Verifying Implementations of CRDTs
SHARED DATA TYPE

q: Queue
\( c: \text{Counter} \)
\[ |q| \leq c \]

\( q.\text{push}(e) \)
\( c.\text{inc()} \)

\( q.\text{val()} \)
\( c.\text{val()} \)

\( c.\text{inc()} \)
Geo-replicated Data Type

\[ q \in \text{Queue} \]
\[ |q| \leq c \]

\[ q.\text{push}(e) \]
\[ c.\text{inc}() \]
\[ q.\text{val}() \]
\[ c.\text{val}() \]

\[ q_3 \in \text{Queue} ? \]
\[ q_1 = q_2 ? \]
\[ |q_1| \leq c_4 ? \]
PROGRAM MODEL (OPERATION)

- $u$: state $\sim (\text{retval, (state } \sim \text{ state}))$
- Prepare (@origin) $u?;$ deliver $u!$
- Read One, Write All
- Deferred-Update Replication
**Program Model (Operation)**

- $u$: state $\rightsquigarrow (\text{retval}, (\text{state} \rightsquigarrow \text{state}))$
- Prepare (@origin) $u?$; deliver $u!$
- Read One, Write All
- Deferred-Update Replication
- \( u: \text{state} \leadsto (\text{retval}, (\text{state} \leadsto \text{state})) \)
- Prepare (@origin) \( u? \); deliver \( u! \)
- Read One, Write All
- Deferred-Update Replication

```
\[ u: \text{state} \leadsto (\text{retval}, (\text{state} \leadsto \text{state})) \]
\[ \text{Prepare (@origin) } u?; \text{ deliver } u! \]
\[ \text{Read One, Write All} \]
\[ \text{Deferred-Update Replication} \]```
PROGRAM MODEL (OPERATION)

- $u: \text{state} \sim (\text{retval}, (\text{state} \sim \text{state}))$
- Prepare (@origin) $u?$; deliver $u!$
- Read One, Write All
- Deferred-Update Replication
PROGRAM MODEL (SYSTEM)
PROGRAM MODEL (SYSTEM)
- Concurrent, Multi-master
- Strong: total order, identical state
- Weak: concurrent, interleaving, no global state
Concurrent, Multi-master
- Strong: total order, identical state
- Weak: concurrent, interleaving, no global state
CRDT EXAMPLES
EXAMPLE: GROW-ONLY COUNTER

\[ x = 0 \]

client

origin replica

replica

replica

inc()
EXAMPLE: GROW-ONLY COUNTER

\[ x = 0 \]

\[ inc() \]

\[ x = ? \]
**Example: Grow-Only Counter**

```
x = 0
```

- **client**
- **origin replica**
- **replica**

```
inc()
```

```
x := x + 1
```

```
x = ?
```
EXAMPLE: GROW-ONLY COUNTER

\[
x = 0 \quad \text{client} \quad \text{inc()}
\]

\[
x := x + 1 \quad \text{origin replica}
\]

\[
x = ? \quad \text{replica}
\]

