A Formalization of the Correctness of the Floodsub Protocol

Ankit Kumar Panagiotis Manolios
Northeastern University
Boston, USA
{kumar.anki,p.manolios}@northeastern.edu

Floodsub is a simple, robust and popular peer-to-peer publish/subscribe (pubsub) protocol, where nodes can arbitrarily leave or join the network, subscribe to or unsubscribe from topics and forward newly received messages to all of their neighbors, except the sender or the originating peer. To show the correctness of Floodsub, we propose its specification: Broadcastsub, in which implementation details like network connections and neighbor subscriptions are elided. To show that Floodsub does really implement Broadcastsub, one would have to show that the two systems have related infinite computations. We prove this by reasoning locally about states and their successors using Well-Founded Simulation (WFS). In this paper, we focus on the mechanization of a proof which shows that Floodsub is a simulation refinement of Broadcastsub using WFS. To the best of our knowledge, ours is the first mechanized refinement-based verification of a real world pubsub protocol.

1 Introduction

Peer-to-Peer (P2P) systems are decentralized distributed systems, which constitute overlay networks built over physical networks, such as the Internet [2]. These systems are characterized by self-organization, being able to handle highly dynamic network configurations, with nodes being able to join or leave the overlay network, which allows for scalability in the size of the networks. Publish/Subscribe (*pubsub*) systems are P2P systems that allow (1) consumers of information (subscriber nodes) to query the system, and (2) producers of information (publisher nodes) to publish information to the system. Publishers are able to send messages to multiple recipients without them having to know who the subscribers are. This is achieved by associating subscriptions and messages with *topics*. For example, in a chat room application, each room is a pubsub topic and clients post chat messages to rooms, which are received by all other clients (subscribers) in the room. The Scribe system [5] was a first attempt at providing topic-based pubsub functionality over the Pastry P2P network [33].

A most basic implementation of pubsub systems is *Floodsub* [36, 1]. In Floodsub, nodes are free to leave or join the network. Every node has information about its neighboring nodes and their subscriptions. Whenever a node joins, subscribes to or unsubscribes from a topic, it updates its neighboring nodes. Messages are forwarded to all neighbors that subscribe to the topic of the message, except the source (originating node) of the message, or the node that forwarded this message. Our implementation of Floodsub is based on its specification [36]. However, we did not do any conformance testing or cross validation against existing implementations. And we do not model Ambient Peer Discovery, which is a way for nodes to learn about their neighbors, and is described in the specification as being external to the protocol. Given our model of Floodsub protocol, how can we verify that it actually is an implementation of a pubsub system? We need a specification for a pubsub system and some notion of correctness.

We propose *Broadcastsub* [1] as the specification for a P2P pubsub system. Broadcastsub nodes can freely leave and join the network. Nodes maintain a list of topics they subscribe to. However, there is no notion of neighboring nodes. Messages are broadcasted “magically” to all the subscribers in a single

Submitted to:
ACL2 Workshop 2025

transition. Notice that Broadcastsub is the simplest P2P pubsub system, where implementation details like subscription updates, neighboring nodes and their subscriptions are abstracted. In this paper, we focus on the mechanization of the proof that Floodsub is a simulation refinement [31] of Broadcastsub using Well-Founded Simulation (WFS) [23] in ACL2 Sedan (ACL2S) [12, 6].

To show that Floodnet implements Broadcastnet, we will prove that Floodnet is a simulation refinement of Broadcastnet. Why are we proving a simulation refinement? Because we are comparing two P2P systems at different levels of abstraction. In a Broadcastsub network, a message broadcast propagates to all subscribers (of the message) instantly. However, in a Floodsub network, a message may require several hops from one node to another, until it reaches all of the subscribers. It is often the case that a lower-level implementation takes several steps to match a step of its higher level specification. Proving a WFS guarantees that Floodsub states and related Broadcastsub states have related computations. This notion of correctness implies that the two systems satisfy the same $ACTL^* \setminus X$ [4] properties. WFS proofs are structural and local, requiring proofs about states and their successors, instead of infinite paths, thereby allowing proofs to be amenable to formal verification. This work is a piece of a larger puzzle that allows us to reason about more complex P2P systems using compositional refinement [25, 24], which we want to extend all the way down to Gossipsub [35]. Since our models are public, protocol engineers will be able to easily define/extend their own P2P systems and attempt to show that their model is a refinement of one of our existing ones. Our proof of refinement spells out exactly how the proof breaks, if these conditions are not satisfied. Hence, our contribution can also be used to tag P2P systems with the kinds of network attacks they are prone to, corresponding to the refinement conditions that were not satisfied.

We make the following contributions: (1) Formal, executable, open and public models of Floodsub and Broadcastsub protocols expressed as transition systems, and (2) a mechanized proof in ACL2S showing that Floodsub is a simulation refinement of Broadcastsub. We discuss related work in Section 5; and ours is the first mechanized refinement-based verification of a real world pubsub protocol. While refinement is a standard formal method, it has never been previously applied to P2P pubsub protocols like FloodSub. Our models and proofs are publicly available in our repository [21]. Overall, our code consists of 476 theorems proved, and 10277 lines of lisp code.

Paper Outline. Section 2 describes Floodnet and Broadcastnet models for Floodsub and Broadcastsub, respectively. Section 3 describes the refinement theorem. Section 4 is a discussion about the theorem proving process and effort that went into this proof. Section 5 discusses related work. Section 6 concludes.

2 Model Descriptions

We model Floodsub and its specification Broadcastsub using transition systems consisting of states and transition relations (boolean functions on 2 states) that depend on transition functions. We call our models Floodnet and Broadcastnet respectively. In this section we will explain each of our transition system models in a top-down fashion. The state models are self explanatory. The interesting parts of the following code listings are the transition relations, where we place conditions, not only as sanity checks or for cases, but also as guards to disallow illegal behaviour, like requiring that a peer leaving a network is already in the network. We will discuss such conditions in detail.

