Prime ?y) &
                               ?y divides n &
                               ?y =/= n)
                              [Prime.non-prime]])}
          pick-witness p for A1-O A-w
           let {sq0 := (IOSEQ p);
                sq1 := (IOSEQ n);
                A-w0 := (Prime p);
                A-w1 := (p divides n);
                A-w2 := (p =/= n);
                A-two := (!chain-last
                          [(A-w0 & A-w1)
                           ==> (exists ?y .
                                 ((Prime ?y) & (?y divides n)))
                                [existence]
                           ==> (n > two)
                                [composites->-two]]);
                A-w3 := (!chain-last
                         [A-w1
                          ==> (p <= n)
                               [divides.divs-less-than]
                          ==> (p < n | p = n)
                               [Less=.definition]]);
                A-w4 := (conclude (p < n)
                         (!dsyl A-w3 A-w2));
                A-w5 := (!chain-last
                         [A-w0
                          ==> (p =/= zero &
                                p =/= one &
                                (forall ?y .
                                 (?y divides p)
                                  <==> (?y = one | ?y = p)))
                               [Prime.definition]
                          ==> (forall ?y .
                                (?y divides p)
```

```
                    <==> (?y = one | ?y = p))
                   [right-and right-and]]);
     A-w6 := (!chain-last
              [A-w4
               ==> (sends-before t Input Output
                                (IOMSG p) (IOMSG n))
                   [input-sends-before]]);
     B :=
     (!chain-last
      [A-w6
       ==> (receives-before t Output Input
                            (IOMSG p) (IOMSG n))
           [SRO]
       ==> (exists ?n0 ?n1 ?s0 ?lsa0 ?lsb0 ?s1
                   ?lsa1 ?lsb1 .
            (about-to-receive t ?n0 ?s0 Output Input
                              ?lsa0 ?lsb0 sq0 (sm p)) &
            (about-to-receive t ?n1 ?s1 Output Input
                              ?lsa1 ?lsb1 sq1 (sm n)) &
            ?n0 < ?n1)
           [(given ['Sieve receives-before-def])]])}
pick-witnesses n0 n1 s0 lso0 lsi0 s1 lso1 lsi1 for B B-w
 let {B-w1 := (about-to-receive t n0 s0 Output Input
                                lso0 lsi0 sq0 (sm p));
      B-w2 := (about-to-receive t n1 s1 Output Input
                                lso1 lsi1 sq1 (sm n));
      B-w3 := (n0 < n1);
      B-w0a := (!uspec A000 n0);
      B-w0b := (!uspec A000 n1);
      B-w0c := (!uspec A000 (S n0));
      B-w4 :=
       (!chain-last
        [B-w1
         ==> (config (sitp t n0) =
              s0 ++ (actor Output lso0)
                 ++ (actor Input lsi0)
                 ++ (MSG sq0 p) &
              lso0 ready-to (sieve-recv (SMSG sq0 p)) &
              (sitp t (S n0)) =
              (sitp t n0) then
               (sieve-recv (SMSG sq0 p)))
             [(given ['Sieve about-to-receive-def])]]);
      B-w20 :=
       (!chain-last
        [B-w2
         ==> (config (sitp t n1) =
              s1 ++ (actor Output lso1)
                 ++ (actor Input lsi1)
                 ++ (MSG sq1 n) &
              lso1 ready-to (sieve-recv (SMSG sq n)) &
              (sitp t (S n1)) =
              (sitp t n1) then
               (sieve-recv (SMSG sq1 n)))
             [(given ['Sieve about-to-receive-def])]]);
      B-w5 := (config (sitp t n0) =
               s0 ++ (actor Output lso0)
                  ++ (actor Input lsi0)
                  ++ (MSG sq0 p));
      B-w5a := (!chain-last
                 [B-w5
                 ==> (config (sitp t n0)
                      = (s0 ++ (actor Input lsi0))
                            ++ (actor Output lso0)