```
payload integer i
  initial 0
query value () : integer j
  let j = i
update increment ()
  downstream ()
  i := i + 1
```

[A Comprehensive Study of CRDT's (Shapiro et al. 2011)]
**EXAMPLE: GROW-ONLY COUNTER**

\[
x = 0 \quad \text{inc()} \quad x = 2
\]

- **Origin Replica**: Payload integer \( i \)
  - Initial: 0
  - Query: \( \text{value } () : \text{integer } j \)
  - Let: \( j = i \)
  - Update: \( \text{increment } () \)
    - Downstream: \( i := i + 1 \)

[A Comprehensive Study of CRDT’s (Shapiro et al. 2011)]
Example: Observed-Remove Set

\[ s = \{ \} \]

- Client
- Origin replica
- Replicas

add(a)
**EXAMPLE: OBSERVED-REMOVE SET**

\[ s = \{ \} \]

- **add(a)**
- **rem(a)**
EXAMPLE: OBSERVED-REMOVE SET

\[ s = \{ \} \]

\( add(a) \)

\( rem(a) \)

\( s = ? \)
EXAMPLE: OBSERVED-REMOVE SET

\[ s = \{ \} \]

\textbf{add}(a)

\begin{align*}
\text{payload set } & S \\
\text{initial } & \emptyset \\
\text{query } & \text{lookup (element } e \text{): boolean } b \\
\text{let } b & = (\exists u : (e, u) \in S) \\
\text{update } & \text{add (element } e) \\
\text{atSource (} e \text{)} \\
\text{let } & \alpha = \text{unique()} \\
\text{downstream (} e, \alpha \text{)} \\
S & := S \cup \{(e, \alpha)\} \\
\text{update } & \text{remove (element } e) \\
\text{atSource (} e \text{)} \\
\text{pre } & \text{lookup}(e) \\
\text{let } R & = \{(e, u) | \exists u : (e, u) \in S\} \\
\text{downstream (} R \text{)} \\
\text{pre } & \forall (e, u) \in R : \text{add}(e, u) \text{ has been delivered} \\
S & := S \setminus R
\end{align*}

[A Comprehensive Study of CRDT’s (Shapiro et al. 2011)]
EXAMPLE: OBSERVED-REMOVE SET

\[ s = \{ \} \]

add(a)

payload set \( S \)
initial \( \emptyset \)

query \( \text{lookup} \) (element \( e \)) : boolean \( b \)
let \( b = (\exists u : (e,u) \in S') \)

update \( \text{add} \) (element \( e \))
atSource \( e \)
let \( \alpha = \text{unique}() \)
downstream \((e, \alpha)\)
\( S := S \cup \{(e, \alpha)\} \)

update \( \text{remove} \) (element \( e \))
atSource \( e \)
pre \( \text{lookup}(e) \)
let \( R = \{(e,u) | \exists u : (e,u) \in S'\} \)
downstream \((R)\)
pre \( \forall(e,u) \in R: \text{add}(e,u) \) has been delivered
\( S := S \setminus R \)
Anomalies of concurrent updates

- **Bank:**
  - $\sigma_{\text{init}} = 100\€$
  - Alice: $\text{withdraw}(20) = \{ \sigma := 120 \}$
  - Bob: $\text{debit}(60) = \{ \sigma := 40 \}$
  - $\sigma = ???$
Anomalies of concurrent updates

- **Bank:**
  - $\sigma_{\text{init}} = 100\€$
  - Alice: $\text{withdraw}(20) = \{ \sigma := 120 \}$
  - Bob: $\text{debit}(60) = \{ \sigma := 40 \}$
  - $\sigma = \text{??？??}$

- **File system:**
  - $\sigma_{\text{init}} = \text{/}$$\text{foo}$
  - Alice: $\text{mkdir} \text{/foo}; \text{mkdir} \text{/foo/bar}$
  - Bob: receives $\text{mkdir} \text{/foo/bar}$
  - $\sigma = \text{??？??}$
Eventual Consistency

Don’t show photos to Bob

- access (Bob, photo) $\Longrightarrow$ ACL (Bob, photo)
- $v$ observed effects of $u$ $\Longrightarrow$ $v$ should be delivered after $u$
- Available: doesn’t slow down sender
Eventual Consistency

- access (Bob, photo) $\rightarrow$ ACL (Bob, photo)
- $v$ observed effects of $u$ $\rightarrow$ $v$ should be delivered after $u$
- Available: doesn’t slow down sender
Eventual Consistency

- access (Bob, photo) $\rightarrow$ ACL (Bob, photo)
- v observed effects of u $\rightarrow$ v should be delivered after u
- Available: doesn’t slow down sender
Eventual Consistency

- access (Bob, photo) $\iff$ ACL (Bob, photo)
- v observed effects of u $\implies$ v should be delivered after u
- Available: doesn’t slow down sender
Eventual Consistency

- access (Bob, photo) $\implies$ ACL (Bob, photo)
- $v$ observed effects of $u$ $\implies$ $v$ should be delivered after $u$
- Available: doesn’t slow down sender
Eventual Consistency

- access (Bob, photo) ➞ ACL (Bob, photo)
- v observed effects of u ➞ v should be delivered after u
- Available: doesn’t slow down sender
Eventual Consistency

- access (Bob, photo) $\rightarrow$ ACL (Bob, photo)
- $v$ observed effects of $u$ $\rightarrow$ $v$ should be delivered after $u$