2.1 Broadcastnet

The state of a Broadcastnet is stored in a map from peers to their corresponding peer-states. We represent peers by natural numbers. A Broadcastnet peer state is a record consisting of (i) `pubs` : the set of topics in which a peer publishes, (ii) `subs` : the set of topics to which a peer subscribes to, and (iii) `seen` : the set of messages a peer has already processed. We use sorted ACL2 lists containing unique elements to represent sets. Hence, set equality is reduced to list equality. Messages are records consisting of (i) `pld` : a message payload of type string (ii) `tp` : the topic in which this message was published, and (iii) `or` : the originating peer for this message.

```
(defdata s-bn (map peer ps-bn))

(defdata-alias peer nat)

(defdata ps-bn (record
               (pubs . lot)
               (subs . lot)
               (seen . lom)))

(defdata lot (listof topic))

(defdata-alias topic var)

(defdata lom (listof mssg))

(defdata mssg (record
               (pld . string)
               (tp . topic)
               (or . peer)))
```

We will now define the transition relation for Broadcastnet. The relation `rel-step-bn` relates 2 Broadcastnet states `s` and `u` iff `s` transitions to `u`. `rel-step-bn` is an `OR` of all the possible ways `s` can transition to `u`. `rel-skip-bn` represents a transition where `s` chooses to skip, hence `u` is `s`.

```
(definec rel-step-bn (s u :s-bn) :bool
  (v (rel-skip-bn s u)
     (rel-broadcast-bn s u)
     (rel-broadcast-partial-bn s u)
     (rel-subscribe-bn s u)
     (rel-unsubscribe-bn s u)
     (rel-leave-bn s u)
     (rel-join-bn s u)))

(definecd rel-skip-bn (s u :s-bn) :bool
  (== u s))
```

`rel-broadcast-bn` defines a relation between `s` and `u`, where `u` represents the state resulting from broadcasting a message in `s`. The broadcast is modeled as an atomic operation in which all subscribers receive the message simultaneously. The function `(br-mssg-witness s u)` is a witness finding function that calculates the message that was broadcast, if one exists. Since `seen` is a set of messages implemented as an ordered list with unique elements, `br-mssg-witness` utilizes this ordering of unique messages to find the broadcasted message.

The boolean function `broadcast-bn-pre` is a conjunction of the following preconditions: (i) the broadcast message is new, *i.e.*, it is not already found in the `seen` set of any of the peers in `s`, (ii) the originating peer of the message exists in `s`, and (iii) the topic of the broadcast message is one in which the originating peer publishes messages. `broadcast-bn-pre` also appears as an input contract (`:ic`) in the definition of the `broadcast` transition function. `(== u (broadcast (br-mssg-witness s u) s))` in the definition of `rel-broadcast-bn` ensures that the message found by `br-mssg-witness` was the sole message broadcast in `s`. We use the `insert-unique` function within `broadcast-help` to add new messages while preserving order and uniqueness of the `seen` set.

In the following code snippets, we use some ACL2S syntax described as follows. `^`, `v` and `!` are macros for `and`, `or` and `not` respectively. In a `property` form, the form following the keyword `:h` is the hypothesis, while the form following the keyword `:b` is the body. `(nin a x)` stands for `(not (in a x))`. `:match` is a powerful ACL2S pattern matching capability which supports predicates, including recognizers automatically generated by `defdata`, disjunctive patterns and patterns containing arbitrary code [30]. For more expressive pattern matching, `!` is used for literal match while `&` is used as a wildcard. In a function definition form, `:ic` and `:oc` abbreviate `:input-contract` and `:output-contract` respectively.

```
(definecd rel-broadcast-bn (s u :s-bn) :bool
  (^ (br-mssg-witness s u)
     (broadcast-bn-pre (br-mssg-witness s u) s)
     (== u (broadcast (br-mssg-witness s u) s))))

(definec broadcast-bn-pre (m :mssg s :s-bn) :bool
  (b* ((origin (mget :or m))
       (origin-st (mget origin s)))
    (^ (new-bn-mssgp m s)
       origin-st
       (in (mget :tp m)
           (mget :pubs origin-st)))))

(definec br-mssg-witness (s u :s-bn) :maybe-mssg
  (cond
   ((v (endp s) (endp u)) nil)
   ((== (car s) (car u)) (br-mssg-witness (cdr s) (cdr u)))
   (t (car (set-difference-equal (mget :seen (cdar u))
                                 (mget :seen (cdar s)))))))

(defdata maybe-mssg (v nil mssg))

(definecd new-bn-mssgp (m :mssg s :s-bn) :bool
  (v (endp s)
     (^ (nin m (mget :seen (cdar s)))
        (new-bn-mssgp m (cdr s)))))

(definecd broadcast (m :mssg s :s-bn) :s-bn
  :ic (broadcast-bn-pre m s)
  (broadcast-help m s))

(definecd broadcast-help (m :mssg st :s-bn) :s-bn
  (match st
```

```
    (() nil)
    (((p . pst) . rst)
     (cons `(,p . ,(if (v (in (mget :tp m) (mget :subs pst))
                          (== p (mget :or m)))
                       (mset :seen
                             (insert-unique m (mget :seen pst))
                             pst)
                     pst))
           (broadcast-help m rst)))))

(definec insert-unique (a :all x :tl) :tl
  (match x
    (() (list a))
    ((!a . &) x)
    ((e . es) (if (<< a e) (cons a x) (cons e (insert-unique a es))))))
```

We will explain `rel-broadcast-partial-bn`, and its necessity when discussing the proof of correctness later in the paper. `rel-subscribe-bn` and `rel-unsubscribe-bn` relate states `s` and `u` where `u` represents the state obtained after a peer in `s` subscribes to or unsubscribes from a set of topics, respectively. `bn-topics-witness` calculates the peer and the set of topics it subcribes to or unsubscribes from, if there exists such peer. Notice that we reuse `bn-topics-witness` in the definition of `rel-unsubscribe-bn`, with the arguments reversed, so as to find the topics that are subscribed to in `s`, but not in `u`. The calculated set of topics are unioned with or removed from the existing set of peer topic subscriptions of the calculated peer, based on whether it is subscribing or unsubscribing. The definition of `unsubscribe-bn` is analogous to that of `subscribe-bn` and is hence omitted.

```
(definecd rel-subscribe-bn (s u :s-bn) :bool
  (^ (bn-topics-witness s u)
     (mget (car (bn-topics-witness s u)) s)
     (== u (subscribe-bn (car (bn-topics-witness s u))
                         (cdr (bn-topics-witness s u))
                         s))))

(definecd rel-unsubscribe-bn (s u :s-bn) :bool
  (^ (bn-topics-witness u s)
     (mget (car (bn-topics-witness u s)) s)
     (== u (unsubscribe-bn (car (bn-topics-witness u s))
                           (cdr (bn-topics-witness u s))
                           s))))

(definec bn-topics-witness (s u :s-bn) :maybe-ptops
  (cond
   ((v (endp s) (endp u)) nil)
   ((== (car s) (car u)) (bn-topics-witness (cdr s) (cdr u)))
   ((^ (== (caar s) (caar u))
       (set-difference-equal (mget :subs (cdar u))
                             (mget :subs (cdar s))))
    (cons (caar s)
          (set-difference-equal (mget :subs (cdar u))
                                (mget :subs (cdar s)))))
   (t nil)))
```

```
(defdata maybe-ptops (v nil (cons peer lot)))

(definecd subscribe-bn (p :peer topics :lot s :s-bn) :s-bn
  :ic (mget p s)
  (let ((pst (mget p s)))
    (mset p (mset :subs (union-equal (mget :subs pst) topics) pst) s)))
```

