A MECHANIZED PROOF OF BOUNDED CONVERGENCE TIME FOR THE DISTRIBUTED PERIMETER SURVEILLANCE SYSTEM (DPSS) ALGORITHM A

DAVID GREVE, JEN DAVIS (COLLINS AEROSPACE)
LAURA HUMPHREY (AFRL)
MAY 2022

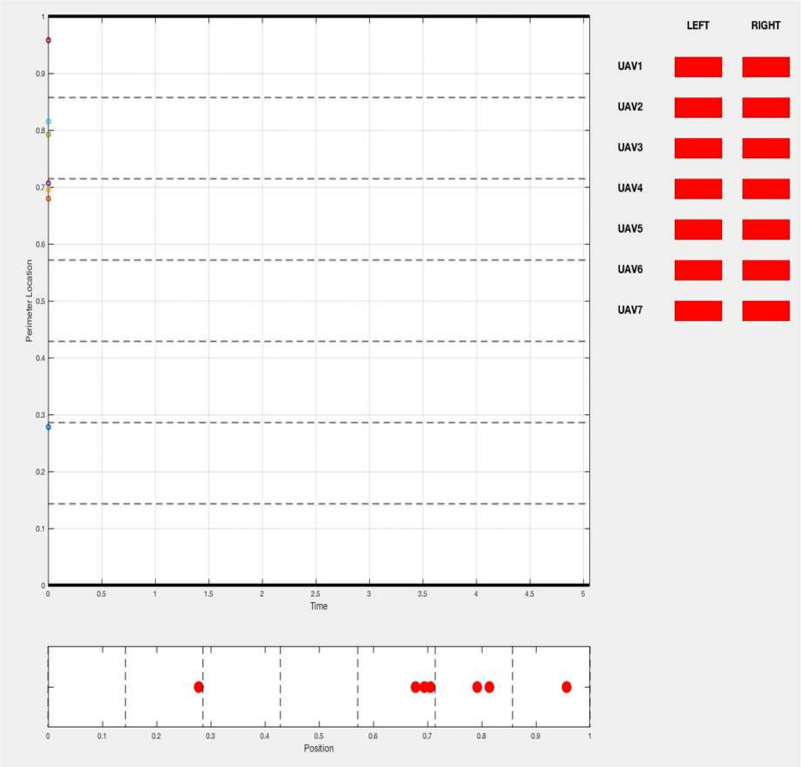
OUTLINE

- **Background**
- DPSS-A Model
- Overview of Convergence Proof
- Specialized ACL2 Utilities
- Conclusion

Collins Aerospace

DISTRIBUTED PERIMETER SURVEILLANCE

- Given a Linear Perimeter ..
- Given a Fleet of **Autonomous** UAVs ..
  - With Limited Communication Ability ..
  - All Traveling at same speed ..
- Find a **Distributed Algorithm**
  - To Surveil the Perimeter
  - That **Converges** such that
    - Each of N UAVs surveys 1/N of perimeter
  - **In Bounded Time**
- DPSS: Distributed Perimeter Surveillance System
  - Kingston, Beard, Holt



Collins Aerospace

DPSS-B ALGORITHM

- UAVs
  - Common, fixed velocity
  - Communicate only when co-incident
- Linear Perimeter
  - Detectable End Points
- Coordination Variables
  - Unknown Perimeter Size
  - Unknown #UAVs
    - Conceptually, UAVs can enter/leave
  - Unknown Relative Position
- Coordination Variable Agreement : 3T (?)

Algorithm B

if agent i (left) rendezvous with neighbor j (right) **then**
  Update perimeter length and team size:
    $P_{R_i} = P_{R_j}$
    $N_{R_i} = N_{R_j} + 1$
  Calculate team size $N = N_{R_i} + N_{L_i} + 1$.
  Calculate perimeter length $P = P_{R_i} + P_{L_i}$.
  Calculate relative index $n = N_{L_i} + 1$.
  Calculate segment endpoints:
    $\mathcal{S}_i = \left\{ \lfloor n - \frac{1}{2}(-1)^n \rfloor P/N, \lfloor n + \frac{1}{2}(-1)^n \rfloor P/N \right\}$.
  Communicate $\mathcal{S}_i$ to neighbor j and receive $\mathcal{S}_j$.
  Calculate shared border position $p_{i,j} = \mathcal{S}_i \cap \mathcal{S}_j$.
  Travel with neighbor j to shared border $p_{i,j}$.
  Set direction to monitor own segment.
else if reached left perimeter endpoint **then**
  Reset perimeter length to the left $P_{L_i} = 0$.
  Reset team size to the left $N_{L_i} = 0$.
  Reverse direction.
else if reached right perimeter endpoint **then**
  Reset perimeter length to the right $P_{R_i} = 0$.
  Reset team size to the right $N_{R_i} = 0$.
  Reverse direction.
else
  Continue in current direction keeping track of traversed perimeter length.
end if

T = Time to traverse perimeter = N

Collins Aerospace

DPSS-A ALGORITHM

- UAVs
  - Common, fixed velocity
  - Communicate only when co-incident
- Linear Perimeter
  - **Known** End Points
- Coordination Variables
  - **Known** Perimeter Size
  - **Known**, fixed #UAVs
  - **Known** Relative Position
- Convergence: 2T (?)

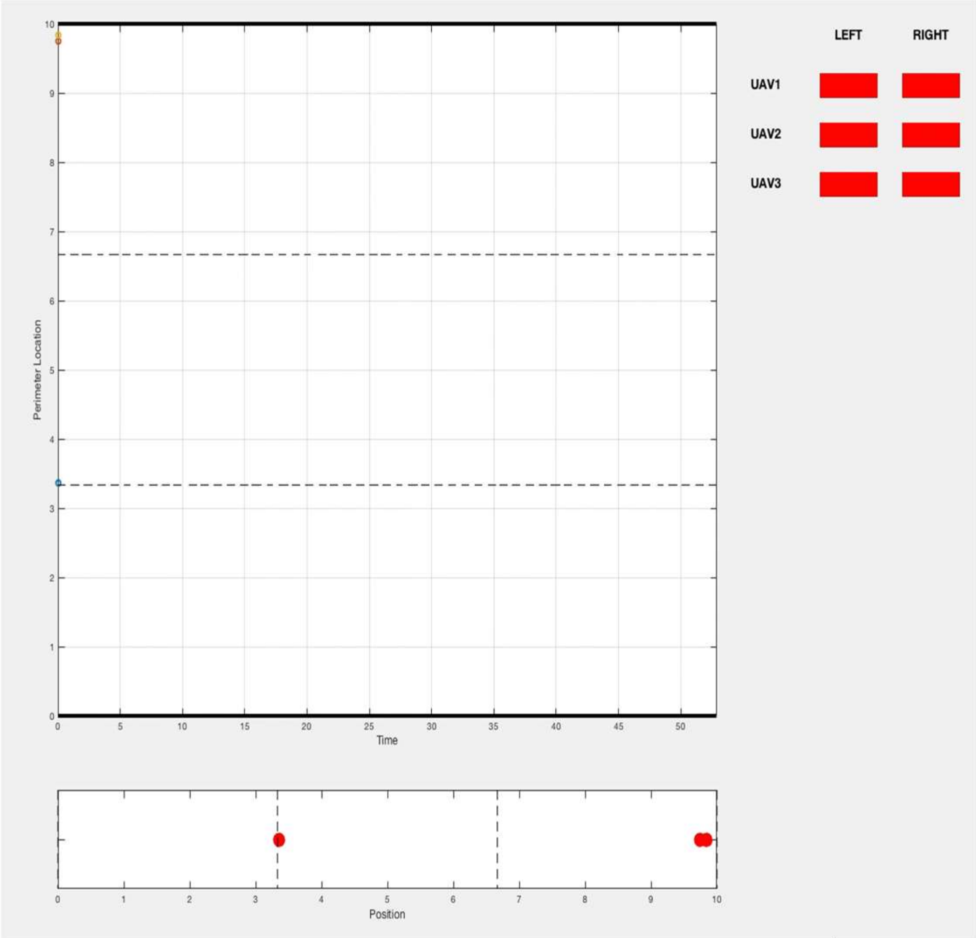
DPSS-B reduces to DPSS-A after 3T (?)

```
if UAV i rendezvous with neighbor j then
  Travel with neighbor j to shared boarder position
  Set direction to monitor own segment
else if reached perimeter endpoint then
  Reverse direction
else
  Continue in current direction
end if
```

Algorithm A

HISTORY

- 2008
  - "Decentralized perimeter surveillance using a team of UAVs"
    - Kingston, Beard, Holt
  - Hand proof for bounded convergence time: 3T + 2T
- 2019
  - "When Human Intuition Fails: Using Formal Methods to Find an Error in the 'Proof' of a Multi-Agent Protocol"
    - Davis, Kingston, Humphrey
  - Model Checker found counterexample to 3T time bound
  - Proved DPSS-B (4+1/3)T convergence
    - 3 UAVs, 80 cores, 512G, 1-11 days
- 2021
  - "Progress on a perimeter surveillance problem"
    - Avigad, Van Doorn
  - Hand Proof of DPSS-A convergence
  - Improved convergence bound from 2T to 2T-1
- 2022
  - ACL2 Mechanized Proof of DPSS-A
    - N UAV's, 1 core, 8G. 30 min



