Formal derivation of a distributed program
These slides present a formal derivation of a distributed system which purpose is to choose a single leader among all the participating processes. The solution is inspired by the *bully algorithm* presented in [1].
Introduction

An experiment in developing a piece of software using Rodin.
Distributed system-level software coordinating hardware of several spacecrafts.

Not a real system: a feasibility case study.

The problem is well-suited for Rodin methods: not large but deep.
Introduction
Introduction

- A fairly simple distributed system
- No failures, time limits, fairness, probabilistic properties
- A combination of two well-known algorithms
Goals

Our task is to derive a correct-by-construction *program* implementing a distributed protocol.

This differs from building a *model* proving the correctness of the protocol.

We are not interested in (re-)proving the correctness of the involved algorithm.
Goals

To achieve our goal we have to give a detailed model of the algorithm and prove its essential properties.

- Decomposition (essential to constructing a program) is not possible without strong invariants
- A considerable semantic gap between available formalisations and Event B
The development strategy

We are going to start with a trivial model of a high-level observation of leader election.

From there we gradually uncover more details about the election mechanism.

We are going to use decomposition to separate the model of environment from the models of constituent processes.
The abstract model is a statement of our intention to construct a leader election protocol. For a system with \( n \) processes a leader is a process index from range \( 1 \ldots n \).

\[
\text{leader} \in 1 \ldots n
\]

At this level, the whole election protocol is addressed by a single event non-deterministically selecting a new leader.

\[
elect = \text{any } nl \text{ where } nl \in 1 \ldots n \text{ then } \text{leader} := nl \text{ end}
\]
First refinement

In this refinement we make a step towards the distribution of state among the process. Each process is free to decide whether it wants to be a new leader or not. Such decision is made by a process independently of other processes and is recorded in a global vector of decisions:

\[
\text{decision} : 1 \ldots n \leftrightarrow 0 \ldots n
\]

At this point we are bringing some ideas of the bully protocol. To determine the leader among the set of willing processes we compare their "bully id’s" - statically allocated unique numbers. In our case, since we already use associate processes with numbers in the decision vector, the bully id of a process is the process number as defined by the decision vector.

\[
\forall i \cdot i \in \text{dom}(\text{decision}) \implies \text{decision}(i) \in \{0, i\}
\]
First refinement

When a process makes a decision it puts into the decision vector either its id, indicating that it is willing to be a new leader, or 0, indicating that it is not willing to be a leader.

\[
\forall i \cdot i \in \text{dom}(\text{decision}) \implies \text{decision}(i) \in \{0, i\}
\]

According to the bully algorithm, a new leader is a process with the maximum id among the processes that are willing to be leaders. Assuming the decision vector is complete, the new leader id is

\[
\max(\text{ran}(\text{decision}))
\]
First refinement

A process registers its decision by updating the decision vector.

\[
\text{\textbf{vote}} = \text{any \ } idx, \ d \ \text{where}
\]
\[
\begin{align*}
&idx \in 1..n \\
&idx \notin \text{dom}(\text{decision}) \\
&d \in \{idx, 0\}
\end{align*}
\]
\[
\text{then}
\]
\[
\text{decision}(idx) := d
\]
\[
\text{end}
\]

Unfortunately, all the processes may refuse being a leader and then the election has to be restarted. This means that a protocol round is potentially divergent.
First refinement

We insist on protocol rounds that result in the election of a leader. In other words, there should never be a situation when every process decides not to be a leader. To exclude such possibility we require that any process willing to initiate the protocol is also committing to be a leader. The corresponding safety property states that whenever the decision vector is not empty there is a record of a process willing to be a new leader:

\[
\text{card}(\text{decision}) \geq 1 \implies \text{decision}^{-1}[1 \ldots n] \neq \emptyset
\]

The new invariant guarantees that at the end of a round we have a new leader id:

\[
\text{dom}(\text{decision}) = 1 \ldots n \implies \max(\text{ran}(\text{decision})) \in 1 \ldots n
\]
First refinement

The vote event violates this property and is replaced by two new events.