rel-join-bn and rel-leave-bn relate states s and u where u is obtained after a peer joins s or leaves s, respectively. bn-join-witness calculates the peer and its peer-state, if there exists a peer that joins s. Its definition depends on the keys of our Broadcastnet state being in order, which is guaranteed by ACL2s maps. A peer joins a Broadcastnet state when a new default Broadcastnet peer state is set for the corresponding peer. A peer leaves a Broadcastnet state when the entry corresponding to the leaving peer is removed from the state.

```
(definecd rel-join-bn (s u :s-bn) :bool
  (^ (bn-join-witness s u)
     (b* ((p (car (bn-join-witness s u)))
          (pst (cdr (bn-join-witness s u))))
       (^ (! (mget p s))
          (== u (join-bn p (mget :pubs pst) (mget :subs pst) s))))))

(definecd rel-leave-bn (s u :s-bn) :bool
  (^ (bn-join-witness u s)
     (mget (car (bn-join-witness u s)) s)
     (== u (leave-bn (car (bn-join-witness u s)) s))))

(definec bn-join-witness (s u :s-bn) :maybe-ppsbn
  (match (list s u)
    ((() ((q . qst) . &)) `(,q . ,qst))
    ((((p . pst) . rs1) ((q . qst) . rs2))
     (cond
      ((== `(,p . ,pst) `(,q . ,qst)) (bn-join-witness rs1 rs2))
      ((!= q p) `(,q . ,qst))  ;; Joining peer found
      (t nil)))
    (& nil)))

(defdata maybe-ppsbn (v nil (cons peer ps-bn)))

(definecd join-bn (p :peer pubs subs :lot s :s-bn) :s-bn
  :ic (! (mget p s)) ;; Join only if peer does not already exist in state
  (mset p (ps-bn pubs subs '()) s))

(definecd leave-bn (p :peer s :s-bn) :s-bn
  :ic (mget p s) ;; Leave only if peer already exists in state
  (match s
    (((!p . &) . rst) rst)
    ((r . rst) (cons r (leave-bn p rst)))))
```

2.2 Floodnet

A Floodnet peer-state is a record consisting of sets `pubs`, `subs` and `seen` which we described previously in context of Broadcastnet peer-states. It also consists of `pending`, which is a set of messages that have not yet been processed, and `nsubs`, a map from topics to list of peers. `nsubs` stores topic subscriptions for neighboring peers.

```
(defdata s-fn (map peer ps-fn))

(defdata ps-fn
  (record (pubs . lot)
          (subs . lot)
          (nsubs . topic-lop-map)
          (pending . lom)
          (seen . lom)))
```

When a message has not been forwarded to neighboring subscribers (processed) it remains in the `pending` set. Once it is processed, it is added to the `seen` set. In our Floodnet model, `pending` and `seen` are sets of messages, instead of queues. This simplifies the model and allows us to not worry about the order in which messages are received. Related states have equal sets of `seen` messages.

We define the transition relation `rel-step-fn` which relates two Floodnet states `s` and `u` iff `s` transitions to `u`. It encodes all the possible ways `s` can transition to `u`. `rel-skip-fn` represents a transition where `s` chooses to skip, hence `u` is `s`.

```
(definec rel-step-fn (s u :s-fn) :bool
  (v (rel-skip-fn s u)
     (rel-produce-fn s u)
     (rel-forward-fn s u)
     (rel-subscribe-fn s u)
     (rel-unsubscribe-fn s u)
     (rel-leave-fn s u)
     (rel-join-fn s u)))

(definecd rel-skip-fn (s u :s-fn) :bool
  (== u s))
```

`rel-produce-fn` relates `s` and `u` where `u` represents the state obtained after a new message has been produced in `s`. The newly produced message is one of the pending messages in `u`. The boolean function `produce-fn-pre` is a conjunction of the following preconditions: (i) the produced message is new *i.e.*, it is not already found in the `seen` or `pending` sets of any of the peers in `s`, (ii) the originating peer of the message exists in `s`, and (iii) the topic of the produced message is one in which the originating peer publishes messages. The new message is added to the set of pending messages of the originating peer.

```
(definecd rel-produce-fn (s u :s-fn) :bool
  (rel-produce-help-fn s u (fn-pending-mssgs u)))

(definec fn-pending-mssgs (s :s-fn) :lom
  (match s
    (() '())
    (((& . pst) . rst) (union-set (mget :pending pst) (fn-pending-mssgs rst)))))

(definec rel-produce-help-fn (s u :s-fn ms :lom) :bool
  (match ms
```

```
    (() nil)
    ((m . rst) (v (^ (produce-fn-pre m s)
                     (== u (produce-fn m s)))
                  (rel-produce-help-fn s u rst)))))

(definec produce-fn-pre (m :mssg s :s-fn) :bool
  (b* ((origin (mget :or m))
       (origin-st (mget origin s)))
    (^ (new-fn-mssgp m s)
       origin-st
       (in (mget :tp m) (mget :pubs origin-st)))))

(definecd new-fn-mssgp (m :mssg s :s-fn) :bool
  (v (endp s)
     (^ (nin m (mget :seen (cdar s)))
        (nin m (mget :pending (cdar s)))
        (new-fn-mssgp m (cdr s)))))

(definecd produce-fn (m :mssg s :s-fn) :s-fn
  :ic (produce-fn-pre m s)
  (mset (mget :or m)
        (add-pending-psfn m (mget (mget :or m) s)) s))

(definecd add-pending-psfn (m :mssg pst :ps-fn) :ps-fn
  (if (v (in m (mget :pending pst))
         (in m (mget :seen pst)))
      pst
    (mset :pending (cons m (mget :pending pst)) pst)))
```

`rel-forward-fn` relates states `s` and `u` where `u` represents the state obtained after a peer in `s` forwards a pending message. Notice that any of the pending messages in `s` are eligible to be forwarded. Notice also that there can be several peers with a given message pending, and the Floodnet can take several possible transitions to states related to the current state by `rel-forward-fn`. To model this in a constructive and deterministic way, we introduce `find-forwarder` as a skolem function which returns the first peer in the state where a given message is pending. It produces a concrete peer p in the call to `forward-fn`. Its output contract (`:oc`) specifies that the message forwarding peer it returns (i) is a peer in `s`, (ii) possesses the given message in its `pending` set, and (iii) that message is not new in `s`.

