Formalising the Comhordú Coordination Model

Colm Bhandal, Mélanie Bouroche, Arthur Hughes

FMIS, Limerick

June 19, 2011
Coordination Model

- Multi-agent systems.
- Provides a means of reasoning about systems of agents.
- Provides a strategy for coordination among agents.
- May be abstract- cover a class of systems.
Contribution of this Work

- This work is a formalisation of the coordination model Comhordú.
- Clarifies - provides an exactness to the description of Comhordú.
- Reveals - oversights in the original model.
- Extends - introduces new and yet necessary concepts for the working of Comhordú.
Comhordú in Context - What and Why

- Comhordú is a coordination model for the class of mobile agents communicating over a wireless network.
- It was developed to tackle the problem of coordination in an unreliable setting.
- Departs from traditional consensus based approaches.
What makes up a Comhordú system?

- **Entities**: There will be a finite collection of entities in the system: robots, cars, unmanned aerial vehicles etc. May have multiple types.

- **Mobility**: All entities will be capable of movement in space.

- **Modes**: Abstraction of what an entity is doing e.g. stopped/moving.

- **Safety**: A property that the system should never violate.
Key Assumptions

- **Fail Safe Modes:** No violation of safety if all the entities are in these modes.

- **Locality:** Violations to safety can only occur between entities that are within a certain range.
The Space Elastic Model

- Underlying physical layer model developed in [3].
- Provides entities with updates on the state of coverage within a time bound *adaptNotif*.
- Entities can use coverage information to deduce whether or not it’s safe to act.
- Implementation is not the concern of this work, which is at a higher layer.
What is the Comhordú Protocol

- A strategy for entities within a Comhordú system to coordinate.
- Hypothesis in previous work [2, 1] is that protocol guarantees safety.
The Comhordú Protocol - How it Works

- Entity wishing to act in some mode begins to periodically broadcast messages.
- Once enough time has elapsed, sender begins acting unless it is disrupted.
- Disruption can be a message from another close by entity that is possibly incompatible as per safety.
- Disruption can also be a degradation in wireless coverage.
- In either case, a disruption warrants a reaction - perhaps “shut down” to a fail safe mode.
Let’s consider some formalisms for modelling Comhordú.

A suitable formalism should include broadcast & time.
Formalisms considered

- Network two-tiered process calculi: too low level, model of time not great.
- UPPAAL model checker: broadcast & time but no infinite behaviour. Need to instantiate the model for a certain scope.
- TCSP: Time and synchronisation between multiple agents like broadcast but no distinction between sender/receiver- problem.
Formalism chosen: TCBS’

- Derivative of Timed Calculus of Broadcasting Systems (TCBS) [5].
- Includes time and broadcast and allows for infinite behaviour.
- Of formalisms considered this was most suitable.
Partial Syntax of the language

\[ T ::= 0 \]

\[ | \text{(in } e\text{)}? T \]

The nil process. Do nothing

\[ | \text{(out } e\text{)}! T \]

Input \( v \) to match \( e \), then become \( T \)

\[ | T + T \]

Choice between processes

\[ | A(\tilde{e}) \]

Parameterised process \( A \)

\[ | \text{del}(d).T \]

Delay by \( d \) and then become \( T \)

\[ | (T | T) \]

Parallel Composition of processes
Partial Syntax of the language

\[ T ::= 0 \]
\[ \mid (\text{in } e)?T \quad \text{The nil process. Do nothing} \]
\[ \mid (\text{out } e)!T \quad \text{Input } v \text{ to match } e, \text{ then become } T \]
\[ \mid T + T \quad \text{Output } \llbracket e \rrbracket, \text{ then become } T \]
\[ \mid A(\vec{e}) \quad \text{Choice between processes} \]
\[ \mid \text{del}(d).T \quad \text{Parameterised process } A \]
\[ \mid (T | T) \quad \text{Delay by } d \text{ and then become } T \]
\[ \quad \text{Parallel Composition of processes} \]
Partial Syntax of the language

\[ T ::= 0 \]
\[ | (\text{in } e)?T \quad \text{The nil process. Do nothing} \]
\[ | (\text{out } e)!T \quad \text{Input } v \text{ to match } e, \text{ then become } T \]
\[ | T + T \quad \text{Output } \llbracket e \rrbracket, \text{ then become } T \]
\[ | A(\vec{e}) \quad \text{Choice between processes} \]
\[ | \text{del}(d).T \quad \text{Parameterised process } A \]
\[ | (T \mid T) \quad \text{Delay by } d \text{ and then become } T \]
\[ | \quad \text{Parallel Composition of processes} \]
Partial Syntax of the language