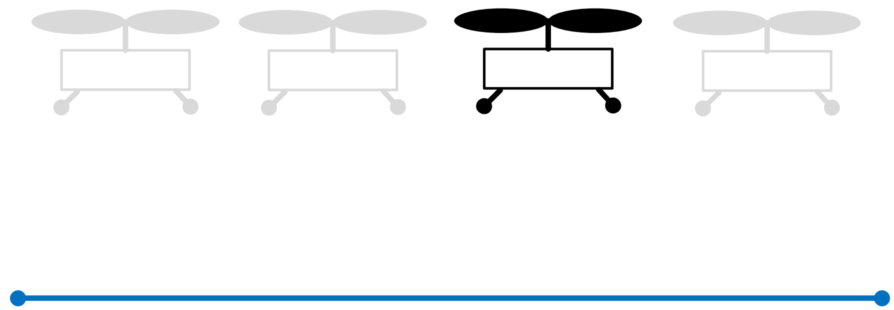
Collins Aerospace

OUTLINE

- Background
- **DPSS-A Model**
- Overview of Convergence Proof
- Specialized ACL2 Utilities
- Conclusion

ACL2 MODEL OF DPSS-A

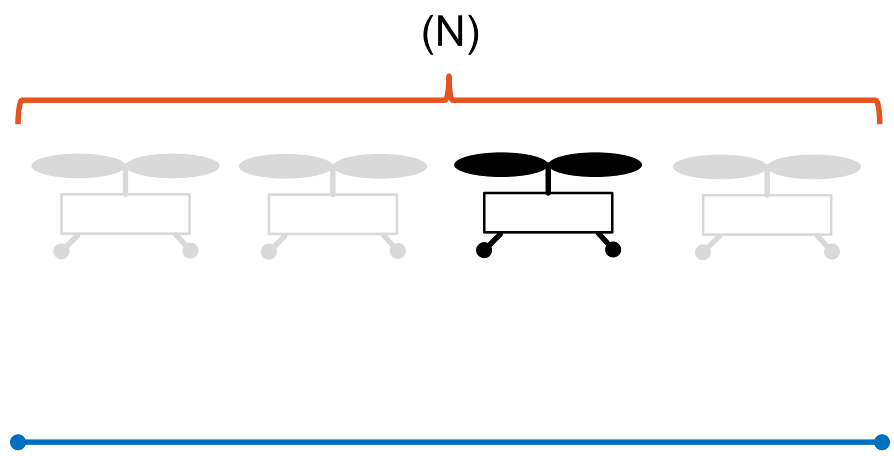
- Global Parameters
  - Number of UAVs: (N)
  - Size of perimeter: (P)
  - Segment Length: (P) / (N)
- UAV-p
  - UAV identifier
    - Natural Number [0 .. (N)-1]
  - Location on perimeter
  - Direction of motion
- UAV segment
  - Left Boundary
    - UAV.id * Segment Length
  - Right Boundary
    - Left Boundary + Segment Length



Collins Aerospace

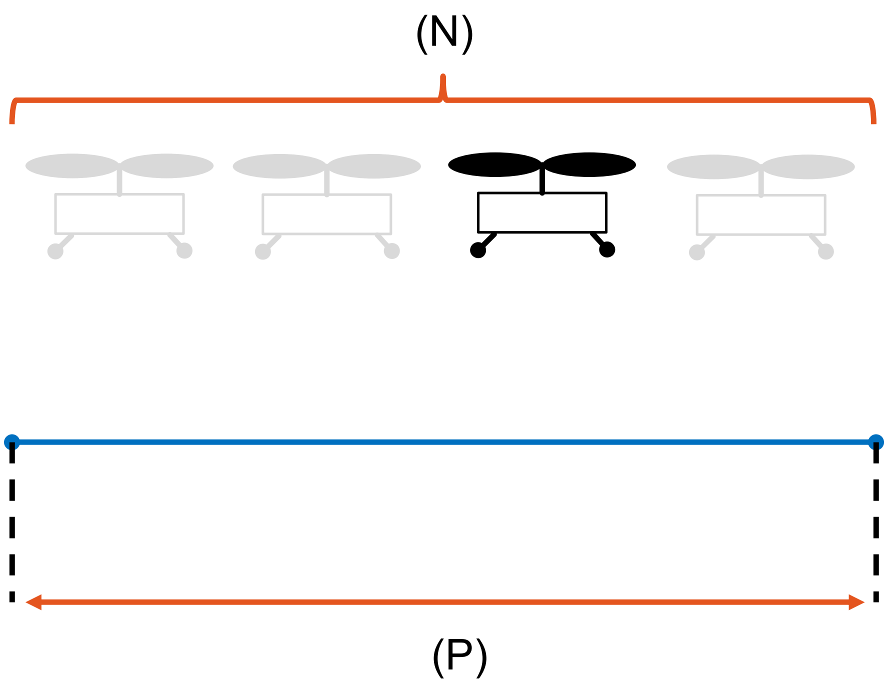
ACL2 MODEL OF DPSS-A

- Global Parameters
  - Number of UAVs: (N)
  - Size of perimeter: (P)
  - Segment Length: (P) / (N)
- UAV-p
  - UAV identifier
    - Natural Number [0 .. (N)-1]
  - Location on perimeter
  - Direction of motion
- UAV segment
  - Left Boundary
    - UAV.id * Segment Length
  - Right Boundary
    - Left Boundary + Segment Length



ACL2 MODEL OF DPSS-A

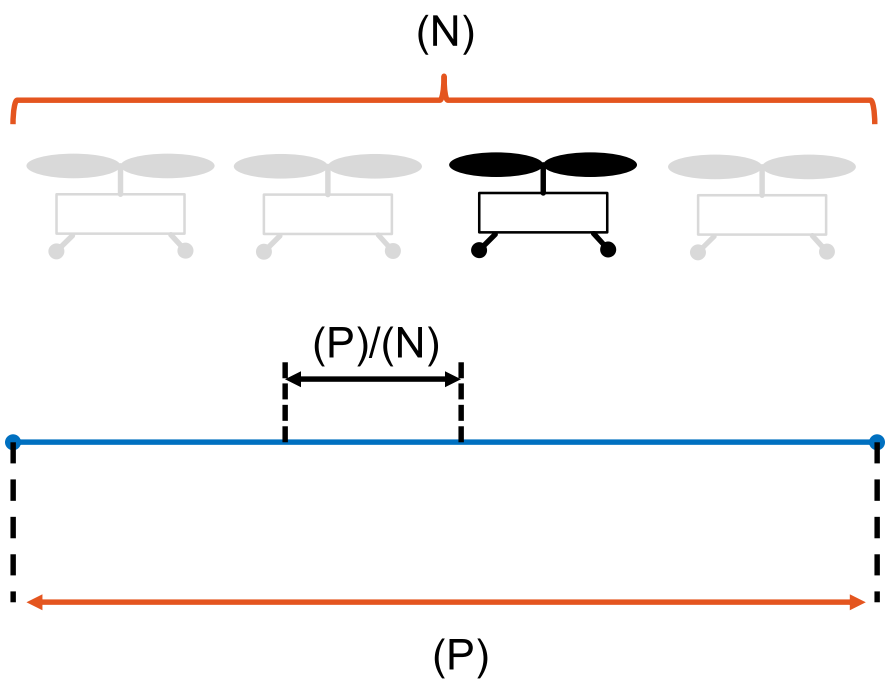
- Global Parameters
  - Number of UAVs: (N)
  - Size of perimeter: (P)
  - Segment Length: (P) / (N)
- UAV-p
  - UAV identifier
    - Natural Number [0 .. (N)-1]
  - Location on perimeter
  - Direction of motion
- UAV segment
  - Left Boundary
    - UAV.id * Segment Length
  - Right Boundary
    - Left Boundary + Segment Length



Collins Aerospace

ACL2 MODEL OF DPSS-A

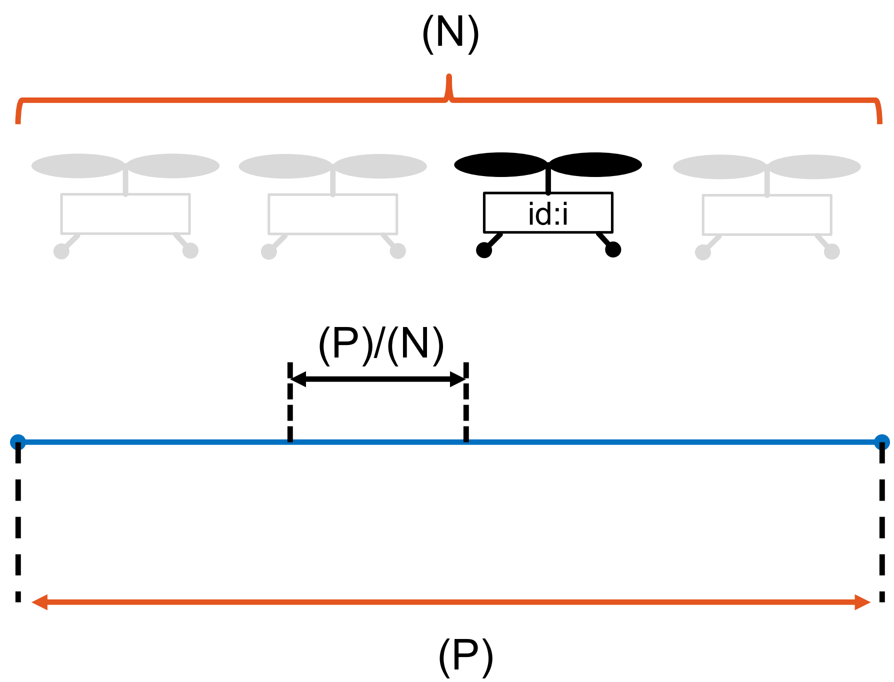
- Global Parameters
  - Number of UAVs: (N)
  - Size of perimeter: (P)
  - Segment Length: (P) / (N)
- UAV-p
  - UAV identifier
    - Natural Number [0 .. (N)-1]
  - Location on perimeter
  - Direction of motion
- UAV segment
  - Left Boundary
    - UAV.id * Segment Length
  - Right Boundary
    - Left Boundary + Segment Length