The `forward-fn` transition function simultaneously updated the state of the peer that forwards the message, using `update-forwarder-fn` and updates the `pending` sets of the neighboring subscribers by inserting the forwarded message using `forward-help-fn`. Note that messages are forwarded to all the peers subscribing to the topic of the message in the `:nsubs` map. If the forwarding peer records its own subscriptions in `:nsubs`, it can lead to `rel-forward-fn` being infinitely enabled. We will ensure that a peer does not include itself in this map, by considering *good* Floodnet states later in the paper.

```
(definecd rel-forward-fn (s u :s-fn) :bool
    (rel-forward-help-fn s u (fn-pending-mssgs s)))

(definec rel-forward-help-fn (s u :s-fn ms :lom) :bool
  (match ms
    (() nil)
    ((m . rst)
```

```
     (v (^ (in m (fn-pending-mssgs s))
           (== u (forward-fn (find-forwarder s m) m s)))
        (rel-forward-help-fn s u rst)))))

(definec find-forwarder (s :s-fn m :mssg) :peer
    :ic (in m (fn-pending-mssgs s))
    :oc (^ (mget (find-forwarder s m) s)
           (in m (mget :pending (mget (find-forwarder s m) s)))
           (! (new-fn-mssgp m s)))
    (match s
      (((p . &)) p)
      (((p . pst) . rst)
       (if (in m (mget :pending pst)) p (find-forwarder rst m)))))

(definecd forward-fn (p :peer m :mssg s :s-fn) :s-fn
  :ic (^ (mget p s)
         (in m (mget :pending (mget p s))))
  (b* ((tp (mssg-tp m))
       (pst (mget p s))
       (nsubs (mget :nsubs pst))
       (fwdnbrs (mget tp nsubs)))
    (forward-help-fn (update-forwarder-fn p m s) fwdnbrs m)))

(definec update-forwarder-fn (p :peer m :mssg s :s-fn) :s-fn
  (match s
    (() '())
    (((!p . pst) . rst) (cons `(,p . ,(forwarder-new-pst pst m)) rst))
    ((r . rst) (cons r (update-forwarder-fn p m rst)))))

(definecd forwarder-new-pst (pst :ps-fn m :mssg) :ps-fn
  (mset :seen
        (insert-unique m (mget :seen pst))
        (mset :pending
              (remove-equal m (mget :pending pst))
              pst)))

(definecd forward-help-fn (s :s-fn nbrs :lop m :mssg) :s-fn
  (match s
    (() '())
    (((q . qst) . rst)
     (cons (if (in q nbrs)
               `(,q . ,(add-pending-psfn m qst))
             `(,q . ,qst))
           (forward-help-fn rst nbrs m)))))
```

`rel-subscribe-fn` and `rel-unsubscribe-fn` relate states `s` and `u` where `u` represents the state obtained after a peer in `s` subscribes to or unsubscribes from a set of topics, respectively. They are very similar to their Broadcastnet counterparts and hence we omit their definitions.

`rel-join-fn` and `rel-leave-fn` relate states `s` and `u` where `u` represents the state obtained after a peer joins `s` or leaves `s`, respectively. `fn-join-witness` calculates the peer and its peer-state, if there exists a peer that joins `s`, and is analogous to `bn-join-witness`. `rel-join-fn` requires that the joining peer (i) is not already in `s`, and (ii) does not exist in its own `:nsubs` map. The second condition is necessary to prevent peers from endlessly forwarding messages to themselves. `join-fn` depends on `new-joinee-st-fn` which returns the state for a newly joined peer, and on `set-subs-sfn` which updates the `:nsubs` map for each of the neighboring peers of the joining node. For the sake of brevity, we omit the definitions of these helper functions. The `rel-leave-fn` relation, similar to `rel-leave-bn` requires that there is a leaving peer, as calculated by `(fn-join-witness u s)` and that it already exist in the state. Notice that there is another requirement, that a leaving peer has no pending messages. This condition allows for graceful exit of leaving peers, guaranteeing that no pending messages are lost along with them.

```
(definecd rel-join-fn (s u :s-fn) :bool
  (^ (fn-join-witness s u)
     (b* ((p (car (fn-join-witness s u)))
          (pst (cdr (fn-join-witness s u)))
          (nbrs (topic-lop-map->lop (mget :nsubs pst))))
       (^ (! (mget p s))
          (nin p nbrs)
          (== u (join-fn p (mget :pubs pst) (mget :subs pst) nbrs s))))))

(definecd join-fn (p :peer pubs subs :lot nbrs :lop s :s-fn) :s-fn
  :ic (^ (! (mget p s))
         (nin p nbrs))
  (set-subs-sfn nbrs
                subs
                p
                (mset p (new-joinee-st-fn pubs subs nbrs s) s)))

(definecd rel-leave-fn (s u :s-fn) :bool
  (^ (fn-join-witness u s)
     (mget (car (fn-join-witness u s)) s)
     (endp (mget :pending (mget (car (fn-join-witness u s)) s)))
     (== u (leave-fn (car (fn-join-witness u s)) s))))

(definecd leave-fn (p :peer s :s-fn) :s-fn
  :ic (mget p s)
  (match s
    (() '())
    (((!p . &) . rst) rst)
    ((r . rst) (cons r (leave-fn p rst)))))
```

3 Correctness and the Refinement Theorem

We consider simulation refinement [23] as the notion of correctness for Floodnet and show that Floodnet is a simulation refinement of Broadcastnet. The key idea of a simulation refinement is to show that every behavior of the concrete system (Floodnet) is allowed by the abstract system (Broadcastnet). If we prove a WFS refinement then we know that for any infinite computation tree starting from some Floodnet state, we can find a related computation tree in Broadcastnet after applying the refinement map. Another consequence is that we preserve any branching time properties, excluding next time, for example, all properties in $ACTL^* \setminus X$ [4].

The refinement map needs to map Floodnet states to "related" Broadcastnet states. Why do we require a refinement map? Because states in different levels of abstractions may represent data differently, or some implementation details from the lower abstraction may simply be missing in the higher level specification. For example, `:nsubs` and `:pending` appear only in Floodnet peer states, not in Broadcastnet peer states. Using a refinement map is like putting on glasses that let us "see" lower-level concrete states as their corresponding abstract specification states. The refinement map that we use is `f2b` shown below. It maps Floodnet states to Broadcastnet states where pending messages have not yet been broadcasted. This is called as the commitment approach to refinement, since we are mapping to states consisting of only those messages that have been fully propagated in Floodnet and are thus considered committed.

```
(definec f2b (s :s-fn) :s-bn
  (f2b-help s (fn-pending-mssgs s)))

(definec f2b-help (s :s-fn ms :lom) :s-bn
  (if (endp s)
      '()
    (cons `(,(caar s) . ,(f2b-st (cdar s) ms))
          (f2b-help (cdr s) ms))))

(definecd f2b-st (ps :ps-fn ms :lom) :ps-bn
  (ps-bn (mget :pubs ps)
         (mget :subs ps)
         (set-difference-equal (mget :seen ps) ms)))
```

To gain a better understanding of our refinement map, we examine example traces of Floodnet and Broadcastnet in Figure 1. On the left side, we have a trace of a Floodnet, consisting of 3 green colored nodes, numbered 1, 2 and 3. The node numbered 3 is connected to nodes 1 and 2. We show pending messages on the top left of a node, and seen messages on the bottom right. So, in the second Floodnet state shown, node 1 has a pending message `m`, after a `produce-fn` transition. On the right side, we have Broadcastnet states such that for each Floodnet state on the left, we have its refinement map on the right and for each transition on the left, we show a corresponding matching transition on the right.