- Available: doesn’t slow down sender
COMMUTATIVE REPLICATED DATA TYPES

- Data type
  - Encapsulates state
COMMUTATIVE REPLICATED DATA TYPES

- Data type
  - Encapsulates state
- Replicated
  - At multiple nodes
  - Semantically a single object
COMMUTATIVE REPLICATED DATA TYPES

- Data type
  - Encapsulates state
- Replicated
  - At multiple nodes
  - Semantically a single object
- Available
  - Update origin replica without coordination
  - Convergence guaranteed by design
  - Decentralized
**Commutative Replicated Data Types**

- Data type
  - Encapsulates state
- Replicated
  - At multiple nodes
  - Semantically a single object
- Available
  - Update origin replica without coordination
  - Convergence guaranteed by design
  - Decentralized

OPERATION-BASED CRDTs

- Operation-based CRDTs
  - Each operation is delivered to each replica
Operation-based CRDTs

- Each operation is delivered to each replica
Operation-based CRDTs

- Each operation is delivered to each replica

- Invariant Checking (CISE)

- Requires causal delivery
STATE-BASED CRDTs

- State-based CRDTs
  - Propagation of states (instead of operations)
State-based CRDTs

- Propagation of states (instead of operations)
STATE-BASED CRDTs

- State-based CRDTs
  - Propagation of states (instead of operations)
  - States are merged on receive
  - Convergence: states resulting from concurrent operations result deterministically on a single state
  - No delivery assumptions
STATE-BASED CRDTs
State is a (join semi-)Lattice
STATE-BASED CRDTs

- State is a (join semi-)Lattice
- Effectors send the state at the origin
  - Lazy update propagation
STATE-BASED CRDTs

- State is a (join semi-)Lattice
- Effectors send the state at the origin
  - Lazy update propagation
- `merge` function joins the state of two replicas
  - Join of the lattice
STATE-BASED CRDTs

- State is a (join semi-)Lattice
- Effectors send the state at the origin
  - Lazy update propagation
- `merge` function joins the state of two replicas
  - Join of the lattice
- Each operation is an inflation in the lattice
Bounded Counter

- N Replicas
- R matrix of positive counts
- U vector of negative counts

<table>
<thead>
<tr>
<th></th>
<th>10</th>
<th>2</th>
<th>2</th>
<th>1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

[Valter Balegas, 2016]
**Bounded Counter**

- N Replicas
- R matrix of positive counts
- U vector of negative counts

\[
\text{Total: } \sum_i R[i][i] - \sum_i U[i]
\]

[Valter Balegas, 2016]
Bounded Counter

- N Replicas
- R matrix of positive counts
- U vector of negative counts
- Invariant: $0 \leq \text{Total}$

\[
\text{Total: } \sum_{i} R[i][i] - \sum_{i} U[i]
\]

[Valter Balegas, 2016]
### Bounded Counter

- **N Replicas**
- **R matrix of positive counts**
- **U vector of negative counts**
- Invariant: $0 \leq \text{Total}$

**Increment** @\(i\)

<table>
<thead>
<tr>
<th></th>
<th>(i)</th>
<th>(i)</th>
<th>(j)</th>
<th>(j)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i)</td>
<td>10</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>(i)</td>
<td>3</td>
<td>6</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>(j)</td>
<td>5</td>
<td>0</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>(j)</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

**Total:** \[ \sum_i R[i][i] - \sum_i U[i] \]

[Valter Balegas, 2016]

<table>
<thead>
<tr>
<th>(i)</th>
<th>(j)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Bounded Counter

- N Replicas
- R matrix of positive counts
- U vector of negative counts
- Invariant: $0 \leq \text{Total}$

- increment $\oplus_i$
- decrement $\ominus_i$

Total: $\sum_i R[i][i] - \sum_i U[i] = 13$

[Valter Balegas, 2016]
Bounded Counter

- N Replicas
- R matrix of positive counts
- U vector of negative counts
- Invariant: 0 ≤ Total

**increment** @i

**decrement** @i

\[
\text{Total: } \sum_i R[i][i] - \sum_i U[i]
\]

**Rights @i:**

\[
R[i][i] + \sum_j R[j][i] - \sum_{j \neq i} R[i][j] - U[i]
\]

[Valter Balegas, 2016]
**Bounded Counter**

- N Replicas
- R matrix of positive counts
- U vector of negative counts
- Invariant: $0 \leq \text{Total}$
- **increment** $@i$
- **decrement** $@i$