The first voting event addresses the protocol initiation stage. A process may decide to initiate the protocol as long as it has not voted already.

\[
\text{initiate} = \text{any } idx \text{ where } \\
idx \in 1 \ldots n \\
idx \notin \text{dom}(\text{decision}) \\
\text{then} \\
\text{decision}(idx) := idx \\
\text{end}
\]

The protocol round may be already active when a process decides to initiate. Then it simply joins the round with a decision to be a new leader.
First refinement

The second voting event allows a process to refuse to be a leader provided it has observed that at least one other process has made a decision.

\[
\begin{align*}
\text{decide} & = \text{any } idx, d \text{ where } \\
& \quad idx \in 1..n \\
& \quad idx \notin \text{dom}(\text{decision}) \\
& \quad d \in \{idx, 0\} \\
& \quad \text{decision} \neq \emptyset \\
\text{then} & \quad \text{decision}(idx) := d \\
\text{end}
\end{align*}
\]
First refinement

The abstract non-deterministic leader election event is refined by event computing the new leader id over the vector of process decisions.

\[
\text{elect} = \text{any } nl \text{ where } \\
\quad \text{dom}(\text{decision}) = 1 \ldots n \\
\quad nl = \text{max}(\text{ran}(\text{decision})) \\
\quad \text{then} \\
\quad \text{leader} := nl \\
\text{end}
\]
In the previous refinement step there are two cases where a process has to look into the memory of other processes: in the decide event a process considers the whole decision vector to figure out if it is possibly an initiating process; in the elect events, the global decision vector is used to compute the new leader id.

Instead of consulting the global decision vector we introduce a local, incomplete views on the overall decision vector. We will show that a local view allows a process to independently compute the new leader id.
To avoid looking into the global decision vector to see what processes have decided, a process maintains a local knowledge about other process decisions.

\[ \text{other} \in 1 \ldots n \rightarrow \mathbb{P}(0 \ldots n) \]

A process local knowledge is consistent with the global decision vector and does not include the decision of the current process.

\[ \forall i \cdot i \in 1 \ldots n \implies \text{other}(i) \subseteq \text{ran}(\{i\} \triangleleft \text{decision}) \]
Second refinement

We need some mechanism to populate the local knowledge of a process with the information about other process decisions without directly looking into the global decision vector. This necessitates a simple communication model which, in our case, has the following general structure:

- process makes a decision
- process sends a message to another process informing it of the decision
- process may receive a message about other process decision; the message is used to update the local knowledge of the process
- when the local knowledge contains the information about all other process decisions, the process is able to determine the overall leader (sometimes this may be done earlier)
Each process maintains the record of processes from which it has received a message.

\[ \text{recv} \in 1 \ldots n \rightarrow \mathbb{P}(1 \ldots n) \]

There is a simple relationship between the set of processes that has communicated their decision and the contents of the local knowledge:

\[ \forall i \cdot i \in 1 \ldots n \implies \text{decision}[\text{recv}(i)] = \text{other}(i) \]
Second refinement

The second part of decision communication is the act of sending a message. This is accomplished in two steps: 1) process commits to sending a message; 2) a message is communicated between two processes.

A process keeps a set of process id’s to which it has committed to communicate a message. Obviously, this set exclude the current process.

\[
\text{pending} \in 1..n \rightarrow \mathbb{P}(1..n)
\]

\[
\forall i \cdot i \in 1..n \implies i \notin \text{pending}(i)
\]

The meaning of \text{pending} the following: what is committed to be sent by one process has not been yet received by another process.

\[
\forall i \cdot i \in 1..n \implies (\forall j \cdot j \in 1..n \land i \in \text{pending}(j) \implies j \notin \text{recv}(i))
\]
The abstract *leader* variable is split into a vector so that each process is able to independently compute the new leader id, which is, of course, the same leader id as computed by the abstraction.

\[
\text{leaders} \in 1 .. n \mapsto 1 .. n \\
\forall i \cdot i \in \text{dom}(\text{leaders}) \implies \text{leaders}(i) = \text{leader}
\]
The central property of this refinement step is that the leader id determined using local knowledge is the actual leader as defined by the global decision vector.