The transitions on the Floodnet side are as follows : (i) Node 1 produces message `m`; (ii) Node 1 forwards its pending message `m` to its connected neighboring peer 3; (iii) Node 1 leaves the network (iv) Node 2 unsubscribes from `(mssg-tp m)`, which is the message topic (iv) Node 3 unsubscribes from `(mssg-tp m)`, and finally (v) Node 3 forwards `m` to node 2. Notice that `f2b` is a clear refinement map where events like joining and leaving are not masked. Hence, in the corresponding Broadcastnet states, leave and unsubscribe transitions are matched with leave and unsubscribe transitions. However, when the message `m` can no longer be forwarded, and is no longer pending, it needs to be matched by a `broadcast`. But notice that on the Broadcastnet side (a) the originating peer (Node 1) is no longer present, which is required for a `broadcast`, due to `broadcast-bn-pre`, and (b) there are no subscribers of `m` left in the network! This issue arises from the fact that broadcasting a message in Floodnet is a highly fragmented operation, taking place over several message hops during which peers are free to leave, join, subscribe or unsubscribe. With so many moving parts in the network, it becomes impossible to specify which nodes will receive the broadcasted message in the Broadcastnet under the refinement map at the time the message is produced. To solve this problem, we generalize the Broadcastnet specification by adding another transition relation: `rel-broadcast-partial-bn` which allows us to relate 2 states `s` and `u` where `u` represents the state obtained after broadcasting a message in `s`, but only partially. This relation is defined using the `broadcast-partial` transition function, which given a message and a list of peers, sends the message to those peers.

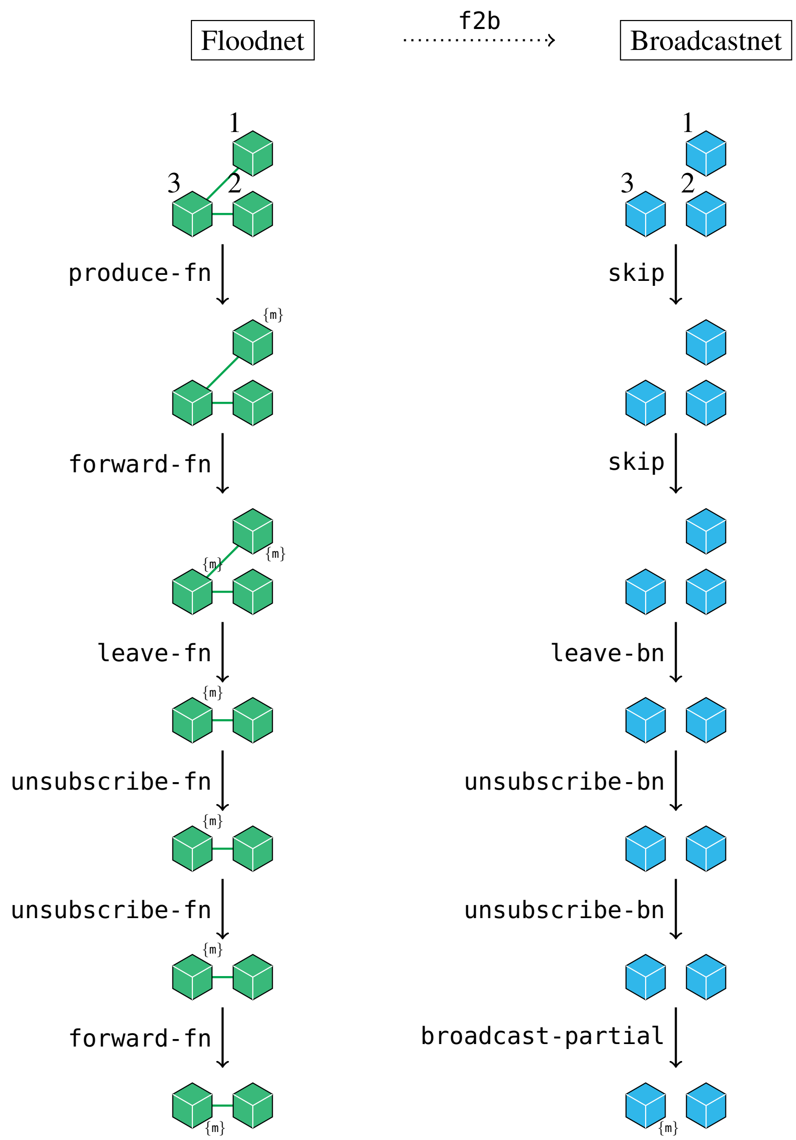

Figure 1: On the left is an example Floodnet trace. Broadcastnet states on the right are refinement maps of the Floodnet states on the left, and every step taken by the Broadcastnet states matches each step taken by the Floodnet states.

When we consider a static configuration where nodes do not change their subscriptions, no existing nodes are leaving, no new nodes are joining, and the network is connected in each topic, the recipients of the message `m` in `broadcast-partial` can be shown to be exactly those in `broadcast` *i.e.*, the subscribers of `(mssg-tp m)`. This aligns with Dijkstra's notion of self-stabilizing distributed systems [11], where the system is guaranteed to reach a legitimate configuration regardless of the initial state. In our case, once the system stabilizes (i.e., peer churn and subscription/unsubscription ceases), the `broadcast-partial` step effectively becomes indistinguishable from a `broadcast` step, reinforcing the view that `broadcast-partial` is a generalization that accommodates transient perturbations while preserving desirable behavior in steady state.

```
(definecd rel-broadcast-partial-bn (s u :s-bn) :bool
  (^ (br-mssg-witness s u)
     (new-bn-mssgp (br-mssg-witness s u) s)
     (== u (broadcast-partial (br-mssg-witness s u)
                              (brd-receivers-bn (br-mssg-witness s u) u)
                              s))))

(definecd broadcast-partial (m :mssg ps :lop s :s-bn) :s-bn
  :ic (new-bn-mssgp m s)
  (broadcast-partial-help m ps s))

(definecd broadcast-partial-help (m :mssg ps :lop st :s-bn) :s-bn
  (match st
    (() nil)
    (((p . pst) . rst)
     (cons `(,p . ,(if (== p (car ps))
                       (mset :seen (insert-unique m (mget :seen pst)) pst)
                     pst))
           (broadcast-partial-help m (if (== p (car ps)) (cdr ps) ps) rst)))))
;; broadcast message receivers in a Broadcastnetwork
(definec brd-receivers-bn (m :mssg s :s-bn) :lop
  (match s
    (() ())
    (((p . pst) . rst) (if (in m (mget :seen pst))
                           (cons p (brd-receivers-bn m rst))
                         (brd-receivers-bn m rst)))))
```