```
\begin{array}{ccc}
 i & j & \\
 10 & 2 & 2 & 1 \\
 3 & 6 & 1 & 0 \\
 5 & 0 & 4 & 2 \\
 1 & 0 & 2 & 1 \\
 & 5 & 2 & 1 & 0
\end{array}
```

```
\text{Total: } \sum_i R[i][i] - \sum_i U[i] = 13
```

```
\text{Rights }@i: 
R[i][i] + \sum_j R[j][i] - \sum_{j \neq i} R[i][j] - U[i] = 2
```

[Valter Balegas, 2016]
**Bounded Counter**

- N Replicas
- R matrix of positive counts
- U vector of negative counts
- Invariant: $0 \leq \text{Total}$
- Increment $\@i$
- Decrement $\@i$
- Transfer $\@i \rightarrow j$

**Total:** $\sum_i R[i][i] - \sum_i U[i]$

**Rights @i:** $R[i][i] + \sum_j R[j][i] - \sum_{j\neq i} R[i][j] - U[i]$

[Valter Balegas, 2016]
**Bounded Counter**

- N Replicas
- R matrix of positive counts
- U vector of negative counts
- Invariant: $0 \leq \text{Total}$
- **increment** @i
- **decrement** @i
- **transfer** @i $\rightarrow$ j

\[ \text{merge}((M_0, V_0), (M_1, V_1)) = (\max(M_0, M_1), \max(V_0, V_1)) \]

[Valter Balegas, 2016]
CHECKING INVARIANTS

STATE-BASED CRDTs
INvariants for SB-CRDTs

- CRDT (lattice) constraints
INvariants for SB-CRDTs

- CRDT (lattice) constraints
  - Operations are inflations
    \[ \forall \text{op}, \sigma, \sigma', \sigma \models \text{Pre}_{\text{op}} \land (\sigma, \sigma') \in [\text{op}] \Rightarrow \sigma \subseteq \sigma' \]
INvariants for SB-CRDTs

- CRDT (lattice) constraints

- Operations are inflations

\[ \forall \text{ op, } \sigma, \sigma', \; \sigma \in \text{Pre}_{\text{op}} \land (\sigma, \sigma') \in \llbracket \text{op} \rrbracket \Rightarrow \sigma \sqsubseteq \sigma' \]

- merge is join (LUB)

\[ \forall \sigma, \sigma', \; \text{merge}(\sigma, \sigma') = \sigma'' \Rightarrow \sigma'' = \text{LUB}_{\subseteq}(\sigma, \sigma') \]
Invariants for SB-CRDTs

- Invariant constraints
Invariants for SB-CRDTs

- Invariant constraints
  - Operations preserve the invariant

\[ \forall \text{op}, \sigma, \sigma', \sigma \models \text{Pre}_{\text{op}} \land (\sigma, \sigma') \in \left[ \text{op} \right] \Rightarrow \sigma' \models \text{Inv} \]
Invariants for SB-CRDTs

- Invariant constraints

  - Operations preserve the invariant

    \[ \forall \text{op}, \sigma, \sigma', \sigma \models \text{Pre}_{\text{op}} \land (\sigma, \sigma') \in \text{[op]} \Rightarrow \sigma' \models \text{Inv} \]

  - \text{merge} preserves the invariant

    \[ \forall \sigma, \sigma', \sigma'', \quad \text{merge}(\sigma, \sigma'') = \sigma' \Rightarrow \sigma' \models \text{Inv} \]
Invariants for SB-CRDTs

- Invariant constraints
  - Operations preserve the invariant