\[ T ::= 0 \quad \text{The nil process. Do nothing} \]
\[ \mid (\text{in } e)?T \quad \text{Input } v \text{ to match } e, \text{ then become } T \]
\[ \mid (\text{out } e)!T \quad \text{Output } \llbracket e \rrbracket, \text{ then become } T \]
\[ \mid T + T \quad \text{Choice between processes} \]
\[ \mid A(\vec{e}) \quad \text{Parameterised process } A \]
\[ \mid \text{del}(d).T \quad \text{Delay by } d \text{ and then become } T \]
\[ \mid (T \mid T) \quad \text{Parallel Composition of processes} \]
Partial Syntax of the language

\[ T ::= 0 \quad \text{The nil process. Do nothing} \]
\[ (\text{in } e)?T \quad \text{Input } v \text{ to match } e, \text{ then become } T \]
\[ (\text{out } e)!T \quad \text{Output } \llbracket e \rrbracket, \text{ then become } T \]
\[ T + T \quad \text{Choice between processes} \]
\[ A(\vec{e}) \quad \text{Parameterised process } A \]
\[ \text{del}(d).T \quad \text{Delay by } d \text{ and then become } T \]
\[ (T | T) \quad \text{Parallel Composition of processes} \]
Partial Syntax of the language

\[
T ::= 0 \quad \text{The nil process. Do nothing}
\]

\[
| (\text{in } e)?T \quad \text{Input } v \text{ to match } e, \text{ then become } T
\]

\[
| (\text{out } e)!T \quad \text{Output } \llbracket e \rrbracket, \text{ then become } T
\]

\[
| T + T \quad \text{Choice between processes}
\]

\[
| A(\vec{e}) \quad \text{Parameterised process } A
\]

\[
| \text{del}(d).T \quad \text{Delay by } d \text{ and then become } T
\]

\[
| (T | T) \quad \text{Parallel Composition of processes}
\]
Partial Syntax of the language

\[ T ::= 0 \quad \text{The nil process. Do nothing} \]
\[ (\text{in } e)\? T \quad \text{Input } v \text{ to match } e, \text{ then become } T \]
\[ (\text{out } e)! T \quad \text{Output } \llbracket e \rrbracket, \text{ then become } T \]
\[ T + T \quad \text{Choice between processes} \]
\[ A(\tilde{e}) \quad \text{Parameterised process } A \]
\[ \text{del}(d).T \quad \text{Delay by } d \text{ and then become } T \]
\[ (T | T) \quad \text{Parallel Composition of processes} \]
Some Semantics of the language

Example Semantics

Inputting a value:

\[
\frac{v = e[\vec{v}/f_v(e)]}{(\text{in } e)?T \xrightarrow{v?} T[\vec{v}/f_v(e)]}
\]

Outputting a value:

\[
\frac{[e] = w}{(\text{out } e)!P \xrightarrow{w!} P}
\]

Delaying:

\[
\frac{d' \leq d}{\text{del}(d).P \xrightarrow{d'} \text{del}(d - d').P}
\]
Some Semantics of the language

**Example Semantics**

**Inputting a value:**

\[ \nu = e[\vec{v}/f_\nu(e)] \]

\( (\text{in } e)?T \xrightarrow{\nu^?} T[\vec{v}/f_\nu(e)] \)

**Outputting a value:**

\[ \llbracket e \rrbracket = w \]

\( (\text{out } e)!P \xrightarrow{w!} P \)

**Delaying:**

\[ d' \leq d \]

\( \text{del}(d).P \xrightarrow{d'} \text{del}(d - d').P \)
Some Semantics of the language

Example Semantics

Inputting a value:

\[
\frac{v = e[\vec{v}/f_v(e)]}{(\text{in } e)?T \xrightarrow{v?} T[\vec{v}/f_v(e)]}
\]

Outputting a value:

\[
\frac{\llbracket e \rrbracket = w}{(\text{out } e)!P \xrightarrow{w!} P}
\]

Delaying:

\[
\frac{d' \leq d}{\text{del}(d).P \xrightarrow{d'} \text{del}(d - d').P}
\]
Overview of the Formalised System

- System is a parallel composition of $n$ entity processes.
- Overall description of entity is quite large- two pages of algebraic expressions.
- Let’s look here at an entity from the top down- just the main component processes.
The big three

\[ E(e, s, v) \overset{\text{def}}{=} \text{Protocol}(e) \mid \text{Environment}(e) \mid \text{Mobile}(0, s, v) \]