We define `rel-B`, which holds for related states. `rel-B` is defined over a set of states combining both Floodnet and Broadcastnet states, which we define as `borf`. Notice that `rel-B` depends on Floodnet states satisfying the `good-s-fnp` predicate. This predicate ensures that each Floodnet state in the trace satisfies certain invariants. Given space constraints, we only list the invariant properties that hold true for `good-s-fnp` states at the end of the following listing.

```
(defdata borf (v s-bn s-fn))

(definec rel-B (x y :borf) :bool
  (v (rel-wf x y)
     (== x y)))

(definec rel-wf (x y :borf) :bool
  (^ (s-fnp x)
     (s-bnp y)
     (good-s-fnp x)
     (== y (f2b x))))

(definec rel-> (s u :borf) :bool
  (v (^ (s-fnp s) (s-fnp u) (good-rel-step-fn s u))
     (^ (s-bnp s) (s-bnp u) (rel-step-bn s u))))
```

```
(definec good-rel-step-fn (s u :s-fn) :bool
  (^ (good-s-fnp s)
     (good-s-fnp u)
     (rel-step-fn s u)))

;; A good-s-fnp state satisfies 2 predicates
(definec good-s-fnp (s :s-fn) :bool
  (^ (p!in-nsubs-s-fn s) (ordered-seenp s)))

;; Invariant 1: A peer p does not track its own subscriptions in the
;; :nsubs map. So, it can not forward a message to itself.
(propertyd prop=p!in-nsubs-s-fn (p :peer tp :topic s :s-fn)
  :h (^ (mget p s) (p!in-nsubs-s-fn s))
  :b (nin p (mget tp (mget :nsubs (mget p s)))))

;; Invariant 2: :seen components of Floodnet peers are ordered.
(property prop=ordered-seenp-cdar (s :s-fn)
  :h (^ s (ordered-seenp s))
  :b (orderedp (mget :seen (cdar s))))

(definec orderedp (x :tl) :bool
  (match x
    (() t)
    ((&) t)
    ((a . (b . &)) (^ (<< a b) (orderedp (cdr x))))))
```

Proving the WFS refinement requires proving the following three theorems: (i) WFS1 states that concrete Floodnet states are related to their corresponding Broadcastnet states under the refinement map (by relation `rel-B`), (ii) WFS2 states that the labelling function labels related states equally, and (iii) WFS3 states that given related states `s` and `w`, and given `s` steps to `u` under the transition relation, there exists a state, say `v`, such that `u` is matched by a step from `w` going to `v` such that `w` is related to `v`.

```
;; WFS1
(property b-maps-f2b (s :s-fn)
  :h (good-s-fnp s)
  :b (rel-B s (f2b s)))

;; WFS2. L is the labelling functions of our combined transition system
(definec L (s :borf) :borf
  (match s
    (:s-bn s)
    (:s-fn (f2b s))))

(property wfs2 (s w :borf)
  :h (rel-B s w)
  :b (== (L s) (L w)))

 ;; WFS3
(defun-sk exists-v-wfs (s u w)
  (exists (v)
    (^ (rel-> w v)
       (rel-B u v))))
```

```
(property wfs3 (s w u :borf)
  :h (^ (rel-B s w)
        (rel-> s u))
  :b (exists-v-wfs s u w))

;; Witness generating function for v
(definec exists-v (s u w :borf) :borf
  :ic (^ (rel-B s w)
         (rel-> s u))
  (if (null s)
      (if (null u)
          nil
        (exists-nil-v u))
    (exists-cons-v s u w)))

;; when w is nil
(definec exists-nil-v (u :borf) :borf
  :ic (^ u (rel-> nil u))
  (match u
    (:s-fn (exists-v1 nil u))
    (:s-bn u)))

;; when w is not nil
(definec exists-cons-v (s u w :borf) :borf
  :ic (^ s (rel-B s w) (rel-> s u))
  (cond
   ((^ (s-bnp s) (s-bnp w)) u)
   ((^ (s-fnp s) (s-bnp w)) (exists-v1 s u))
   ((^ (s-fnp s) (s-fnp w)) u)))

;; when s and u are Floodnet states
(definec exists-v1 (s u :s-fn) :s-bn
  :ic (good-s-fnp s)
  (cond
   ((rel-skip-fn s u) (f2b s))
   ((^ (rel-forward-fn s u)
       (!= (f2b s) (f2b u)))
    (broadcast-partial (br-mssg-witness (f2b s) (f2b u))
                       (brd-receivers-bn (br-mssg-witness (f2b s) (f2b u))
                                         (f2b u))
                       (f2b s)))
   (t (f2b u))))
```

4 Proof Organization

Given that we define our models using transition functions from states to states, whereas the refinement theorem is expressed in terms of transition relations, proving the monolithic refinement theorem can be a daunting task. In this section, we describe how we approached the mechanization of the refinement proof. The entire codebase can be logically partitioned into four stages:

- **State models and transition functions**: State models are described using `defdata`, and transition functions on the state models are described using `definec` and appear in files `bn-trx.lisp` and `fn-trx.lisp`. Apart from the input state, functions may accept additional arguments. For example `forward-fn` 2.2 accepts a peer `p` along with a message `m` that `p` forwards. These functions usually have input contracts to ensure that the extra arguments satisfy certain properties, for example, `p` should be a peer in the state `s`, and it should have `m` in its `pending` set of messages.
- **Properties of functions under the refinement map**: Given that we have defined functions that accept states and output states, we then prove theorems relating Floodnet states to their corresponding Broadcastnet states under the refinement map in file `f2b-commit.lisp`. For example, here is one such theorem:

```
(property prop=forward-fn (p :peer m :mssg s :s-fn)
 :h (^ (mget p s)
       (in m (mget :pending (mget p s)))
       (== (fn-pending-mssgs (forward-fn p m s))
           (fn-pending-mssgs s)))
 :b (== (f2b (forward-fn p m s))
        (f2b s)))
```

  Notice that these theorems still depends on variables that have not yet been skolemized, so as to ease the theorem proving process.
- **Transition relations**: We define the transitions relations for both Broadcastnet and Floodnet in `trx-rels.lisp`. Transition relations are boolean functions over two state variables, and hence, we also define witness functions for non-state variables appearing in the transition functions. In our running example, we instantiate `p` with `(find-forwarder s m)` and `m` could be any one of the pending messages in the state.
- **Combined states, transitions relations and correctness theorems**: Finally we prove the WFS theorems and their helper properties in `f2b-sim-ref.lisp`.