Collins Aerospace

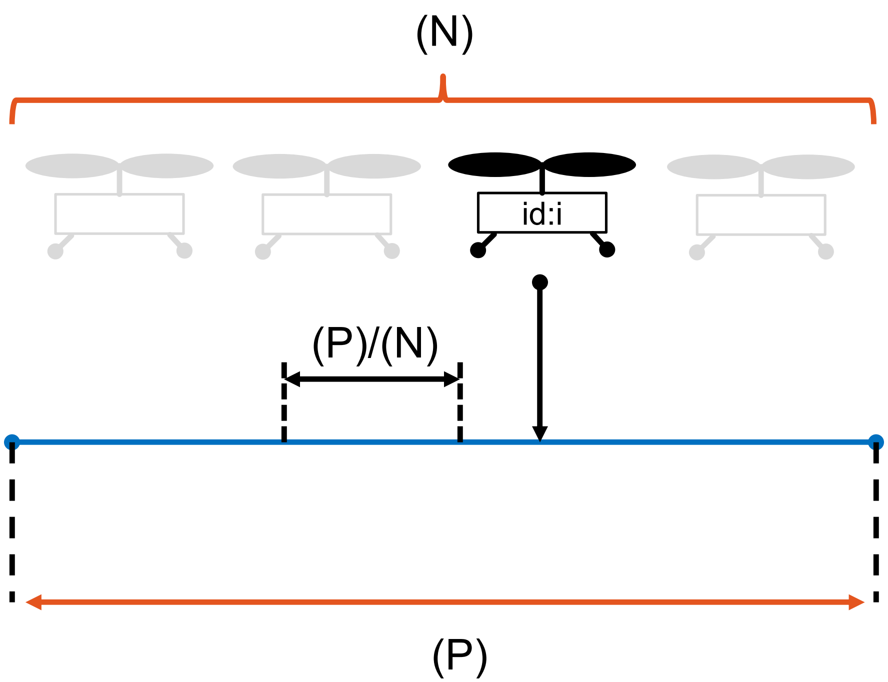
ACL2 MODEL OF DPSS-A

- Global Parameters
  - Number of UAVs: (N)
  - Size of perimeter: (P)
  - Segment Length: (P) / (N)
- UAV-p
  - UAV identifier
    - Natural Number [0 .. (N)-1]
  - Location on perimeter
  - Direction of motion
- UAV segment
  - Left Boundary
    - UAV.id * Segment Length
  - Right Boundary
    - Left Boundary + Segment Length



ACL2 MODEL OF DPSS-A

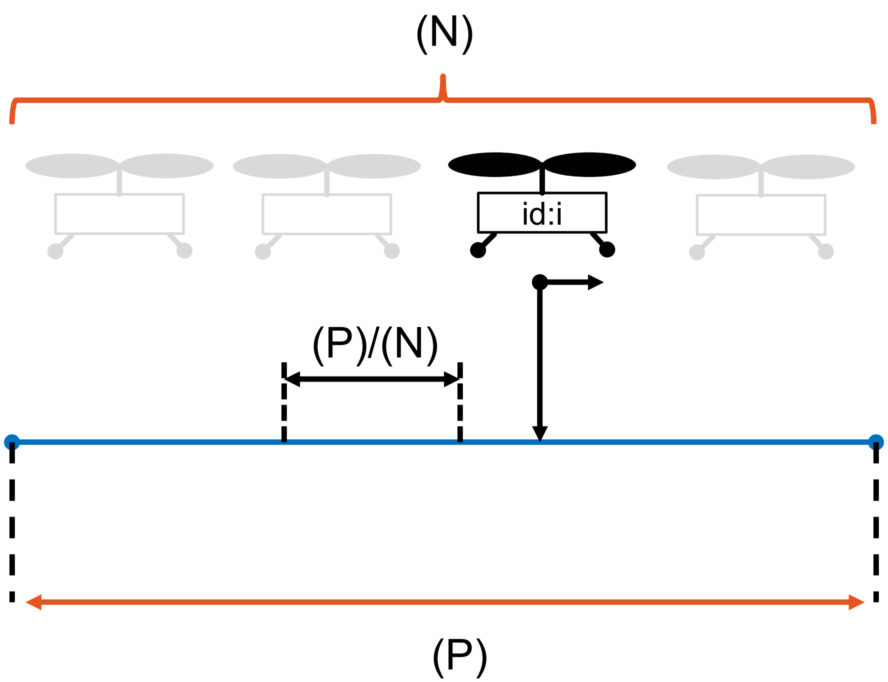
- Global Parameters
  - Number of UAVs: (N)
  - Size of perimeter: (P)
  - Segment Length: (P) / (N)
- UAV-p
  - UAV identifier
    - Natural Number [0 .. (N)-1]
  - Location on perimeter
  - Direction of motion
- UAV segment
  - Left Boundary
    - UAV.id * Segment Length
  - Right Boundary
    - Left Boundary + Segment Length



Collins Aerospace

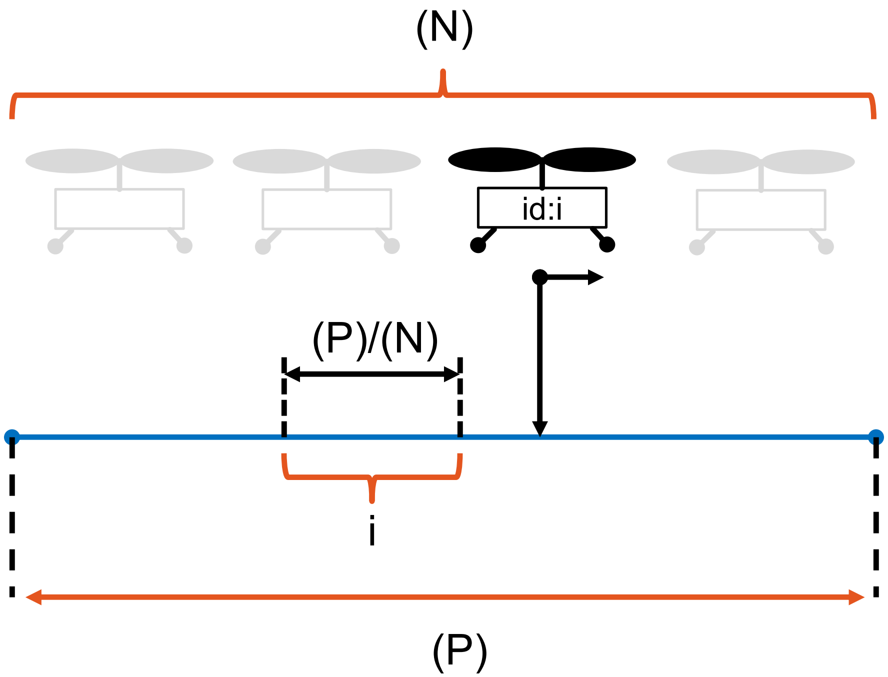
ACL2 MODEL OF DPSS-A

- Global Parameters
  - Number of UAVs: (N)
  - Size of perimeter: (P)
  - Segment Length: (P) / (N)
- UAV-p
  - UAV identifier
    - Natural Number [0 .. (N)-1]
  - Location on perimeter
  - Direction of motion
- UAV segment
  - Left Boundary
    - UAV.id * Segment Length
  - Right Boundary
    - Left Boundary + Segment Length



ACL2 MODEL OF DPSS-A

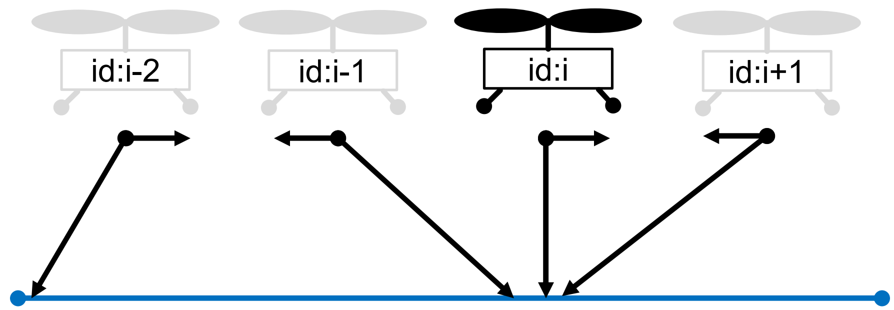
- Global Parameters
  - Number of UAVs: (N)
  - Size of perimeter: (P)
  - Segment Length: (P) / (N)
- UAV-p
  - UAV identifier
    - Natural Number [0 .. (N)-1]
  - Location on perimeter
  - Direction of motion
- UAV segment
  - Left Boundary
    - UAV.id * Segment Length
  - Right Boundary
    - Left Boundary + Segment Length