                            ++ (MSG sq0 p))
                     [++A ++C ++A]]);
      B-w5b := (!chain-last
                 [B-w5a
                 ==> (config (sitp t n0)
                      = (s0 ++ (actor Input lsi0)
                            ++ (MSG sq0 p))
                            ++ (actor Output lso0))
                     [++A ++C ++A]]);
      B-w21 := (config (sitp t n1) =
                 s1 ++ (actor Output lso1)
                   ++ (actor Input lsi1)
                   ++ (MSG sq1 n));
      B-w22 := (!chain-last
                 [B-w21
```

```
                    ==> (config (sitp t n1)
                         = (s1 ++ (actor Input lsi1))
                               ++ (actor Output lso1)
                               ++ (MSG sq1 n))
                        [++A ++C ++A]]);
         B-w23 := (!chain-last
                   [B-w22
                    ==> (config (sitp t n1)
                         = (s1 ++ (actor Input lsi1)
                               ++ (MSG sq1 n))
                               ++ (actor Output lso1))
                         [++C ++A]]);
         B-w6 :=
          (lso0 ready-to (sieve-recv (SMSG sq0 p)));
         B-w24 :=
          (lso1 ready-to (sieve-recv (SMSG sq1 n)));
         B-w7 :=
          ((sitp t (S n0)) =
           (sitp t n0) then (sieve-recv (SMSG sq0 p)));
         B-wa :=
          (!chain-last
           [(B-w5a & B-w6)
            ==> (config ((sitp t n0) then
                         (sieve-recv (SMSG sq0 p))) =
                 ((s0 ++ (actor Input lsi0))
                      ++ (actor Output
                          (next lso0
                           (sieve-recv (SMSG sq0 p))))))
                 [TR]
            ==> (config (sitp t (S n0)) =
                 ((s0 ++ (actor Input lsi0))
                      ++ (actor Output
                          (next lso0
                           (sieve-recv (SMSG sq0 p))))))
                [B-w7]]);
          B-w25 :=
           ((sitp t (S n1)) =
            (sitp t n1) then (sieve-recv (SMSG sq1 n)));
          B-w26 :=
           (!chain-last
            [(B-w22 & B-w24)
             ==> (config ((sitp t n1) then
                          (sieve-recv (SMSG sq1 n))) =
                  ((s1 ++ (actor Input lsi1))
                       ++ (actor Output
                           (next lso1
                            (sieve-recv (SMSG sq1 n))))))
                 [TR]
             ==> (config (sitp t (S n1)) =
                  ((s1 ++ (actor Input lsi1))
                       ++ (actor Output
                           (next lso1
                            (sieve-recv (SMSG sq1 n))))))
                 [B-w25]]);
           B-w8 :=
            (!chain-last
             [B-w6
              ==> (exists ?ct .
                   (SMSG sq0 p) =
                   (msg (OISEQ lso0) ?ct)
                 & (lso0 ready-to-pred (sieve-recv ?ct)))
                 [RTR]]);
           B-w27 :=
            (!chain-last
             [B-w24
              ==> (exists ?ct .
                   (SMSG sq1 n) =
                    (msg (OISEQ lso1) ?ct)
                 & (lso1 ready-to-pred (sieve-recv ?ct)))
                 [RTR]])}
    pick-witness ct0 for B-w8 C-w
    pick-witness ct1 for B-w27 C2-w
     let {C-wl := ((SMSG sq0 p) =
                   (msg (OISEQ lso0) ct0));
          C-wr := (lso0 ready-to-pred (sieve-recv ct0));
          C-w0 := (!chain-last
                   [C-wl
                    ==> (sq0 = (OISEQ lso0) &
                          (sm p) = ct0)
                         [msg-injective]]);
          C-w0l := (sq0 = (OISEQ lso0));
          C-w0r := ((sm p) = ct0);
          C-w1 :=
           (!chain-last
            [C-wr
             ==> (lso0 ready-to-pred (sieve-recv (sm p)))
                 [C-w0r]
             ==> (Input = Input &
                  (exists ?n ?l .