During development, some of the previous iterations of our model deviated from the metatheory. For example, in one iteration of the final theorem, the restrictions on the transition relations, which serve as guards against illegal behaviors, emerged as part of the hypotheses of WFS3. It forced us to understand the nature of good states, and to derive the required hypotheses from the invariants of the good states. In an another iteration of our models, the transition relations corresponding to each of the transitions a model can make, was augmented with natural numbers, such that transitions on the Floodnet states were matched by transitions on the Broadcastnet states bearing the same number. Eventually, this arrangement seemed unnecessary because even without the natural numbers, the theorem prover was able to pick the required transition based on theorems proved on them, and because of their definitions being disabled. Hence we would recommend to stick to the metatheory when implementing proofs of refinement, and always write the top level theorems, before embarking on proving lower-level theorems.

5 Related Work

Proof mechanization in context of P2P systems has been explored previously. Azmy et. al. [3] formally verified a safety property of Pastry, a P2P Distributed Hash Table (DHT), in TLA+ [22]. The safety property is that of correct delivery, which states that at any point in time, there is at most one node that answers a lookup request for a key, and this node must be the closest live node to that key. Their proof assumes that nodes never fail, which is a likely event in any P2P system. Zave [41] utilized the Alloy tool [14] to produce counter-examples to show that no published version of Chord is correct w.r.t. the liveness property of the Chord ring-maintenance protocol: that the protocol can eventually repair all disruptions in the ring structure, given ample time and no further disruptions while it is working. Kumar et. al. modeled the Gossipsub [35] P2P protocol in ACL2S, formulated safety properties for its scoring function and showed using counter-examples that for some applications like Ethereum which configure Gossipsub in a particular way, it is possible for Sybil nodes to violate those properties, thereby creating large scale partition or eclipse attacks on the network [19, 18]. To the best of our knowledge, none of the previous work has attempted to prove the correctness of a P2P system by showing it as a refinement of a higher level specification.

There exist several provers to formally check properties of distributed systems, such as Dafny [13], TLA+, Ivy [32] and DistAlgo [34]. They operate by reducing a given specification to a decidable logic formula expressed entirely in First Order Logic. The basic tactic involves forming a conjunction of protocol invariants, invert it, and then using an SMT solver to (possibly) search for a counterexample. The issue in such systems is a lack of expressivity, which does not allow capturing properties over infinite traces. Another issue is that our models can be arbitrary, with nodes leaving and joining and with arbitrary pending messages in transit across a network. And we are reasoning about all possible behaviors of the protocol, which could not be done if we were to be limited to a decidable fragment of logic.

We wrote our models and proved our theorems in ACL2S. The ACL2 Sedan (ACL2S) [12, 6] is an extension of the ACL2 theorem prover[15, 16, 17]. On top of the capabilities of ACL2, ACL2S provides the following: (1) A powerful type system via the `defdata` data definition framework [9] and the `definec` and `property` forms, which support typed definitions and properties. (2) Counterexample generation capability via the `cgen` framework, which is based on the synergistic integration of theorem proving, type reasoning and testing [8, 10, 7]. (3) A powerful termination analysis based on calling-context graphs [29] and ordinals [26, 27, 28]. (4) An (optional) Eclipse IDE plugin [6]. (5) The ACL2S systems programming framework (ASPF) [40] which enables the development of tools in Common Lisp that use ACL2, ACL2S and Z3 as a service [38, 37, 20, 39].

6 Conclusions and Future Work

In this paper, we described our ACL2S models for Broadcastsub and Floodsub, Broadcastnet and Floodnet respectively, and proposed Broadcastnet as a specification of Floodnet. For both the models, we explained our transition systems (including state and transition relations) and design decisions. We described our refinement map `f2b`, the combined transition system and the equivalence relation `rel-B` relating related states. Finally we explained the refinement theorem.

In the future we would like to show that in a static configuration where a Floodnet is connected in each of the topics, a `forward-fn` transition can be matched by either a `broadcast-partial` or a `broadcast` transition. We would also like to refine Floodnet progressively until we approach a specification close to Gossipsub. By contrasting this lowest layer of our refinement chain to Gossipsub, we will be able to find and explain security issues in Gossipsub from a refinement point of view.

Acknowledgements We thank the anonymous reviewers for their thoughtful feedback and suggestions, which helped improve the quality and clarity of this work.

References

[1] *What is Publish/Subscribe*. `https://docs.libp2p.io/concepts/pubsub/overview/`. Accessed 12 May 2023.

[2] Ioannis Aekaterinidis & Peter Triantafillou (2018): *Peer-to-Peer Publish-Subscribe Systems*. In: *Encyclopedia of Database Systems, Second Edition*, doi:10.1007/978-1-4614-8265-9_1221.

[3] Noran Azmy, Stephan Merz & Christoph Weidenbach (2016): *A Rigorous Correctness Proof for Pastry*. In: *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, doi:10.1007/978-3-319-33600-8_5.

[4] Michael C. Browne, Edmund M. Clarke & Orna Grumberg (1988): *Characterizing Finite Kripke Structures in Propositional Temporal Logic*. doi:10.1016/0304-3975(88)90098-9.

[5] Miguel Castro, Peter Druschel, A-M Kermarrec & Antony IT Rowstron (2002): *SCRIBE: A large-scale and decentralized application-level multicast infrastructure*. *IEEE Journal on Selected Areas in communications*.

[6] Harsh Chamarthi, Peter C. Dillinger, Panagiotis Manolios & Daron Vroon (2011): *The "ACL2" Sedan Theorem Proving System*. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, doi:10.1007/978-3-642-19835-9_27.

[7] Harsh Raju Chamarthi (2016): *Interactive Non-theorem Disproving*. Ph.D. thesis, Northeastern University, doi:10.17760/D20467205.

[8] Harsh Raju Chamarthi, Dillinger Peter C., Matt Kaufmann & Panagiotis Manolios (2011): *Integrating testing and interactive theorem proving*. doi:10.4204/EPTCS.70.1.

[9] Harsh Raju Chamarthi, Dillinger Peter C. & Panagiotis Manolios (2014): *Data Definitions in the ACL2 Sedan*. *ACL2*.

[10] Harsh Raju Chamarthi & Panagiotis Manolios (2011): *Automated specification analysis using an interactive theorem prover*. In Per Bjesse & Anna Slobodová, editors: *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11*, FMCAD Inc., pp. 46–53. Available at `http://dl.acm.org/citation.cfm?id=2157665`.

[11] Edsger W. Dijkstra (1974): *Self-stabilizing Systems in Spite of Distributed Control*. *Commun. ACM*, doi:10.1145/361179.361202.