Collins Aerospace

WF-ENSEMBLE (TYPE)

- uav-list-p
  - a list of UAVs
- sequential-id-list-p
  - the UAV identifiers [0 .. (N)-1] are consistent with their position in the list
- Total of (N) UAVs
  - Length of uav-list is (N)
- ordered-location-list-p
  - Locations are non-decreasing
  - The location of a UAV with a lower ID is always left of (or equal) a UAV with a higher ID




```
(def::type-predicate wf-ensemble (ens)
  :type-name wf-ensemble
  :non-executable t
  :rewrite t
  :forward-chain-cases nil
  :body (and (uav-list-p ens)
             (sequential-id-list-p ens)
             (equal (len ens) (N))
             (ordered-location-list-p ens)
             ))
```

DPSS-A ALGORITHM

- event-for-uav
  - Recognizes conditions under which a UAV changes direction
- flip-on-events
  - Changes each UAV's direction under appropriate conditions (event-for-uav)
  - "Interesting" part of algorithm
- update-location-all
  - Change all UAV locations by some increment/time step
  - "continue in current direction" part of algorithm

```
if UAV i rendezvous with neighbor j then
  Travel with neighbor j to shared boarder position
  Set direction to monitor own segment
else if reached perimeter endpoint then
  Reverse direction
else
  Continue in current direction
end if
```

Algorithm A

DPSS-A ALGORITHM

- event-for-uav
  - Recognizes conditions under which a UAV changes direction
- flip-on-events
  - Changes each UAV's direction under appropriate conditions (event-for-uav)
  - "Interesting" part of algorithm
- update-location-all
  - Change all UAV locations by some increment/time step
  - "continue in current direction" part of algorithm

if UAV i rendezvous with neighbor j **then**
  Travel with neighbor j to shared boarder position
  Set direction to monitor own segment
else if reached perimeter endpoint **then**
  Reverse direction
else
  Continue in current direction
end if

Algorithm A

DPSS-A ALGORITHM

- event-for-uav
  - Recognizes conditions under which a UAV changes direction
- flip-on-events
  - Changes each UAV's direction under appropriate conditions (event-for-uav)
  - "Interesting" part of algorithm
- update-location-all
  - Change all UAV locations by some increment/time step
  - "continue in current direction" part of algorithm

```
if UAV i rendezvous with neighbor j then
  Travel with neighbor j to shared boarder position
  Set direction to monitor own segment
else if reached perimeter endpoint then
  Reverse direction
else
  Continue in current direction
end if
```

Algorithm A

Flipping takes Zero Time !

DPSS-A ALGORITHM

- event-for-uav
  - Recognizes conditions under which a UAV changes direction
- flip-on-events
  - Changes each UAV's direction under appropriate conditions (event-for-uav)
  - "Interesting" part of algorithm
- update-location-all
  - Change all UAV locations by some increment/time step
  - "continue in current direction" part of algorithm

if UAV i rendezvous with neighbor j **then**
  Travel with neighbor j to shared boarder position
  Set direction to monitor own segment
else if reached perimeter endpoint **then**
  Reverse direction
else
  Continue in current direction
end if

Algorithm A

Advances Time

Collins Aerospace

EVENT-FOR-UAV

- An **event** represents the conditions under which a UAV will change direction
  - Left-most UAV encounters left perimeter while moving left
  - Right-most UAV encounters right perimeter while moving right
  - UAV encounters left neighbor while moving left on or beyond the UAV's left segment boundary
  - UAV encounters right neighbor while moving right on or beyond the UAV's right segment boundary

```
(def::un event-for-uav (i ens)
  (declare (type t i ens))
  (let ((i   (uav-id-fix i)))
    (or
     ;; Left-most drone encounters left perimeter
     (and
      (= i 0)
      (< (UAV->direction (ith-uav i ens)) 0)
      (equal (UAV->location (ith-uav i ens))
             (left-perimeter-boundary)))
     ;; Right-most drone encounters right perimeter
     (and
      (= (1+ i) (N))
      (< 0 (UAV->direction (ith-uav i ens)))
      (equal (UAV->location (ith-uav i ens))
             (right-perimeter-boundary)))
     ;; Encounter left drone on or beyond left boudary
     (and
      (< 0 i)
      (< (UAV->direction (ith-uav i ens)) 0)
      (<= (UAV->location (ith-uav i ens))
          (UAV-left-boundary (ith-uav i ens)))
      (equal (UAV->location (ith-uav (1- i) ens))
             (UAV->location (ith-uav i ens)))
      )
     ;; Encounter right drone on or beyond right boundary
     (and
      (< (1+ i) (N))
      (< 0 (UAV->direction (ith-uav i ens)))
      (<= (UAV-right-boundary (ith-uav i ens))
          (UAV->location (ith-uav i ens)))
      (equal (UAV->location (ith-uav i ens))
             (UAV->location (ith-uav (1+ i) ens))))
     )))
```

Collins Aerospace

STEP-TIME: EVENT BASED SIMULATOR

```
(def::un next-step (dt ens)
  (declare (xargs :fty ((nnrat uav-list) rational uav-list)))
  (let ((ens (flip-on-events ens)))
    (let ((step (min dt (always-smallest-min-time-to-impending-impact ens))))
      (let ((ens (update-location-all step ens)))
        (mvlist (- dt step) ens)))))
```

```
(def::ung step-time (dt ens)
  (declare (xargs :signature ((t t) uav-list-p)
                  :verify-guards nil))
  (let ((dt  (nnrat-fix dt))
        (ens (uav-list-fix! ens)))
    (if (<= dt 0) ens
      (metlist ((dt ens) (next-step dt ens))
        (step-time dt ens)))))
```

STEP-TIME: EVENT BASED SIMULATOR

```
(def::un next-step (dt ens)
  (declare (xargs :fty ((nnrat uav-list) rational uav-list)))
  (let ((ens (flip-on-events ens)))
    (let ((step (min dt (always-smallest-min-time-to-impending-impact ens))))
      (let ((ens (update-location-all step ens)))
        (mvlist (- dt step) ens)))))
```

```
(def::ung step-time (dt ens)
  (declare (xargs :signature ((t t) uav-list-p)
                  :verify-guards nil))
  (let ((dt  (nnrat-fix dt))
        (ens (uav-list-fix! ens)))
    (if (<= dt 0) ens
      (metlist ((dt ens) (next-step dt ens))
        (step-time dt ens)))))
```

Given a requested time step and the current state of the system ..

Collins Aerospace

STEP-TIME: EVENT BASED SIMULATOR

```
(def::un next-step (dt ens)
  (declare (xargs :fty ((nnrat uav-list) rational uav-list)))
  (let ((ens (flip-on-events ens)))
    (let ((step (min dt (always-smallest-min-time-to-impending-impact ens))))
      (let ((ens (update-location-all step ens)))
        (mvlist (- dt step) ens)))))
```

Change the direction of any UAV experiencing an event

```
(def::ung step-time (dt ens)
  (declare (xargs :signature ((t t) uav-list-p)
                  :verify-guards nil))
  (let ((dt  (nnrat-fix dt))
        (ens (uav-list-fix! ens)))
    (if (<= dt 0) ens
      (metlist ((dt ens) (next-step dt ens))
        (step-time dt ens)))))
```

STEP-TIME: EVENT BASED SIMULATOR

```
(def::un next-step (dt ens)
  (declare (xargs :fty ((nnrat uav-list) rational uav-list)))
  (let ((ens (flip-on-events ens)))
    (let ((step (min dt (always-smallest-min-time-to-impending-impact ens))))
      (let ((ens (update-location-all step ens)))
        (mvlist (- dt step) ens)))))
```

```
(def::ung step-time (dt ens)
  (declare (xargs :signature ((t t) uav-list-p)
                  :verify-guards nil))
  (let ((dt  (nnrat-fix dt))
        (ens (uav-list-fix! ens)))
    (if (<= dt 0) ens
      (metlist ((dt ens) (next-step dt ens))
        (step-time dt ens)))))
```

Compute the minimum of the requested time step and the smallest time step to the next event

Collins Aerospace

STEP-TIME: EVENT BASED SIMULATOR

```
(def::un next-step (dt ens)
  (declare (xargs :fty ((nnrat uav-list) rational uav-list)))
  (let ((ens (flip-on-events ens)))
    (let ((step (min dt (always-smallest-min-time-to-impending-impact ens))))
      (let ((ens (update-location-all step ens)))
        (mvlist (- dt step) ens)))))
```

Advance the state of the system by the minimum time step.

```
(def::ung step-time (dt ens)
  (declare (xargs :signature ((t t) uav-list-p)
                  :verify-guards nil))
  (let ((dt  (nnrat-fix dt))
        (ens (uav-list-fix! ens)))
    (if (<= dt 0) ens
      (metlist ((dt ens) (next-step dt ens))
        (step-time dt ens)))))
```

STEP-TIME: EVENT BASED SIMULATOR

```
(def::un next-step (dt ens)
  (declare (xargs :fty ((nnrat uav-list) rational uav-list)))
  (let ((ens (flip-on-events ens)))
    (let ((step (min dt (always-smallest-min-time-to-impending-impact ens))))
      (let ((ens (update-location-all step ens)))
        (mvlist (- dt step) ens)))))
```

Return the remaining time and the new state of the system

```
(def::ung step-time (dt ens)
  (declare (xargs :signature ((t t) uav-list-p)
                  :verify-guards nil))
  (let ((dt  (nnrat-fix dt))
        (ens (uav-list-fix! ens)))
    (if (<= dt 0) ens
      (metlist ((dt ens) (next-step dt ens))
        (step-time dt ens)))))
```

Collins Aerospace

STEP-TIME: EVENT BASED SIMULATOR

```
(def::un next-step (dt ens)
  (declare (xargs :fty ((nnrat uav-list) rational uav-list)))
  (let ((ens (flip-on-events ens)))
    (let ((step (min dt (always-smallest-min-time-to-impending-impact ens))))
      (let ((ens (update-location-all step ens)))
        (mvlist (- dt step) ens)))))
```

```
(def::ung step-time (dt ens)
  (declare (xargs :signature ((t t) uav-list-p)
                  :verify-guards nil))
  (let ((dt  (nnrat-fix dt))
        (ens (uav-list-fix! ens)))
    (if (<= dt 0) ens
      (metlist ((dt ens) (next-step dt ens))
        (step-time dt ens)))))
```

Do this repeatedly until the system has advanced by the requested quantity of time

STEP-TIME TERMINATION?