\[ \forall i \cdot i \in \text{dom}(\text{decision}) \land \\
\text{recv}(i) = \text{dom}([i] \triangleleft \text{decision}) \implies \\
\max(\text{ran}(\text{decision})) = \max(\text{other}(i) \cup \{\text{decision}(i)\}) \]

This may be considered as a proof of the protocol correctness.
Second refinement

There are two new events: one for a process sending a decision message and another for the reception of a message. The sending event generates a new message that has not yet been received by some destination process to.

\[
\text{send} = \text{any} \ idx, to \ \text{where} \\
\quad idx \in \text{dom}(\text{decision}) \\
\quad to \in 1..n \setminus \{idx\} \\
\quad idx \notin \text{recv}(to) \\
\quad \text{then} \\
\quad \text{pending}(idx) := \text{pending}(idx) \cup \{to\} \\
\quad \text{end}
\]

The event is divergent. It will be made convergent in the next refinement step.
Second refinement

We also need a message reception event. The event updates the local view \(\textit{other}\) of a process and updates sets of received and pending messages.

\[
\text{receive} = \text{any } idx, to \text{ where} \\
\quad idx \in \text{dom}(\text{pending}) \\
\quad \text{pending}(idx) \neq \emptyset \\
\quad to \in \text{pending}(idx) \\
\text{then} \\
\quad \text{recv}(to) := \text{recv}(to) \cup \{idx\} \\
\quad \text{other}(to) := \text{other}(to) \cup \{\text{decision}(idx)\} \\
\quad \text{pending}(idx) := \text{pending}(idx) \setminus \{to\} \\
\text{end}
\]
A process is able to compute the overall leader once the local knowledge contains the decisions of all other processes.

\[ \text{leaders}(idx) := \max(\text{other}(idx) \cup \{\text{decision}(idx)\}) \]

A process may be able to find the leader even before obtaining the complete local knowledge (e.g., because it has the maximum id and is willing to be a leader). Such optimisations are best introduced via refinement after the decomposition step.
Second refinement: animation

- decision
- other
- pending
- recv
- leader
Second refinement: animation
Second refinement: animation

- decision
- other
- pending
- recv
- leader
Second refinement: animation

- **Decision**: 1
  - Other: 2
  - Pending: ?
  - Recev: 2
  - Leader: ?

- **Column 2**
  - Other: ?
  - Pending: ?
  - Recev: ?
  - Leader: ?

- **Column 3**
  - Other: ?
  - Pending: ?
  - Recev: ?
  - Leader: ?
Second refinement: animation

<table>
<thead>
<tr>
<th>decision</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

| other    | 2 | ? | ? | ? | ? |
| recv     | 2 | ? | ? | ? | ? |
Second refinement: animation

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>decision</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>other</td>
<td>2</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>pending</td>
<td>3</td>
<td>?</td>
<td>3</td>
</tr>
<tr>
<td>recv</td>
<td>2</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>leader</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
Second refinement: animation

```
<table>
<thead>
<tr>
<th>decision</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

| other    | 2 | ? | ? |
|          | 3 | ? | 0 |

| pending  | ? | ? | ? |
|          | 3 | ? | ? |

| recv     | 2 | ? | 1 |
|          | 3 | ? | ? |

| leader   | ? | ? | ? |
|          | ? | ? | ? |
```
Second refinement: animation

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>decision</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>other</td>
<td>2 3</td>
<td>?  ?</td>
<td>0  ?</td>
</tr>
<tr>
<td>pending</td>
<td>?  ?</td>
<td>3  ?</td>
<td>2  ?</td>
</tr>
<tr>
<td>recv</td>
<td>2 3</td>
<td>?  ?</td>
<td>1  ?</td>
</tr>
<tr>
<td>leader</td>
<td>3</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
Second refinement: animation
Second refinement: animation

```
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>decision</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>other</td>
<td>2 3</td>
<td>3 0</td>
<td>0 2</td>
</tr>
<tr>
<td>recv</td>
<td>2 3</td>
<td>3 1</td>
<td>1 2</td>
</tr>
<tr>
<td>leader</td>
<td>3</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
```
Second refinement: animation

\[
\begin{array}{c|c|c}
\text{decision} & 1 & 2 \\
\hline
0 & 2 & 3 \\
\text{other} & 2 & 3 \\
\hline
\text{pending} & ? & ? \\
\hline
\text{recv} & 2 & 3 \\
\hline
\text{leader} & 3 & 3 \\
\end{array}
\]
In the second refinement step there are several cases where what is logically a private memory of a process is accessed directly by another process. Let us assume the following structural property of the model

\[
\text{var}(i) \quad \text{is private memory of process } i
\]