                   (sm p) = (sm ?n) &
                   lso0 = (output out ?l)))
                 [Output.ready-to-def]
             ==> (exists ?n ?l .
                  (sm p) = (sm ?n) &
                  lso0 = (output out ?l))
                 [right-and]]);
          C2-wl := ((SMSG sq1 n) =
                    (msg (OISEQ lso1) ct1));
          C2-wr := (lso1 ready-to-pred (sieve-recv ct1));
          C2-w0 := (!chain-last
                    [C2-wl
                     ==> (sq1 = (OISEQ lso1) &
                          (sm n) = ct1)
                         [msg-injective]]);
           C2-w0l := (sq1 = (OISEQ lso1));
           C2-w0r := ((sm n) = ct1);
           C2-w1 :=
            (!chain-last
             [C2-wr
              ==> (lso1 ready-to-pred (sieve-recv (sm n)))
                  [C2-w0r]
              ==> (Input = Input &
                   (exists ?n ?l .
                    (sm n) = (sm ?n) &
                    lso1 = (output out ?l)))
                  [Output.ready-to-def]
              ==> (exists ?n ?l .
                   (sm n) = (sm ?n) &
                   lso1 = (output out ?l))
                  [right-and]])}
     pick-witnesses n0' l0 for C-w1 D-w
     pick-witnesses n1' l1 for C2-w1 D2-w
      let {D-w1 := (lso0 = (output out l0));
           D2-w1 := (lso1 = (output out l1));
           D-w2 :=
            (!chain-last
             [B-wa
              ==> (config (sitp t (S n0)) =
                   ((s0 ++ (actor Input lsi0))
                        ++ (actor Output
                             (next (output out l0)
                              (sieve-recv (SMSG sq0 p))))))
                  [D-w1]]);
           D-w3 :=
            (!chain-last
             [B-w5b
              ==> (config (sitp t n0) =
                   (s0 ++ (actor Input lsi0)
                       ++ (MSG sq0 p))
                       ++ (actor Output (output out l0)))
                  [D-w1]
              ==> (exists ?s .
                   (config (sitp t n0) =
                    ?s ++ (actor Output (output out l0))))
                  [existence]
              ==> ((actor' Output (output out l0)) in
                   (config (sitp t n0)))
                  [ISO2]]);
           D-w4 :=
            (!chain-last
             [(IC t)
              ==> (forall ?n ?l .
                   (((actor' Output (output out ?l)) in
                    (config (sitp t ?n)))
                    ==> (~(one In ?l))))
                  [one-isnt-in-output]]);
           D-w5 := (!chain-last
                    [D-w3
                     ==> (~(one In l0))
                         [D-w4]]);
           D-w6 := (p In l1);
           _ :=
            (conclude D-w6
             (!two-cases
              assume Cs1 := (p In l0)
               let {Cs1-0 :=
                    (!chain-last
                     [B-w5b
                      ==> (config (sitp t n0) =
                           (s0 ++ (actor Input lsi0)
                               ++ (MSG sq0 p))
                               ++ (actor Output
                                   (output out l0)))
                          [D-w1]]);
                    Cs1-1 :=
                     (!chain-last
                      [B-w3
                       ==> (n0 <= n1)
                           [Less=.Implied-by-<]
                       ==> ((sitp t n0) -->>* (sitp t n1))
                           [ITC]]);
                    Cs1-2 :=
                     (!chain-last
                      [(B-w0a & B-w0b & Cs1-0 &
                        B-w23 & Cs1-1)
                       ==> (output out l0 output-subset
                             lso1)
                           [output-monotonic]
                       ==> ((output out l0) output-subset
                             (output out l1))
                           [D2-w1]
                       ==> (forall ?x .
                             ?x In l0 ==> ?x In l1)
                           [output-subset-ext]])}
               (!chain-last
                [Cs1 ==> (p In l1)
                         [Cs1-2]])
              assume Cs2 := (~(p In l0))
               let {Cs2-0 :=
                    (forall ?m .