- step-time
  - Time steps (rational) may be arbitrarily small
  - No guarantee we are making progress
- def::ung
  - Admit recursive functions without proof of termination
- Admits 3 interrelated functions
  - step-time
    - The function we want guarded by step-time-domain
  - step-time-domain
    - A predicate that guards against non-terminating inputs
  - step-time-measure
    - A function that decreases on every recursive call of step-time

```
(encapsulate
    ()

  (local
   (defthmd step-time-definition-alt
     (equal (step-time dt ens)
            (if (not (step-time-domain dt ens)) (uav-list-fix! ens)
              (let ((dt  (nnrat-fix dt))
                    (ens (uav-list-fix! ens)))
                (if (<= dt 0) ens
                  (metlist ((dt ens) (next-step dt ens))
                    (step-time dt ens))))))))

  (local
   (defthmd step-time-measure-alt
     (equal (step-time-measure dt ens)
            (if (not (step-time-domain dt ens)) 0
              (let ((dt  (nnrat-fix dt))
                    (ens (uav-list-fix! ens)))
                (if (<= dt 0) 0
                  (metlist ((dt ens) (next-step dt ens))
                    (+ 1 (step-time-measure dt ens)))))))))

  )
```

Collins Aerospace

ASSUME STEP-TIME ALWAYS TERMINATES

- We assume that the step-time function terminates on all inputs

```
(defun-sk step-time-always-terminates ()
  (forall (dt ens) (step-time-domain dt ens)))

(defthm step-time-always-terminates-implication
  (implies
   (step-time-always-terminates)
   (step-time-domain dt ens))
  :hints (("Goal" :use step-time-always-terminates-necc)))
```

OUTLINE

- Background
- DPSS-A Model
- **Overview of Convergence Proof**
- Specialized ACL2 Utilities
- Conclusion

Collins Aerospace

KEY AVD INVARIANTS

- Property of the current state
  - Given a state, return true if the property holds
- Cellular (local)
  - Expressed in terms of itself and its two neighbors
- Invariant
  - Once true, always true

"We say that [two drones] **have met** by time T if either they started together, moving in the same direction, or they have been involved in a meet or bounce event"

"We say a drone is **left synchronized** at time T if beyond that point it never goes to the left of its left endpoint."

HAVE MET (LEFT)

- For all but the rightmost UAV ..
- If the UAV is moving left ..
  - .. if it is right of its rightmost boundary ..
    - .. then the UAV to its right is escorting it back to their shared boundary
  - .. If it is in its segment but not on the left boundary ..
    - .. then the UAV is the same distance from its right boundary as the UAV to its right.
    - .. and If it is left of its right boundary ..
      - .. then the UAV is moving right.

> “We say that [two drones] **have met** by time T if either they started together, moving in the same direction, or they have been involved in a meet or bounce event”

```
(def::un have-met-Left-p (i ens)
  (declare (xargs :fty ((uav-id uav-list) bool)))
  (let ((uavi  (ith-uav i ens))
        (right (ith-uav (+ i 1) ens)))
    (implies
     (and
      (< i (+ -1 (N)))
      (< (UAV->direction uavi) 0))
     (and
      (implies
       (< (UAV-right-boundary uavi) (UAV->location uavi))
       (and
        (< (UAV->direction right) 0)
        (equal (UAV->location uavi) (UAV->location right))))
      (implies
       (and (<= (UAV->location uavi) (UAV-right-boundary uavi))
            (<  (UAV-left-boundary uavi) (UAV->location uavi)))
       (and
        (implies
         (< (UAV->location uavi) (UAV-right-boundary uavi))
         (< 0 (UAV->direction right)))
        (equal (average (UAV->location uavi) (UAV->location right))
               (UAV-right-boundary uavi))))
      ))))
```

Collins Aerospace

LEFT SYNCHRONIZED

- For all UAVs not the leftmost UAV ..
  - The average of its location with its left neighbor's location is not less than its left segment boundary
  - If it is moving left and is not co-incident with its left neighbor ..
    - .. then its left neighbor is moving right.

"We say a drone is **left synchronized** at time T if beyond that point it never goes to the left of its left endpoint."

```
(def::un LEFT-SYNCHRONIZED-p (j ens)
  (declare (xargs :fty ((uav-id uav-list) bool)))
  (implies
   ;;
   ;; The leftmost UAV is left synchronized.
   ;; For all UAV's greater than zero ..
   ;;
   (< 0 j)
   (and
    ;;
    ;; Their average loction with their left neighbor
    ;; is not less than their left boundary.
    ;;
    ;;             J
    ;; |--------|--------|
    ;;      x    ^    x
    (<= (UAV-left-boundary (ith-uav j ens))
        (average (UAV->location (ith-uav (+ -1 j) ens))
                 (UAV->location (ith-uav j ens))))
    (implies
     (and
      ;; And either they are moving right or they are
      ;; co-incident with their left neighbor ..
      (< (UAV->direction (ith-uav j ens)) 0)
      (not (equal (UAV->location (ith-uav j ens))
                  (UAV->location (ith-uav (+ -1 j) ens)))))
     ;;
     ;; or, if they are moving left ..
     ;; and they are not co-incident with their left neighbor ..
     ;; .. then the left neighbor is moving right.
     ;;
     ;;             J
     ;; |--------|--------|
     ;;      >    ^    <
     (< 0 (UAV->direction (ith-uav (+ -1 j) ens)))))))
```

KEY CONTRIBUTION

- Property of the current state
  - Given a state, return true if the property holds
- Cellular (local)
  - Expressed in terms of itself and its two neighbors
- Invariant
  - Once true, always true

Precise formalization of the central invariants w/to a concrete model of DPSS behavior

Eliminates reliance on fallible human intuition

Avoids a repeat of previous hand “proof” failure

Collins Aerospace

PROOF OUTLINE

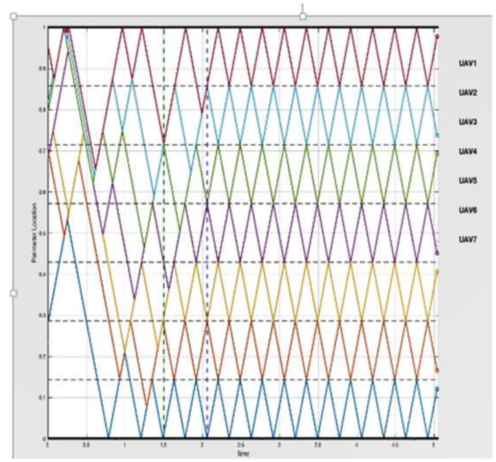
- Have Met (left) is invariant over step-time
- Have Met for a UAV is established following that UAV's first event
- Every UAV has had an event after T time increments
  - Worst case UAV separation
- Left Synchronized is invariant over step-time if Have Met (left)
- After all UAVs Have Met, all UAVs are left synchronized in T-1 time increments
  - Induction from left to right
- "Right" properties are symmetric, also established in 2T-1
- Full synchronization implies behavioral periodicity with period of 2 time increments

The bound improved from 2T to 2T-1 thanks to a more precise notion of convergence

FINAL CONVERGENCE THEOREM

```
(defun-sk dpss-location-convergence (ens)
  (forall (i)
    (equal (UAV->location (ith-uav i (step-time (* 2 (one)) ens)))
           (UAV->location (ith-uav i ens)))))


(defthm dpss-location-convergence-after-2T-1
  (implies
   (and
    (wf-ensemble ens)
    (step-time-always-terminates))
   (dpss-location-convergence (step-time (- (* 2 (TEE)) (ONE)) ens)))
  :hints (("Goal" :in-theory '(dpss-location-convergence-after-2T-1-helper))))
```



OUTLINE

- Background
- DPSS-A Model
- Overview of Convergence Proof
- **Specialized ACL2 Utilities**
- Conclusion

SPECIALIZED ACL2 UTILITIES

- def::ung
  - Admits partial recursive functions with induction schemes
- pattern::hint
  - Pattern matching for computed proof hints
- def::linear
  - Support for specialized linear rules