We do need an exception for this rule to permit information flow between the processes. The sole exception is \text{pending}(i) which may be read by another process \(j\). Around this exception we are going to build the model of communication.
The private memory assumption is violated in several places.

\[
\text{send} = \text{any } idx, \text{to where }
\]
\[
\begin{align*}
idx & \in \text{dom}(\text{decision}) \\
to & \in 1 \ldots n \setminus \{idx\} \\
idx & \not\in \text{recv}(\text{to})
\end{align*}
\]
\[
\text{then}
\]
\[
\text{pending}(idx) := \text{pending}(idx) \cup \{to\}
\]
\[
\text{end}
\]

recv(to) should not be read by process idx.
Third refinement

To deal with $idx \notin \text{recv}(to)$ issues we introduce an additional data structure that keeps the id’s of processes to which a decision update has been sent

\[
\text{sent} \in 1..n \rightarrow \mathbb{P}(1..n)
\]

$sent(i)$ is a complete history of outgoing messages of process $i$; it includes all the messages currently being transmitted

\[
\forall i \cdot i \in 1..n \Rightarrow \text{pending}(i) \subseteq \text{sent}(i)
\]
Third refinement

With the addition of \textit{sent} we get a very desirable property of the model: asynchronous communication.

\[
\forall i \cdot i \in 1..n \implies
(\forall j \cdot j \in 1..n \implies
(j \in \text{sent}(i) \setminus \text{pending}(i) \iff i \in \text{recv}(j)))
\]

if a process \(j\) has received a decision update from process \(i\) then process \(i\) has sent a decision update to process \(j\) and this update message is not currently in transition (not in set \textit{pending}); the same property holds in the other direction: what has been sent and is not being transmitted has been received.

The combination of \textit{sent}, \textit{pending} and \textit{recv} describes a one-to-one asynchronous communication channel.
Updated send event.

\[
\text{send} = \text{any } idx, to \text{ where } \\
\ldots \\
to \notin \text{sent}(idx) \quad // \quad idx \notin \text{recv}(to) \\
\text{then} \\
\text{sent}(idx) := \text{sent}(idx) \cup \{to\} \\
\ldots \\
\end{array}
\]

Now we are able to prove the convergence of send.
Fourth refinement

After introducing sent, vector recv is not used anymore in the role of prevention of duplicate messages. Therefore, it may be refined into a simpler data type.

\[
\begin{align*}
\text{irec} &\in 1 \ldots n \rightarrow 0 \ldots n - 1 \\
\forall i \cdot i \in 1 \ldots n \implies \text{irec}(i) = \text{card}(\text{recv}(i))
\end{align*}
\]

Everywhere in the model, a usage of recv is refined into an expression defined on irec.
Fifth refinement

One remaining violation of the private memory rule is reading the decision vector of another process when receiving a message.

\[
\text{receive} = \text{any } idx, \text{to where} \\
\quad idx \in \text{dom}(\text{pending}) \\
\quad \text{pending}(idx) \neq \emptyset \\
\quad \text{to} \in \text{pending}(idx) \\
\text{then} \\
\quad \text{irec(to)} := \text{irec(to)} \cup \{idx\} \\
\quad \text{other(to)} := \text{other(to)} \cup \{\text{decision}(idx)\} \\
\quad \text{pending}(idx) := \text{pending}(idx) \setminus \{\text{to}\} \\
\text{end}
\]

The solution is to communicate the decision of the sending process along with id of the destination process.
Fifth refinement