[12] Peter C. Dillinger, Panagiotis Manolios, Daron Vroon & J. Strother Moore (2007): *ACL2s: "The ACL2 Sedan"*. In: *Proceedings of the 7th Workshop on User Interfaces for Theorem Provers (UITP 2006)*, doi:10.1016/j.entcs.2006.09.018.

[13] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty & Brian Zill (2015): *IronFleet: proving practical distributed systems correct*. In: *Proceedings of the 25th Symposium on Operating Systems Principles*, doi:10.1145/2815400.2815428.

[14] Daniel Jackson (2019): *Alloy: a language and tool for exploring software designs*. doi:10.1145/3338843.

[15] Matt Kaufmann, Panagiotis Manolios & J Strother Moore (2000): *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, doi:10.1007/978-1-4615-4449-4.

[16] Matt Kaufmann, Panagiotis Manolios & J Strother Moore (2000): *Computer-Aided Reasoning: Case Studies*. Kluwer Academic Publishers, doi:10.1007/978-1-4757-3188-0.

[17] Matt Kaufmann & J Strother Moore (2022): *ACL2 homepage*. Available at `https://www.cs.utexas.edu/users/moore/acl2/`.

[18] Ankit Kumar, Max von Hippel, Panagiotis Manolios & Cristina Nita-Rotaru (2023): *Verification of GossipSub in ACL2s*. In: *International Workshop on the ACL2 Theorem Prover and Its Applications*, doi:10.4204/EPTCS.393.10.

[19] Ankit Kumar, Max von Hippel, Panagiotis Manolios & Cristina Nita-Rotaru (2024): *Formal Model-Driven Analysis of Resilience of GossipSub to Attacks from Misbehaving Peers*. In: *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*, doi:10.1109/SP54263.2024.00017.

[20] Ankit Kumar & Panagiotis Manolios (2021): *Mathematical Programming Modulo Strings*. In: *Formal Methods in Computer Aided Design, FMCAD*, doi:10.34727/2021/ISBN.978-3-85448-046-4_36.

[21] Ankit Kumar & Panagiotis Manolios (2025): *Proof of Refinement of Floodsub*. Available at `https://github.com/ankitku/FloodsubRef`. In submission.

[22] Leslie Lamport (2002): *Specifying systems: the TLA+ language and tools for hardware and software engineers*.

[23] Panagiotis Manolios (2001): *Mechanical Verification of Reactive Systems*. Ph.D. thesis, The University of Texas at Austin, Department of Computer Sciences, Austin TX.

[24] Panagiotis Manolios & Sudarshan K. Srinivasan (2004): *Automatic Verification of Safety and Liveness for XScale-Like Processor Models Using WEB Refinements*. In: *Design, Automation and Test in Europe Conference and Exposition, DATE*, doi:10.1109/DATE.2004.1268844.

[25] Panagiotis Manolios & Sudarshan K. Srinivasan (2008): *A Refinement-Based Compositional Reasoning Framework for Pipelined Machine Verification*. *IEEE Trans. Very Large Scale Integr. Syst.*, doi:10.1109/TVLSI.2008.918120.

[26] Panagiotis Manolios & Daron Vroon (2003): *Algorithms for Ordinal Arithmetic*. In: *Conference on Automated Deduction CADE*, doi:10.1007/978-3-540-45085-6_19.

[27] Panagiotis Manolios & Daron Vroon (2004): *Integrating Reasoning about Ordinal Arithmetic into ACL2*. In: *Formal Methods in Computer-Aided Design FMCAD*, LNCS, Springer–Verlag, doi:10.1007/978-3-540-30494-4_7.

[28] Panagiotis Manolios & Daron Vroon (2005): *Ordinal Arithmetic: Algorithms and Mechanization*. *Journal of Automated Reasoning*, doi:10.1007/s10817-005-9023-9.

[29] Panagiotis Manolios & Daron Vroon (2006): *Termination Analysis with Calling Context Graphs*. In: *Computer Aided Verification CAV*, doi:10.1007/11817963_36.

[30] Pete Manolios (2023): *Reasoning About Programs*. Available at `https://www.ccs.neu.edu/home/pete/courses/Logic-and-Computation/2023-Fall/lectures.html`. Lecture notes, Northeastern University, CS 2800: Logic and Computation, Accessed 21 Apr 2025.

[31] Robin Milner (1971): *An Algebraic Definition of Simulation Between Programs*. In: *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*. Available at `http://ijcai.org/Proceedings/71/Papers/044.pdf`.

[32] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv & Sharon Shoham (2016): *Ivy: safety verification by interactive generalization*. doi:10.1145/2908080.2908118.

[33] Antony I. T. Rowstron & Peter Druschel (2001): *Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems*. In: *International Conference on Distributed Systems Platforms*.

[34] Kumar Shivam, Vishnu Paladugu & Yanhong A. Liu (2023): *Specification and Runtime Checking of Derecho, A Protocol for Fast Replication for Cloud Services*. doi:10.1145/3584684.3597275.

[35] Dimitris Vyzovitis: *gossipsub: An extensible baseline pubsub protocol*. `https://github.com/libp2p/specs/blob/master/pubsub/gossipsub/README.md`. Accessed 28 Nov 2022.

[36] Dimitris Vyzovitis (2020): *GossipSub v1.0: An extensible baseline pubsub protocol*. `https://github.com/libp2p/specs/blob/master/pubsub/gossipsub/gossipsub-v1.0-old.md`. Accessed 23 Jan 2025.

[37] Andrew T. Walter, Benjamin Boskin, Seth Cooper & Panagiotis Manolios (2019): *Gamification of Loop-Invariant Discovery from Code*. In: *Proceedings of the Seventh AAAI Conference on Human Computation and Crowdsourcing, HCOMP*, doi:10.1609/HCOMP.V7I1.5277.

[38] Andrew T. Walter, David A. Greve & Panagiotis Manolios (2022): *Enumerative Data Types with Constraints*. In: *Formal Methods in Computer-Aided Design, FMCAD*, doi:10.34727/2022/ISBN.978-3-85448-053-2_-25.

[39] Andrew T. Walter, Ankit Kumar & Panagiotis Manolios (2023): *Proving Calculational Proofs Correct*. In: *Proceedings of the 18th International Workshop on the ACL2 Theorem Prover and Its Applications*, doi:10.4204/EPTCS.393.11.

[40] Andrew T. Walter & Panagiotis Manolios (2022): *ACL2s Systems Programming*. In: *Workshop on the ACL2 Theorem Prover and its Applications*, doi:10.4204/EPTCS.359.12.

[41] Pamela Zave (2012): *Using lightweight modeling to understand chord*. *Comput. Commun. Rev.*, doi:10.1145/2185376.2185383.