- \( E(e, s, v) \): Entity identified by \( e \) in position \( s \) with velocity \( v \).
- \( \text{Protocol}(e) \): Component responsible for executing Comhordú protocol.
- \( \text{Environment}(e) \): Models the environment and the space elastic model interface.
- \( \text{Mobile}(0, s, v) \): Models the position/velocity of the entity over time.
The big three

\[ E(e, s, v) \overset{\text{def}}{=} \text{Protocol}(e) | \text{Environment}(e) | \text{Mobile}(0, s, v) \]

- \( E(e, s, v) \): Entity identified by \( e \) in position \( s \) with velocity \( v \).
- Protocol\((e)\): Component responsible for executing Comhordú protocol.
- Environment\((e)\): Models the environment and the space elastic model interface.
- Mobile\((0, s, v)\): Models the position/velocity of the entity over time.
The big three

\[ E(e, s, v) \overset{\text{def}}{=} \text{Protocol}(e) \mid \text{Environment}(e) \mid \text{Mobile}(0, s, v) \]

- **E(e, s, v)**: Entity identified by \( e \) in position \( s \) with velocity \( v \).
- **Protocol(e)**: Component responsible for executing Comhordú protocol.
- **Environment(e)**: Models the environment and the space elastic model interface.
- **Mobile(0, s, v)**: Models the position/velocity of the entity over time.
The big three

\[ E(e, s, v) \overset{\text{def}}{=} \text{Protocol}(e) \mid \text{Environment}(e) \mid \text{Mobile}(0, s, v) \]

- \( E(e, s, v) \): Entity identified by \( e \) in position \( s \) with velocity \( v \).
- \( \text{Protocol}(e) \): Component responsible for executing Comhordú protocol.
- \( \text{Environment}(e) \): Models the environment and the space elastic model interface.
- \( \text{Mobile}(0, s, v) \): Models the position/velocity of the entity over time.
The big three

\[ E(e, s, v) \overset{\text{def}}{=} \text{Protocol}(e) \mid \text{Environment}(e) \mid \text{Mobile}(0, s, v) \]

- \( E(e, s, v) \): Entity identified by \( e \) in position \( s \) with velocity \( v \).
- \( \text{Protocol}(e) \): Component responsible for executing Comhordú protocol.
- \( \text{Environment}(e) \): Models the environment and the space elastic model interface.
- \( \text{Mobile}(0, s, v) \): Models the position/velocity of the entity over time.
The big three

\[ E(e, s, v) \overset{\text{def}}{=} \text{Protocol}(e) \mid \text{Environment}(e) \mid \text{Mobile}(0, s, v) \]

- **E(e, s, v)**: Entity identified by \( e \) in position \( s \) with velocity \( v \).
- **Protocol(e)**: Component responsible for executing Comhordú protocol.
- **Environment(e)**: Models the environment and the space elastic model interface.
- **Mobile(0, s, v)**: Models the position/velocity of the entity over time.
Protocol Components

\[ \text{Protocol}(e) \overset{\text{def}}{=} \text{Active}(0) \mid \text{LisAct}(e, 0) \mid \text{ANCheckA}(0) \mid \ldots \]

- **Active(0)**: Periodically broadcasts the current mode of operation & position.
- **LisAct(e, 0)**: Listens for incoming messages from other entities and starts appropriate reactions if necessary.
- **ANCheckA(0)**: Similar to above, but listens for coverage notification messages.
Protocol Components

\[ \text{Protocol}(e) \overset{\text{def}}{=} \text{Active}(0) \mid \text{LisAct}(e, 0) \mid \text{ANCheckA}(0) \mid \ldots \]

- **Active(0):** Periodically broadcasts the current mode of operation & position.

- **LisAct(e, 0):** Listens for incoming messages from other entities and starts appropriate reactions if necessary.

- **ANCheckA(0):** Similar to above, but listens for coverage notification messages.
Protocol Components

\[ \text{Protocol}(e) \overset{\text{def}}{=} \text{Active}(0) | \text{LisAct}(e, 0) | \text{ANCheckA}(0) | \ldots \]

- **Active(0):** Periodically broadcasts the current mode of operation & position.

- **LisAct(e, 0):** Listens for incoming messages from other entities and starts appropriate reactions if necessary.