PATTERN::HINT

- ACL2 applies many kinds of lemmas automatically
  - Induction, rewrite, forward-chaining, linear
- Some lemmas are hard to express as useful rules
  - Quantified formulae
  - Have to be instantiated “by hand” using “hints”
- ACL2 computed hint facility
  - Allows hints to be computed
- pattern::hint
  - Computed hint facility based on pattern matching
  - Sophisticated, compositional pattern matching primitives
    - (:or (< x y) (not (< y x)))
  - Supports automated lemma Instantiation

```
;; ======================================================================
;;
;; This book provides a pattern driven computed hint facility.
;;
;; For example, given the following pattern hint:
;;
;; :hints ((pattern::hint
;;           (< x y)
;;           :use ((:instance helpful-lemma (a x) (b y))))
;;
;; the following subboal:
;;
;; (implies
;;   (and
;;     (< (foo x) 7)
;;     (<= x (foo x)))
;;   (< (foo a) (foo 7)))
;;
;; will result in a hint with one instance of helpful-lemma:
;;
;; :use ((:instance helpful-lemma (a (foo x)) (b 7)))
;;
;; With the following pattern hint:
;;
;; :hints ((pattern::hint
;;           (<= x y)
;;           :use ((:instance helpful-lemma (a x) (b y))))
;;
;; it will result in a hint with two instances of helpful-lemma:
;;
;; :use ((:instance helpful-lemma (a x)       (b (foo x)))
;;       (:instance helpful-lemma (a (foo 7)) (b (foo a))))
;;
;; ======================================================================
```

PATTERN::HINT CAPABILITIES

```
(defthm nonsense-example
  (< x y)
  :hints ((pattern::hint defined-hint)
          (pattern::hint
           (:and
            (not (consp x))
            (:or (integerp x) (stringp x))
            (:first (acl2-numberp y) (integerp y))
            (:either (integerp a))
            (:term (+ p q))
            (:match a (append x y))
            (:literally w x)
            (:commutes (equal x y) (y x))
            (:replicate (equal x y) (a x) (b y))
            (:call (lte-match) (c d))
            (:syntaxp (not (cw "Found Match ~x0 ~x1" a b)))
            (:bind-free (expand-term a) (x y))
            (:not (foo y))
            (:if (:equal x y) (:bind (n x) (p y))
                 (:implies (< a b) (integerp z)))
            (:check (< a b))
            (:keep x y)
            )
           :use       (silly-rule (:instance silly-rule
                                             (x x)
                                             (y y)))
           :expand    ((beta x))
           :cases     ((gamma x y))
           :restrict  ((dangerous-rule ((v1 a) (v2 b))))
           :in-theory (e/d (joe) (fred))
           :do-not    '(preprocess)
           :limit     nil
           :repeat    nil
           :slow      nil
           :stable    t
           :name      frank
          )))
```

Patterns (the `:and` block)

Actions (Computed Hints) (`:use` through `:do-not`)

Options (`:limit` through `:name`)

```
:limit     [nil]   ;; When numeric, limit the total number of times the hint can fire
:repeat    [nil]   ;; When true, allow duplicate hint instances in successive subgoals.
:slow      [nil]   ;; When numeric, limits the number of hint instances per application.
                   ;; When true, limits the number of instances to 1
:stable    [t]     ;; Apply hints only when stable under simplification.
:name      frank   ;; The name reported when the hint fires.
```

Collins Aerospace

PATTERN::HINT CAPABILITIES

```
(defthm nonsense-example
  (< x y)
  :hints ((pattern::hint defined-hint)
          (pattern::hint
           (:and
            (not (consp x))
            (:or (integerp x) (stringp x))
            (:first (acl2-numberp y) (integerp y))
            (:either (integerp a))
            (:term (+ p q))
            (:match a (append x y))
            (:literally w x)
            (:commutes (equal x y) (y x))
            (:replicate (equal x y) (a x) (b y))
            (:call (lte-match) (c d))
            (:syntaxp (not (cw "Found Match ~x0 ~x1" a b)))
            (:bind-free (expand-term a) (x y))
            (:not (foo y))
            (:if (:equal x y) (:bind (n x) (p y))
                 (:implies (< a b) (integerp z)))
            (:check (< a b))
            (:keep x y)
            )
           :use       (silly-rule (:instance silly-rule
                                   (x x)
                                   (y y)))
           :expand    ((beta x))
           :cases     ((gamma x y))
           :restrict  ((dangerous-rule ((v1 a) (v2 b))))
           :in-theory (e/d (joe) (fred))
           :do-not   '(preprocess)
           :limit    nil
           :repeat   nil
           :slow     nil
           :stable   t
           :name     frank
          )))
```

Collins Aerospace

PATTERN::HINT CAPABILITIES

```
(defthm nonsense-example
  (< x y)
  :hints ((pattern::hint defined-hint)
          (pattern::hint
           (:and
            (not (consp x))
            (:or (integerp x) (stringp x))
            (:first (acl2-numberp y) (integerp y))
            (:either (integerp a))
            (:term (+ p q))
            (:match a (append x y))
            (:literally w x)
            (:commutes (equal x y) (y x))
            (:replicate (equal x y) (a x) (b y))
            (:call (lte-match) (c d))
            (:syntaxp (not (cw "Found Match ~x0 ~x1" a b)))
            (:bind-free (expand-term a) (x y))
            (:not (foo y))
            (:if (:equal x y) (:bind (n x) (p y))
                 (:implies (< a b) (integerp z)))
            (:check (< a b))
            (:keep x y)
            )
           :use       (silly-rule (:instance silly-rule
                                             (x x)
                                             (y y)))
           :expand    ((beta x))
           :cases     ((gamma x y))
           :restrict  ((dangerous-rule ((v1 a) (v2 b))))
           :in-theory (e/d (joe) (fred))
           :do-not    '(preprocess)
           :limit    nil
           :repeat   nil
           :slow     nil
           :stable   t
           :name     frank
          )))
```

Collins Aerospace

PATTERN::HINT CAPABILITIES

```
(defthm nonsense-example
  (< x y)
  :hints ((pattern::hint defined-hint)
          (pattern::hint
           (:and
            (not (consp x))
            (:or (integerp x) (stringp x))
            (:first (acl2-numberp y) (integerp y))
            (:either (integerp a))
            (:term (+ p q))
            (:match a (append x y))
            (:literally w x)
            (:commutes (equal x y) (y x))
            (:replicate (equal x y) (a x) (b y))
            (:call (lte-match) (c d))
            (:syntaxp (not (cw "Found Match ~x0 ~x1" a b)))
            (:bind-free (expand-term a) (x y))
            (:not (foo y))
            (:if (:equal x y) (:bind (n x) (p y))
                 (:implies (< a b) (integerp z)))
            (:check (< a b))
            (:keep x y)
            )
           :use       (silly-rule (:instance silly-rule
                                   (x x)
                                   (y y)))
           :expand    ((beta x))
           :cases     ((gamma x y))
           :restrict  ((dangerous-rule ((v1 a) (v2 b))))
           :in-theory (e/d (joe) (fred))
           :do-not   '(preprocess)
           :limit    nil
           :repeat   nil
           :slow     nil
           :stable   t
           :name     frank
          )))
```

Collins Aerospace

PATTERN::HINT CAPABILITIES

```
(defthm nonsense-example
  (< x y)
  :hints ((pattern::hint defined-hint)
          (pattern::hint
           (:and
            (not (consp x))
            (:or (integerp x) (stringp x))
            (:first (acl2-numberp y) (integerp y))
            (:either (integerp a))
            (:term (+ p q))
            (:match a (append x y))
            (:literally w x)
            (:commutes (equal x y) (y x))
            (:replicate (equal x y) (a x) (b y))
            (:call (lte-match) (c d))
            (:syntaxp (not (cw "Found Match ~x0 ~x1" a b)))
            (:bind-free (expand-term a) (x y))
            (:not (foo y))
            (:if (:equal x y) (:bind (n x) (p y))
                 (:implies (< a b) (integerp z)))
            (:check (< a b))
            (:keep x y)
            )
           :use       (silly-rule (:instance silly-rule
                                   (x x)
                                   (y y)))
           :expand    ((beta x))
           :cases     ((gamma x y))
           :restrict  ((dangerous-rule ((v1 a) (v2 b))))
           :in-theory (e/d (joe) (fred))
           :do-not   '(preprocess)
           :limit    nil
           :repeat   nil
           :slow     nil
           :stable   t
           :name     frank
          )))
```

Collins Aerospace

PATTERN::HINT CAPABILITIES

```
(defthm nonsense-example
  (< x y)
  :hints ((pattern::hint defined-hint)
          (pattern::hint
           (:and
            (not (consp x))
            (:or (integerp x) (stringp x))
            (:first (acl2-numberp y) (integerp y))
            (:either (integerp a))
            (:term (+ p q))
            (:match a (append x y))
            (:literally w x)
            (:commutes (equal x y) (y x))
            (:replicate (equal x y) (a x) (b y))
            (:call (lte-match) (c d))
            (:syntaxp (not (cw "Found Match ~x0 ~x1" a b)))
            (:bind-free (expand-term a) (x y))
            (:not (foo y))
            (:if (:equal x y) (:bind (n x) (p y))
                 (:implies (< a b) (integerp z)))
            (:check (< a b))
            (:keep x y)
            )
           :use       (silly-rule (:instance silly-rule
                                    (x x)
                                    (y y)))
           :expand    ((beta x))
           :cases     ((gamma x y))
           :restrict  ((dangerous-rule ((v1 a) (v2 b))))
           :in-theory (e/d (joe) (fred))
           :do-not   '(preprocess)
           :limit    nil
           :repeat   nil
           :slow     nil
           :stable   t
           :name     frank
          )))
```