\textit{decision}(idx) is embedded into a message sent by process \textit{idx} to some other process \textit{to}. To accommodate this additional information we define 'extended' versions of \textit{sent} and \textit{pending}:

\[
\begin{align*}
xsent &\in 1 \ldots n \rightarrow (1 \ldots n \rightarrow 0 \ldots n) \\
xpending &\in 1 \ldots n \rightarrow (1 \ldots n \rightarrow 0 \ldots n)
\end{align*}
\]

The abstract \textit{sent} and \textit{pending} are related to \textit{xsent} and \textit{xpending} as the first projection:

\[
\begin{align*}
\forall i \cdot i \in 1 \ldots n \implies sent(i) &= prj1[xsent(i)] \\
\forall i \cdot i \in 1 \ldots n \implies pending(i) &= prj1[xpending(i)]
\end{align*}
\]

Hence, new variables are exactly the same as old ones except for an additional record.
Fifth refinement

This additional record (the payload of a message) has a fixed value within the scope of a process. It holds the process decision.

\[
\forall i \cdot i \in 1..n \land x_{sent}(i) \neq \emptyset \implies \{decision(i)\} = prj2[x_{sent}(i)]
\]
Fifth refinement

Let us take a closer look at the information presented in $x_{sent}$ (and also $x_{pending}$ which is a subset of $x_{sent}$).

$$x_{sent} \in 1..n \rightarrow (1..n \rightarrow 0..n)$$

The domain of $x_{sent}$ is the id of a sending process. Its range is a set of pairs in the form of ($target\ process$, $decision$) where $decision$ is the decision of the sending process. All in all, we have a rather familiar structure of a simple network protocol message:

$$\langle source\ address \rangle | \langle target\ address \rangle | \langle payload \rangle$$

An address is a process id and the payload is the decision of a sending process.
Fifth refinement

At this stage we have achieved the following:

- defined private memory for each process
- defined a simple model for inter-process communication
- demonstrated that the communication is asynchronous
- via data refined arrived at what looks like a message in a point-to-point network protocol

We are ready to decompose the model.
We make a transition from a model of distributed protocol to a model of program realising the distributed protocol. The program that we aim to produce is going to be concerned solely with the computations on the private state of a single process. The behaviour pertaining to inter-process communication will be placed in a separate model.
The system we are trying to build is going to operate on top of the existing hardware and some service-level software. To ensure that the resultant system is deployable we formulate environment assumptions. In our case, the most important aspect of the environment is the networking infrastructure, which we refer to as *middleware*. In simplistic terms, the sole functionality of the middleware is the delivery of a message from one process to another.
Decomposition

Assumptions about the middleware:

▶ the middleware implements a simple point-to-point communication protocol; as a message it expects a data structure containing source and target addresses of the network points as well as the data to be delivered to the target point;

▶ for any message sent, it is guaranteed that the message is eventually delivered\(^1\);

▶ when a message is delivered, the sender gets a delivery receipt;

▶ the middleware is not able to access the internal memory of a process; it only observes the buffer of output messages.

---

\(^1\)In reality, this means that the failure to deliver a message aborts the whole protocol and the consequences of this are dealt with outside of the scope of the modelled system.
Decomposition

Three processes: P, Q, R

Middleware
To extract the model of the middleware we consider the communications among a collection of processes as observations of the messages sent and received by the processes.

We reinterpret the information exchange among processes in the terms of messages histories.
Message histories of a process

\[ ih \in 1 \ldots il \rightarrow MSG \quad \text{input history sequence} \]
\[ oh \in 1 \ldots ol \rightarrow MSG \quad \text{output history sequence} \]
\[ ol \geq r \land olr \leq QL \quad \text{output queue capacity} \]
The *receive* operation reacts on an incoming message. At the interface level the observed effect is a new message in the input history.

\[
\begin{align*}
\text{receive} &= \text{any } m \text{ pre} \\
&m \in \text{MSG} \\
\text{post} \\
&ih' = ih \cup \{il + 1 \mapsto m\} \\
&il' = il + 1 \\
\text{end}
\end{align*}
\]

An actual model would define further pre- and post-conditions.
Sending a message

Operation *deliver* marks a message in the output queue as processed. It also frees one slot in the output message queue.