- **ANCheckA(0):** Similar to above, but listens for coverage notification messages.
Protocol Components

\[
\text{Protocol}(e) \overset{\text{def}}{=} \text{Active}(0) \mid \text{LisAct}(e, 0) \mid \text{ANCheckA}(0) \mid \ldots
\]

- **Active(0):** Periodically broadcasts the current mode of operation & position.

- **LisAct(e, 0):** Listens for incoming messages from other entities and starts appropriate reactions if necessary.

- **ANCheckA(0):** Similar to above, but listens for coverage notification messages.
Protocol Components

\[
\text{Protocol}(e) \overset{\text{def}}{=} \text{Active}(0) \mid \text{LisAct}(e, 0) \mid \text{ANCheckA}(0) \mid \ldots
\]

- **Active(0):** Periodically broadcasts the current mode of operation & position.

- **LisAct(e, 0):** Listens for incoming messages from other entities and starts appropriate reactions if necessary.

- **ANCheckA(0):** Similar to above, but listens for coverage notification messages.
Active, the Message Sending Process

\[
\text{Active}(i) \overset{\text{def}}{=} (\text{out } \text{read}()!((\text{in } \text{read}(s))?((\text{out } \text{toServer}(A, i, s))! (\text{del}(p(i)).\text{Active}(i) + A') + A') + A') + A'
\]

\[
A' \overset{\text{def}}{=} (\text{in } \text{switch}()?\text{Active}(j)
\]

- Active process of an entity in mode \( i \) broadcasts messages.
- Make a read request to get the position.
- Input the position.
- Output message including mode and position.
- Wait for \( p(i) \) units of time, the period, then recursion happens.
- At all points, listen for mode switches.
- Listener sub-process: Given \( \text{switch}(j) \) message, switch to mode \( j \).
Active, the Message Sending Process

\[
\text{Active}(i) \equiv (\text{out } \text{read}())!((\text{in } \text{read}(s))?(\text{out } \text{toServer}(A, i, s))!
\]
\[
(\text{del}(p(i)).\text{Active}(i) + A') + A') + A'
\]

\[
A' \equiv (\text{in } \text{switch}(j))?\text{Active}(j)
\]

- Active process of an entity in mode \(i\) broadcasts messages.
- Make a read request to get the position.
- Input the position.
- Output message including mode and position.
- Wait for \(p(i)\) units of time, the period, then recursion happens.
- At all points, listen for mode switches.
- Listener sub-process: Given \(\text{switch}(j)\) message, switch to mode \(j\).
Active, the Message Sending Process

\[ \text{Active}(i) \overset{\text{def}}{=} (\text{out } \text{read}\langle \rangle)!((\text{in } \text{read}\langle s \rangle)?((\text{out } \text{toServer}\langle A, i, s \rangle)!\text{(del}(p(i))\text{.Active}(i) + A') + A') + A') + A' \]

\[ A' \overset{\text{def}}{=} (\text{in } \text{switch}\langle j \rangle)?\text{Active}(j) \]

- Active process of an entity in mode \( i \) broadcasts messages.
- Make a read request to get the position.
- Input the position.
- Output message including mode and position.
- Wait for \( p(i) \) units of time, the period, then recursion happens.
- At all points, listen for mode switches.
- Listener sub-process: Given \( \text{switch}\langle j \rangle \) message, switch to mode \( j \).
Active, the Message Sending Process

\[
\text{Active}(i) \overset{\text{def}}{=} (\text{out read})!((\text{in read})?((\text{out toServer})\langle A, i, s \rangle)! (\text{del}(p(i)).\text{Active}(i) + A') + A') + A'
\]

\[
A' \overset{\text{def}}{=} (\text{in switch})?\text{Active}(j)
\]

- Active process of an entity in mode \( i \) broadcasts messages.
- Make a read request to get the position.
- **Input the position.**
- Output message including mode and position.
- Wait for \( p(i) \) units of time, the period, then recursion happens.
- At all points, listen for mode switches.
- Listener sub-process: Given \( \text{switch}\langle j \rangle \) message, switch to mode \( j \).
Active, the Message Sending Process

\[
\text{Active}(i) \overset{\text{def}}{=} (\text{out read}(\langle \rangle))!((\text{in read}(\langle s \rangle))?(\text{out toServer}(\langle A, i, s \rangle))! (\text{del}(p(i)).\text{Active}(i + A') + A') + A') + A'
\]

\[
A' \overset{\text{def}}{=} (\text{in switch}(\langle j \rangle))?\text{Active}(j)
\]