Collins Aerospace

PATTERN::HINT CAPABILITIES

```
(defthm nonsense-example
  (< x y)
  :hints ((pattern::hint defined-hint)
          (pattern::hint
           (:and
            (not (consp x))
            (:or (integerp x) (stringp x))
            (:first (acl2-numberp y) (integerp y))
            (:either (integerp a))
            (:term (+ p q))
            (:match a (append x y))
            (:literally w x)
            (:commutes (equal x y) (y x))
            (:replicate (equal x y) (a x) (b y))
            (:call (lte-match) (c d))
            (:syntaxp (not (cw "Found Match ~x0 ~x1" a b)))
            (:bind-free (expand-term a) (x y))
            (:not (foo y))
            (:if (:equal x y) (:bind (n x) (p y))
                 (:implies (< a b) (integerp z)))
            (:check (< a b))
            (:keep x y)
            )
           :use       (silly-rule (:instance silly-rule
                                             (x x)
                                             (y y)))
           :expand    ((beta x))
           :cases     ((gamma x y))
           :restrict  ((dangerous-rule ((v1 a) (v2 b))))
           :in-theory (e/d (joe) (fred))
           :do-not   '(preprocess)
           :limit    nil
           :repeat   nil
           :slow     nil
           :stable   t
           :name     frank
          )))
```

PATTERN::HINT CAPABILITIES

```
(defthm nonsense-example
  (< x y)
  :hints ((pattern::hint defined-hint)
          (pattern::hint
           (:and
            (not (consp x))
            (:or (integerp x) (stringp x))
            (:first (acl2-numberp y) (integerp y))
            (:either (integerp a))
            (:term (+ p q))
            (:match a (append x y))
            (:literally w x)
            (:commutes (equal x y) (y x))
            (:replicate (equal x y) (a x) (b y))
            (:call (lte-match) (c d))
            (:syntaxp (not (cw "Found Match ~x0 ~x1" a b)))
            (:bind-free (expand-term a) (x y))
            (:not (foo y))
            (:if (:equal x y) (:bind (n x) (p y))
                 (:implies (< a b) (integerp z)))
            (:check (< a b))
            (:keep x y)
            )
           :use       (silly-rule (:instance silly-rule
                                   (x x)
                                   (y y)))
           :expand    ((beta x))
           :cases     ((gamma x y))
           :restrict  ((dangerous-rule ((v1 a) (v2 b))))
           :in-theory (e/d (joe) (fred))
           :do-not   '(preprocess)
           :limit    nil
           :repeat   nil
           :slow     nil
           :stable   t
           :name     frank
          )))
```

Collins Aerospace

PATTERN::HINT CAPABILITIES

```
(defthm nonsense-example
  (< x y)
  :hints ((pattern::hint defined-hint)
          (pattern::hint
           (:and
            (not (consp x))
            (:or (integerp x) (stringp x))
            (:first (acl2-numberp y) (integerp y))
            (:either (integerp a))
            (:term (+ p q))
            (:match a (append x y))
            (:literally w x)
            (:commutes (equal x y) (y x))
            (:replicate (equal x y) (a x) (b y))
            (:call (lte-match) (c d))
            (:syntaxp (not (cw "Found Match ~x0 ~x1" a b)))
            (:bind-free (expand-term a) (x y))
            (:not (foo y))
            (:if (:equal x y) (:bind (n x) (p y))
                 (:implies (< a b) (integerp z)))
            (:check (< a b))
            (:keep x y)
            )
           :use       (silly-rule (:instance silly-rule
                                    (x x)
                                    (y y)))
           :expand    ((beta x))
           :cases     ((gamma x y))
           :restrict  ((dangerous-rule ((v1 a) (v2 b))))
           :in-theory (e/d (joe) (fred))
           :do-not   '(preprocess)
           :limit    nil
           :repeat   nil
           :slow     nil
           :stable   t
           :name     frank
          )))
```

→

```
(def::pattern-function lte-match ()
  (:or (< a b)
       (not (< b a)))
  :returns (a b))
```

PATTERN::HINT CAPABILITIES

```
(defthm nonsense-example
  (< x y)
  :hints ((pattern::hint defined-hint)
          (pattern::hint
           (:and
            (not (consp x))
            (:or (integerp x) (stringp x))
            (:first (acl2-numberp y) (integerp y))
            (:either (integerp a))
            (:term (+ p q))
            (:match a (append x y))
            (:literally w x)
            (:commutes (equal x y) (y x))
            (:replicate (equal x y) (a x) (b y))
            (:call (lte-match) (c d))
            (:syntaxp (not (cw "Found Match ~x0 ~x1" a b)))
            (:bind-free (expand-term a) (x y))
            (:not (foo y))
            (:if (:equal x y) (:bind (n x) (p y))
                 (:implies (< a b) (integerp z)))
            (:check (< a b))
            (:keep x y)
            )
           :use       (silly-rule (:instance silly-rule
                                   (x x)
                                   (y y)))
           :expand    ((beta x))
           :cases     ((gamma x y))
           :restrict  ((dangerous-rule ((v1 a) (v2 b))))
           :in-theory (e/d (joe) (fred))
           :do-not    '(preprocess)
           :limit     nil
           :repeat    nil
           :slow      nil
           :stable    t
           :name      frank
          )))
```

PATTERN::HINT CAPABILITIES

```
(defthm nonsense-example
  (< x y)
  :hints ((pattern::hint defined-hint)
          (pattern::hint
           (:and
            (not (consp x))
            (:or (integerp x) (stringp x))
            (:first (acl2-numberp y) (integerp y))
            (:either (integerp a))
            (:term (+ p q))
            (:match a (append x y))
            (:literally w x)
            (:commutes (equal x y) (y x))
            (:replicate (equal x y) (a x) (b y))
            (:call (lte-match) (c d))
            (:syntaxp (not (cw "Found Match ~x0 ~x1" a b)))
            (:bind-free (expand-term a) (x y))
            (:not (foo y))
            (:if (:equal x y) (:bind (n x) (p y))
                 (:implies (< a b) (integerp z)))
            (:check (< a b))
            (:keep x y)
            )
           :use       (silly-rule (:instance silly-rule
                                   (x x)
                                   (y y)))
           :expand    ((beta x))
           :cases     ((gamma x y))
           :restrict  ((dangerous-rule ((v1 a) (v2 b))))
           :in-theory (e/d (joe) (fred))
           :do-not   '(preprocess)
           :limit    nil
           :repeat   nil
           :slow     nil
           :stable   t
           :name     frank
          )))
```

Collins Aerospace

PATTERN::HINT CAPABILITIES

```
(defthm nonsense-example
  (< x y)
  :hints ((pattern::hint defined-hint)
          (pattern::hint
           (:and
            (not (consp x))
            (:or (integerp x) (stringp x))
            (:first (acl2-numberp y) (integerp y))
            (:either (integerp a))
            (:term (+ p q))
            (:match a (append x y))
            (:literally w x)
            (:commutes (equal x y) (y x))
            (:replicate (equal x y) (a x) (b y))
            (:call (lte-match) (c d))
            (:syntaxp (not (cw "Found Match ~x0 ~x1" a b)))
            (:bind-free (expand-term a) (x y))
            (:not (foo y))
            (:if (:equal x y) (:bind (n x) (p y))
                 (:implies (< a b) (integerp z)))
            (:check (< a b))
            (:keep x y)
            )
           :use      (silly-rule (:instance silly-rule
                                            (x x)
                                            (y y)))
           :expand    ((beta x))
           :cases     ((gamma x y))
           :restrict  ((dangerous-rule ((v1 a) (v2 b))))
           :in-theory (e/d (joe) (fred))
           :do-not   '(preprocess)
           :limit    nil
           :repeat   nil
           :slow     nil
           :stable   t
           :name     frank
          )))
```

Collins Aerospace

PATTERN::HINT CAPABILITIES

```
(defthm nonsense-example
  (< x y)
  :hints ((pattern::hint defined-hint)
          (pattern::hint
           (:and
            (not (consp x))
            (:or (integerp x) (stringp x))
            (:first (acl2-numberp y) (integerp y))
            (:either (integerp a))
            (:term (+ p q))
            (:match a (append x y))
            (:literally w x)
            (:commutes (equal x y) (y x))
            (:replicate (equal x y) (a x) (b y))
            (:call (lte-match) (c d))
            (:syntaxp (not (cw "Found Match ~x0 ~x1" a b)))
            (:bind-free (expand-term a) (x y))
            (:not (foo y))
            (:if (:equal x y) (:bind (n x) (p y))
                 (:implies (< a b) (integerp z)))
            (:check (< a b))
            (:keep x y)
            )
           :use       (silly-rule (:instance silly-rule
                                  (x x)
                                  (y y)))
           :expand    ((beta x))
           :cases     ((gamma x y))
           :restrict  ((dangerous-rule ((v1 a) (v2 b))))
           :in-theory (e/d (joe) (fred))
           :do-not   '(preprocess)
           :limit    nil
           :repeat   nil
           :slow     nil
           :stable   t
           :name     frank
          )))
```