\[
m \leftarrow \text{deliver} = \text{pre}
\]

\[
\text{ol} > r
\]

\[
\text{post}
\]

\[
m' = oh(r + 1)
\]

\[
r' = r + 1
\]

\[
\text{end}
\]

When the operation is called it returns a message to be delivered.
A component has a thread of control that allows it accomplish some tasks independently of middleware and other components. One observed effect of the thread execution is the generation of new messages in the output queue.

\[
\begin{align*}
    \text{proc} & = \text{guarantee} \\
    ol' & \geq ol \land ol' r \leq \text{QUEUE\_LENGTH} \\
    oh & = 1 \ldots ol \triangle oh'
\end{align*}
\]

The process guarantee states that the process may change the output queue by adding new messages to the queue tail until the capacity limit is reached.
Coordination

When two components are communicating the following properties are maintained:

\[
\begin{align*}
\text{left}_{\text{ih}} &= 1 \ldots \text{right}_r \triangleleft \text{right}_{\text{oh}} \\
\text{left}_{\text{il}} &= \text{right}_r
\end{align*}
\]
Asynchronous communication

In terms of message histories:

\[ left_{ih} = 1 \ldots right_r \triangleleft right_{oh} \]

In the leader election model:

\[
\forall i \cdot i \in 1 \ldots n \implies \\
(\forall j \cdot j \in 1 \ldots n \implies \\
(j \in sent(i) \setminus pending(i) \iff i \in recv(j)))
\]
Data refinement

Input history

\[
\text{irec}(P_{\text{pid}}) = P_{\text{il}} \\
\text{other}(P_{\text{pid}}) = \text{ran}(P_{\text{ih}})
\]
Data refinement

Output history

\[
\begin{aligned}
x_{\text{sent}}(P_{\text{pid}}) &= \text{ran}(P_{\text{oh}}) \\
x_{\text{pending}}(P_{\text{pid}}) &= \text{ran}(P_{\text{r}} + 1..P_{\text{ol}} \triangleleft P_{\text{oh}})
\end{aligned}
\]
Phantom variables

The state of a phantom variable is a part of the private state of a process: its value may not be communicated to other processes. It may be, however, be used in a global invariant to prove the refinement relation. 

`decision` and `leader` are two such private variables that have to appear in the global invariant.

\[
P_{\text{phantom\_decision}} \neq \emptyset \iff P_{\text{pid}} \in \text{dom}(\text{decision})
\]

\[
P_{\text{phantom\_decision}} \neq \emptyset \implies P_{\text{phantom\_decision}} = \{\text{decision}(P_{\text{pid}})\}
\]

\[
P_{\text{phantom\_leader}} \neq \emptyset \iff P_{\text{pid}} \in \text{dom}(\text{leaders})
\]

\[
P_{\text{phantom\_leader}} \neq \emptyset \implies P_{\text{phantom\_leader}} = \{\text{leaders}(P_{\text{pid}})\}
\]
## Proof statistics

<table>
<thead>
<tr>
<th>Step</th>
<th>Total</th>
<th>Auto</th>
<th>Manual</th>
<th>Manual %</th>
</tr>
</thead>
<tbody>
<tr>
<td>elect0</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>elect1</td>
<td>16</td>
<td>13</td>
<td>3</td>
<td>19%</td>
</tr>
<tr>
<td>elect2</td>
<td>76</td>
<td>61</td>
<td>15</td>
<td>20%</td>
</tr>
<tr>
<td>elect3</td>
<td>23</td>
<td>23</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>elect4</td>
<td>19</td>
<td>12</td>
<td>7</td>
<td>37%</td>
</tr>
<tr>
<td>elect5</td>
<td>37</td>
<td>29</td>
<td>8</td>
<td>22%</td>
</tr>
<tr>
<td>elect6</td>
<td>131</td>
<td>106</td>
<td>25</td>
<td>19%</td>
</tr>
<tr>
<td>Overall</td>
<td>361</td>
<td>296</td>
<td>65</td>
<td>18%</td>
</tr>
</tbody>
</table>