- Active process of an entity in mode \( i \) broadcasts messages.
- Make a read request to get the position.
- Input the position.
- Output message including mode and position.
- Wait for \( p(i) \) units of time, the period, then recursion happens.
- At all points, listen for mode switches.
- Listener sub-process: Given \( \text{switch}(\langle j \rangle) \) message, switch to mode \( j \).
Active, the Message Sending Process

\[
\text{Active}(i) \overset{\text{def}}{=} (\text{out read}\langle\rangle)!((\text{in read}\langle s\rangle)?((\text{out toServer}\langle A, i, s\rangle)!
(\text{del}(p(i)).\text{Active}(i) + A') + A') + A') + A' \\
A' \overset{\text{def}}{=} (\text{in switch}\langle j\rangle)?\text{Active}(j)
\]

- Active process of an entity in mode \(i\) broadcasts messages.
- Make a read request to get the position.
- Input the position.
- Output message including mode and position.
- Wait for \(p(i)\) units of time, the period, then recursion happens.
- At all points, listen for mode switches.
- Listener sub-process: Given \(\text{switch}\langle j\rangle\) message, switch to mode \(j\).
Active, the Message Sending Process

\[
\text{Active}(i) \overset{\text{def}}{=} \text{out read}(\langle \rangle)!(\text{in read}(\langle s \rangle)?(\text{out toServer}(A, i, s))! (\text{del}(p(i)).\text{Active}(i) + A') + A') + A' \\
A' \overset{\text{def}}{=} \text{in switch}(\langle j \rangle)?\text{Active}(j)
\]

- Active process of an entity in mode \( i \) broadcasts messages.
- Make a read request to get the position.
- Input the position.
- Output message including mode and position.
- Wait for \( p(i) \) units of time, the period, then recursion happens.
- At all points, listen for mode switches.
- Listener sub-process: Given \( \text{switch}(\langle j \rangle) \) message, switch to mode \( j \).
Active, the Message Sending Process

\[
\text{Active}(i) \overset{\text{def}}{=} (\text{out read⟨⟩})!((\text{in read⟨s⟩})?((\text{out toServer⟨A, i, s⟩})!
(\text{del}(p(i)).\text{Active}(i) + A') + A') + A') + A'
\]

\[
A' \overset{\text{def}}{=} (\text{in switch⟨j⟩})?\text{Active}(j)
\]

- Active process of an entity in mode \(i\) broadcasts messages.
- Make a read request to get the position.
- Input the position.
- Output message including mode and position.
- Wait for \(p(i)\) units of time, the period, then recursion happens.
- At all points, listen for mode switches.
- **Listener sub-process:** Given \(\text{switch⟨j⟩}\) message, switch to mode \(j\).
Active, the Message Sending Process

\[
\text{Active}(i) \overset{\text{def}}{=} (\text{out } \text{read}⟨⟩)!((\text{in } \text{read}⟨s⟩)?((\text{out } \text{toServer}⟨A, i, s⟩)! \\
(\text{del}(p(i)).\text{Active}(i) + A′) + A′) + A′)
\]

\[
A′ \overset{\text{def}}{=} (\text{in } \text{switch}⟨j⟩)?\text{Active}(j)
\]

- Active process of an entity in mode \textit{i} broadcasts messages.
- Make a read request to get the position.
- Input the position.
- Output message including mode and position.
- Wait for \textit{p(i)} units of time, the period, then recursion happens.
- At all points, listen for mode switches.
- Listener sub-process: Given \textit{switch}⟨\textit{j}⟩ message, switch to mode \textit{j}.
The System in Action

- We’ve built an Active process out of the language syntax.
- Now let us evolve it via the language semantics.
A trace of the Active Process

\[ [e] = w \]

(out \(e\)!\(P \xrightarrow{w!} P\))

Active(\(i\)) \(\overset{\text{def}}{=}\)

(out \(\text{read}()\)!((in \(\text{read}()\)?((out toServer\(\langle A, i, s\rangle\))!)
(del(\(p(i))\).Active(\(i\) + A’) + A’) + A’) + A’

Active(\(i\)) \xrightarrow{\text{read}()!} P_1 \xrightarrow{\text{read}()\langle 5.5,8.0\rangle?} P_2 \xrightarrow{\text{toServer}(A,i,(5.5,8.0))!} P_3