                     ?m In l0 ==> ~(?m divides p));
                    _ :=
                     (conclude Cs2-0
                      pick-any m':N
                       assume P := (m' In l0)
                        let {P0 := (~(m' divides p));
                             P1 :=
                              (conclude
                               ((m' divides p) ==>
                                (m' = one | m' = p))
                               (!left-iff
                                (!uspec A-w5 m')));
                             P2 := (m' =/= p);
                             _ :=
                              (conclude P2
                               (!by-contradiction
                                (m' =/= p)
                                assume (m' = p)
                                 (!absurd
                                  (m' In l0)
                                  (!chain-last
                                   [Cs2
                                    ==> (~(m' In l0))
                                        [(m' = p)]])))));
                             P3 := (m' =/= one);
                             _ :=
                              (conclude P3
                               (!by-contradiction P3
                                assume (m' = one)
                                 (!absurd
                                  (m' In l0)
                                  (!chain-last
                                   [D-w5
                                    ==> (~(m' In l0))
                                        [(m' = one)]])))));
                             P4 := (conclude
                                    (~(m' = one | m' = p))
                                    (!dm (!both P3 P2)))}
                        (!mt P1 P4));
                    Cs2-1 :=
                     (!chain-last
                      [(and (!reflex Output)
                            (!reflex Input)
                            (!reflex (SMSG sq0 p))
                            Cs2-0)
                       ==> ((next (output out l0)
                                  (sieve-recv (SMSG sq0 p)))
                            = (output out (p :: l0)))
                           [Output.next-def]]);
                    Cs2-2 :=
                     (!chain-last
                      [D-w2
                       ==> (config (sitp t (S n0)) =
                            ((s0 ++ (actor Input lsi0))
                                 ++ (actor Output
                                     output out (p :: l0))))
                           [Cs2-1]]);
                    Cs2-3 := (!chain-last
                              [B-w3
                               ==> ((S n0) <= n1)
                                   [Less=.discrete]
                               ==> ((sitp t (S n0)) -->>*
                                    (sitp t n1))
                                   [ITC]]);
                    Cs2-4 := (!chain-last
                              [(B-w0c & B-w0b & Cs2-2 &
                                B-w23 & Cs2-3)
                                ==> (output out (p :: l0)
                                     output-subset lso1)
                                    [output-monotonic]]);
                    Cs2-5 :=
                     (!chain-last
                      [Cs2-4
```

```
            ==> ((output out (p :: l0))
                 output-subset
                 (output out l1))
                [D2-w1]
            ==> (forall ?x .
                 ?x In (p :: l0) ==> ?x In l1)
                [output-subset-ext]]);
         Cs2-6 := (conclude (p In (p :: l0))
                   (!uspec* In.head [p l0]))}
 (!chain-last
    [Cs2-6
     ==> (p In l1)
         [Cs2-5]])));
D-w7a :=
 (!chain-last
  [(IC t)
   ==> (forall ?n ?s ?l ?x .
        (config (sitp t ?n) =
        ?s ++ (actor Output (output out ?l)) &
        ?x In ?l & ?x > two)
        ==>
        (exists ?n' .
         ?n' < ?n &
         (output-about-to-receive (sitp t ?n')
                                  ?x)))
       [must-have-received]]);
D-w8a :=
 (!chain-last
  [(B-w22 & B-w24 & B-w25)
   ==> (exists ?ls ?s .