PATTERN::HINT CAPABILITIES

```
(defthm nonsense-example
  (< x y)
  :hints ((pattern::hint defined-hint)
          (pattern::hint
           (:and
            (not (consp x))
            (:or (integerp x) (stringp x))
            (:first (acl2-numberp y) (integerp y))
            (:either (integerp a))
            (:term (+ p q))
            (:match a (append x y))
            (:literally w x)
            (:commutes (equal x y) (y x))
            (:replicate (equal x y) (a x) (b y))
            (:call (lte-match) (c d))
            (:syntaxp (not (cw "Found Match ~x0 ~x1" a b)))
            (:bind-free (expand-term a) (x y))
            (:not (foo y))
            (:if (:equal x y) (:bind (n x) (p y))
                 (:implies (< a b) (integerp z)))
            (:check (< a b))
            (:keep x y)
            )
           :use       (silly-rule (:instance silly-rule
                                              (x x)
                                              (y y)))
           :expand    ((beta x))
           :cases     ((gamma x y))
           :restrict  ((dangerous-rule ((v1 a) (v2 b))))
           :in-theory (e/d (joe) (fred))
           :do-not   '(preprocess)
           :limit    nil
           :repeat   nil
           :slow     nil
           :stable   t
           :name     frank
          )))
```

Actions (Computed Hints)

PATTERN::HINT CAPABILITIES

```
(defthm nonsense-example
  (< x y)
  :hints ((pattern::hint defined-hint)
          (pattern::hint
           (:and
            (not (consp x))
            (:or (integerp x) (stringp x))
            (:first (acl2-numberp y) (integerp y))
            (:either (integerp a))
            (:term (+ p q))
            (:match a (append x y))
            (:literally w x)
            (:commutes (equal x y) (y x))
            (:replicate (equal x y) (a x) (b y))
            (:call (lte-match) (c d))
            (:syntaxp (not (cw "Found Match ~x0 ~x1" a b)))
            (:bind-free (expand-term a) (x y))
            (:not (foo y))
            (:if (:equal x y) (:bind (n x) (p y))
                 (:implies (< a b) (integerp z)))
            (:check (< a b))
            (:keep x y)
            )
           :use       (silly-rule (:instance silly-rule
                                             (x x)
                                             (y y)))
           :expand    ((beta x))
           :cases     ((gamma x y))
           :restrict  ((dangerous-rule ((v1 a) (v2 b))))
           :in-theory (e/d (joe) (fred))
           :do-not   '(preprocess)
           :limit    nil
           :repeat   nil
           :slow     nil
           :stable   t
           :name     frank
          )))
```

```
(def::pattern-hint defined-hint
  (:or (<= (nfix x) (goo i y))
       (< (nfix x) (goo i y)))
  :slow t
  :expand ((goo i y)))
```

PATTERN::HINT CAPABILITIES

```
(defthm nonsense-example
  (< x y)
  :hints ((pattern::hint defined-hint)
          (pattern::hint
           (:and
            (not (consp x))
            (:or (integerp x) (stringp x))
            (:first (acl2-numberp y) (integerp y))
            (:either (integerp a))
            (:term (+ p q))
            (:match a (append x y))
            (:literally w x)
            (:commutes (equal x y) (y x))
            (:replicate (equal x y) (a x) (b y))
            (:call (lte-match) (c d))
            (:syntaxp (not (cw "Found Match ~x0 ~x1" a b)))
            (:bind-free (expand-term a) (x y))
            (:not (foo y))
            (:if (:equal x y) (:bind (n x) (p y))
                 (:implies (< a b) (integerp z)))
            (:check (< a b))
            (:keep x y)
            )
           :use       (silly-rule (:instance silly-rule
                                             (x x)
                                             (y y)))
           :expand    ((beta x))
           :cases     ((gamma x y))
           :restrict  ((dangerous-rule ((v1 a) (v2 b))))
           :in-theory (e/d (joe) (fred))
           :do-not   '(preprocess)
           :limit    nil
           :repeat   nil
           :slow     nil
           :stable   t
           :name     frank
          )))
```

Pattern hints fired **986** times over the course of the entire proof and generated **1353** lemma instances (not all pattern hints instantiate lemmas and some may instantiate more than one lemma)

DEF::LINEAR

- ACL2 provides automated support for applying linear facts
  - Only rules of a particular form can be (good) linear rules

```
(implies
 (pred x)
 (< (foo x) x))
```

- Sadly, one of our **key** properties isn't a good ACL2 linear rule
  - The location of a UAV with a lower ID is always left of (or equal) a UAV with a higher ID

```
(implies
 (< x y)
 (< (foo x) (foo y)))
```

- def::linear
  - Forces ACL2 to apply the rule "the way it should"

```
(def::linear location-linear
  (implies
   (and
    (syntaxp (not (equal i j)))
    (<= (uav-id-fix i) (uav-id-fix j))
    (wf-ensemble ens))
   (<= (uav->location (ith-uav i ens))
       (uav->location (ith-uav j ens)))))
```

Collins Aerospace

DEF::LINEAR

- ACL2 provides automated support for applying linear facts
  - Only rules of a particular form can be (good) linear rules

```
(implies
 (pred x)
 (< (foo x) x))
```

```
(implies
 (< x y)
 (< (foo x) (foo y)))
```

- Sadly, one of our **key** properties isn't a good ACL2 linear rule
  - The location of a UAV with a lower ID is always left of (or equal) a UAV with a higher ID
- def::linear
  - Forces ACL2 to apply the rule "the way it should"

```
(def::linear location-linear
  (implies
   (and
    (syntaxp (not (equal i j)))
    (<= (uav-id-fix i) (uav-id-fix j))
    (wf-ensemble ens))
   (<= (uav->location (ith-uav i ens))
       (uav->location (ith-uav j ens)))))
```

DEF::LINEAR

- ACL2 provides automated support for applying linear facts
  - Only rules of a particular form can be (good) linear rules

```
(implies
 (pred x)
 (< (foo x) x))
```

```
(implies
 (< x y)
 (< (foo x) (foo y)))
```

- Sadly, one of our **key** properties isn't a good ACL2 linear rule
  - The location of a UAV with a lower ID is always left of (or equal) a UAV with a higher ID

- def::linear
  - Forces ACL2 to apply the rule "the way it should"

```
(def::linear location-linear
  (implies
   (and
    (syntaxp (not (equal i j)))
    (<= (uav-id-fix i) (uav-id-fix j))
    (wf-ensemble ens))
   (<= (uav->location (ith-uav i ens))
       (uav->location (ith-uav j ens)))))
```

Collins Aerospace

DEF::LINEAR GUTS

```
(def::linear location-linear
  (implies
   (and
    (syntaxp (not (equal i j)))
    (<= (uav-id-fix i) (uav-id-fix j))
    (wf-ensemble ens))
   (<= (uav->location (ith-uav i ens))
       (uav->location (ith-uav j ens)))))
```

```
(ENCAPSULATE
 NIL
 (DEFTHM
  LOCATION-LINEAR
  (IMPLIES (AND (SYNTAXP (NOT (EQUAL I J)))
                (<= (UAV-ID-FIX I) (UAV-ID-FIX J))
                (WF-ENSEMBLE ENS))
           (NOT (< (UAV->LOCATION (ITH-UAV J ENS))
                   (UAV->LOCATION (ITH-UAV I ENS)))))
  :RULE-CLASSES
  ((:LINEAR
    :TRIGGER-TERMS ((UAV->LOCATION (ITH-UAV I ENS)))
    :COROLLARY
    (IMPLIES (AND (LINEAR::LINEAR-BINDER LOCATION-LINEAR
                                         ((UAV->LOCATION (ITH-UAV J ENS)))
                                         (J))
                  (SYNTAXP (NOT (EQUAL I J)))
                  (<= (UAV-ID-FIX I) (UAV-ID-FIX J))
                  (WF-ENSEMBLE ENS))
             (NOT (< (UAV->LOCATION (ITH-UAV J ENS))
                     (UAV->LOCATION (ITH-UAV I ENS))))))
   (:LINEAR
    :TRIGGER-TERMS ((UAV->LOCATION (ITH-UAV J ENS)))
    :COROLLARY
    (IMPLIES (AND (LINEAR::LINEAR-BINDER LOCATION-LINEAR
                                         ((UAV->LOCATION (ITH-UAV I ENS)))
                                         (I))
                  (SYNTAXP (NOT (EQUAL I J)))
                  (<= (UAV-ID-FIX I) (UAV-ID-FIX J))
                  (WF-ENSEMBLE ENS))
             (NOT (< (UAV->LOCATION (ITH-UAV J ENS))
                     (UAV->LOCATION (ITH-UAV I ENS))))))))))
```

Initial trigger

Bind free; pattern match linear pot

Collins Aerospace

DEF::LINEAR PERFORMANCE

- Developed late in proof
  - pattern::hint used to instantiate location-linear
- More expensive than pattern::hint
  - pattern::hint fires late, only when needed
  - def::linear fires throughout
- More Automated
- Final proof
  - Mixture of def::linear and pattern::hint

```
(def::linear location-linear
  (implies
   (and
    (syntaxp (not (equal i j)))
    (<= (uav-id-fix i) (uav-id-fix j))
    (wf-ensemble ens))
   (<= (uav->location (ith-uav i ens))
       (uav->location (ith-uav j ens)))))
```

OUTLINE

- Background
- DPSS-A Model
- Overview of Convergence Proof
- Specialized ACL2 Utilities
- **Conclusion**

CONCLUSION

- Mechanized proof of DPSS-A Convergence in ACL2
    - Based on Hand proof by Avigad/van Doorn
- Key Contribution
    - Precise formalization of central invariants w/to a concrete model of DPSS behavior
- Useful ACL2 Artifacts
    - pattern::hint
    - def::linear
- Future Work
    - Proof of DPSS-B convergence bound