\(d' \xrightarrow{\text{switch}()?} P_4 \xrightarrow{\text{Active}(j)}\)
A trace of the Active Process

\[ \nu = e[\vec{v}/f\nu(e)] \]
\[ (\text{in } e)?T \xrightarrow{\nu?} T[\vec{v}/f\nu(e)] \]

\[ \mathit{P}_1 \overset{\text{def}}{=} (\text{in } \text{read}(s))?((\text{out toServer}(A, i, s))! (\text{del}(p(i)).\text{Active}(i) + A') + A') + A' \]

\[
\begin{align*}
\text{Active}(i) & \xrightarrow{\text{read}()}! \mathit{P}_1 \xrightarrow{\text{read}((5.5,8.0))?} \mathit{P}_2 \xrightarrow{\text{toServer}(A,i,(5.5,8.0))!} \mathit{P}_3 \\
& \xrightarrow{d'} \mathit{P}_4 \xrightarrow{\text{switch}(j)?} \text{Active}(j)
\end{align*}
\]
A trace of the Active Process

\[
[e] = w \\
\text{(out } e)!!P \xrightarrow{w!} P
\]

\[
P_2 \overset{\text{def}}{=} \text{(out } toServer\langle A, i, (5.5, 8.0)\rangle!!(\text{del}(p(i)).\text{Active}(i) + A') + A'
\]

\[
\begin{align*}
\text{Active}(i) & \xrightarrow{\text{read}\langle\rangle!} P_1 & \text{read}\langle(5.5,8.0)\rangle? & \xrightarrow{} P_2 & \text{toServer}\langle A, i, (5.5,8.0)\rangle! & \xrightarrow{} P_3 \\
& \xrightarrow{d'p} P_4 & \text{switch}\langle j\rangle? & \xrightarrow{} \text{Active}(j)
\end{align*}
\]
A trace of the Active Process

\[ \text{del}(d).P \xrightarrow{d'} \text{del}(d - d').P \]

\[ P_3 \overset{\text{def}}{=} \text{del}(p(i)).\text{Active}(i) + A' \]

Active(i) \xrightarrow{\text{read}()}! P_1 \xrightarrow{\text{read}((5.5,8.0))?} P_2 \xrightarrow{\text{toServer}(A,i,(5.5,8.0))!} P_3

\[ d' \xrightarrow{} P_4 \xrightarrow{\text{switch}(<j>)?} \text{Active}(j) \]
A trace of the Active Process

\[ v = e[\vec{v}/fv(e)] \]
\[ (in\ e)?T \xrightarrow{v?} T[\vec{v}/fv(e)] \]

\[ P_4 \overset{\text{def}}{=} \]
\[ \text{del}(p(i) - d').\text{Active}(i) + A' \]

\[ \text{Active}(i) \xrightarrow{\text{read}()!} P_1 \xrightarrow{\text{read}((5.5,8.0))?} P_2 \xrightarrow{\text{toServer}(A,i,(5.5,8.0))!} P_3 \]
\[ d' \xrightarrow{} P_4 \xrightarrow{\text{switch}(j)?} \text{Active}(j) \]
A trace of the Active Process

\[
\text{Active}(i) \stackrel{\text{read}()}! \rightarrow P_1 \stackrel{\text{read}((5.5,8.0))?}{\rightarrow} P_2 \stackrel{\text{toServer}(A,i,(5.5,8.0))!}{\rightarrow} P_3
\]

\[
d' \rightarrow P_4 \stackrel{\text{switch}(j)?}{\rightarrow} \text{Active}(j)
\]
Summary

- Comhordú: A coordination protocol for systems of autonomous entities communicating over a wireless network.
- TCBS’: A formal process calculus for specifying real-time, broadcasting systems.
- Comhordú in TCBS’: A formalisation of Comhordú in TCBS’.
Benefits of Formalising Comhordú

- Ambiguity present in the original informal description has been removed.
- More detail is provided as to how the protocol is actually achieved e.g. overlapping broadcast for "soft handover".
- A formal model can be verified by rigorous methods.
Future Work

- Explore the model with a property logic.
- Approximate the model in a model checker such as UPPAAL [4].
- Focus on instances of the model as its state space may be too large to explore.
And that’s it!

- Thank you all for listening.
- Questions?
- Comments?
M. Bouroche and V. Cahill.
We don’t need to agree to coordinate.
2008.

Mlanie Bouroche.
*Real-Time Coordination of Mobile Autonomous Entities.*

B. Hughes, R. Meier, R. Cunningham, and V. Cahill.
Towards real-time middleware for vehicular ad hoc networks.

K.G. Larsen, P. Pettersson, and W. Yi.
UPPAAL in a nutshell.

K. Prasad.
Broadcasting in time.
*Coordination Languages and Models*, pages 321–328, 1996.