        (config (sitp t n1) =
         ?s ++ (actor Output ?ls)
            ++ (MSG (IOSEQ n) n) &
         ?ls ready-to (sieve-recv (IOMSG n)) &
          ((sitp t (S n1)) =
           ((sitp t n1) then
            (sieve-recv (IOMSG n))))))
       [existence]
   ==> (output-about-to-receive (sitp t n1) n)
```

```
                    [output-about-to-receive-def]]);
                D-w8b :=
                 (!chain-last
                  [B-w23
                   ==> (config (sitp t n1) =
                        (s1 ++ (actor Input lsi1)
                            ++ (MSG sq1 n))
                            ++ (actor Output (output out l1)))
                       [D2-w1]]);
                D-w7 := (~(n In l1));
                _ :=
                 (!by-contradiction D-w7
                  assume (n In l1)
                   let {P-1 :=
                        (!chain-last
                         [D-w8b
                          ==> ((config (sitp t n1) =
                               (s1 ++ (actor Input lsi1)
                                   ++ (MSG sq1 n))
                                   ++ (actor Output
                                       (output out l1)))
                                 & (n In l1) & (n > two))
                              [augment]
                          ==> (exists ?n' .
                               ?n' < n1 &
                               (output-about-to-receive
                                 (sitp t ?n') n))
                              [D-w7a]])}
                   pick-witness n' for P-1 P-w
                    let {Pu := (!uspec A000 n');
                         P-w1 := (!chain-last
                                  [(n' < n1)
                                   ==> (n' =/= n1)
                                       [Less.not-equal]]);
                         P-w2 := (!chain-last
                                  [(output-about-to-receive
                                    (sitp t n') n)
                                   ==> (Pu & B-w0b &
                                        (output-about-to-receive
                                         (sitp t n') n) &
                                        D-w8a)
                                       [augment]
                                   ==> (n' = n1)
                                       [output-receives-once]])}
                    (!absurd P-w1 (!bdn P-w2)));
                D-w8 := (~(n In ls));
                D-w9b := (!chain-last
                           [(A0 & A1 & A-two)
                            ==> (exists ?n' .
                                 ?n' < m &
                                 (output-about-to-receive
                                  (sitp t ?n') n))
                                [D-w7a]]);
                D-w9 := (n1 < m);
                _ :=
                 (conclude D-w9
                  pick-witness n1' for D-w9b T-w
                   let {Tu := (!uspec A000 n1');
                        T-wl := (n1' < m);
                        T-wr := (output-about-to-receive
                                  (sitp t n1') n);
                        T-w1 :=
                         (!chain-last
                          [(Tu & B-w0b & T-wr & D-w8a)
                           ==> (n1' = n1)
                               [output-receives-once]])}
                   (!chain-last
                    [T-wl ==> (n1 < m) [T-w1]]));
                D-w10 := (config (sitp t (S n1)) =
                          (s1 ++ (actor Input lsi1))
                              ++ (actor Output (output out l1)));
                D-tmp :=
                 (!chain-last
                  [(D-w6 & A-w1)
                   ==> (exists ?m . ?m In l1 & ?m divides n)
                       [existence]
                   ==> ((next (output out l1)
                              (sieve-recv (SMSG sq1 n)))
```

```
                              = (output out l1))
                             [Output.next-def-some-divide]]);
                     _ :=
                      (conclude D-w10
                       (!chain-last
                        [B-w26
                         ==> (config (sitp t (S n1)) =
                              ((s1 ++ (actor Input lsi1)) ++
                               (actor Output
                                (next (output out l1)
                                 (sieve-recv (msg sq1 (sm n)))))))
                             [D2-w1]
                         ==> (config (sitp t (S n1)) =
                              ((s1 ++ (actor Input lsi1))
                                   ++ (actor Output (output out l1))))
                             [D-tmp]]));
                     D-w11 :=
                      (!chain-last
                       [((unique-ids config t) &
                         D-w8a & D-w10 & D-w7)
                        ==> (forall ?n' ?s' ?l' .
                             ?n' > n1 &
                             (config (sitp t ?n') =
                              ?s' ++
                               (actor Output (output out ?l')))
                             ==>
                             (~(n In ?l')))
                            [output-filtered]]);
                     _ :=
                      (conclude D-w8
                       (!chain-last
                        [(D-w9 & A0)
                          ==> (~(n In ls))
                              [D-w11]]))}
                (!absurd A1 D-w8)))}
      (!chain-last
        [Prem ==> (Correct (sitp t m))
                  [Correctness.definition]])
```