    \[ \forall \text{op}, \sigma, \sigma', \sigma \models \text{Pre}_{\text{op}} \land (\sigma, \sigma') \in [\text{op}] \Rightarrow \sigma' \models \text{Inv} \]
  - merge preserves the invariant
    \[ \forall \sigma, \sigma', \sigma'', (\sigma, \sigma'') \models \text{Pre}_{\text{merge}} \land \\
    \text{merge}(\sigma, \sigma'') = \sigma' \Rightarrow \sigma' \models \text{Inv} \]
Invariant constraints

Operations preserve the invariant

\[ \forall \text{op}, \sigma, \sigma', \sigma \models \text{Pre}_\text{op} \land (\sigma, \sigma') \in \text{[op]} \Rightarrow \sigma' \models \text{Inv} \]

merge preserves the invariant

\[ \forall \sigma, \sigma', \sigma'', (\sigma, \sigma'') \models \text{Pre}_\text{merge} \land \text{merge}(\sigma, \sigma'') = \sigma' \Rightarrow \sigma' \models \text{Inv} \]

Eg: Bounded Counter

Total(\sigma) \geq 0 \land \text{Total}(\sigma') \geq 0
INVARIANTS FOR SB-CRDTs

- Invariant constraints
INVARINTS FOR SB-CRDTs

- Invariant constraints
  - `merge` can execute at any time
Invariants for SB-CRDTs

- Invariant constraints
  - \texttt{merge} can execute at any time
  - Operations preserve the \texttt{merge} Pre

\[
\forall \text{op}, \sigma, \sigma', \sigma'', \left( \sigma \models \text{Pre}_{\text{op}} \land (\sigma, \sigma'') \models \text{Pre}_{\text{merge}} \land (\sigma, \sigma') \in \left[\text{op}\right] \right) \Rightarrow (\sigma', \sigma'') \models \text{Pre}_{\text{merge}}
\]
\section*{Invariants for SB-CRDTs}

- Invariant constraints
  - \textbf{merge} can execute at any time
  - Operations preserve the \textbf{merge} Pre

\[ \forall \text{op}, \sigma, \sigma', \sigma'', \left( \sigma \models \text{Pre}_{\text{op}} \land (\sigma, \sigma'') \models \text{Pre}_{\text{merge}} \land (\sigma, \sigma') \in \text{[op]} \right) \Rightarrow (\sigma', \sigma'') \models \text{Pre}_{\text{merge}} \]

- \textbf{merge} Pre is an invariant
EXAMPLE: TOKEN IMPLEMENTATION
(Eg.) Mutual Exclusion CRDT
(Eg.) Mutual Exclusion CRDT

- At most one replica has rights to the mutual exclusion token
(Eg.) Mutual Exclusion CRDT

- At most one replica has rights to the mutual exclusion token
  - An array V[R] to indicate who has the “lock”
At most one replica has rights to the mutual exclusion token
- An array $V[R]$ to indicate who has the “lock”
- Escrow
  - Owner transfer the token to the next owner
(E.g.) **Mutual Exclusion CRDT**

- At most one replica has rights to the mutual exclusion token
  - An array $V[R]$ to indicate who has the “lock”
  - Escrow
    - Owner transfer the token to the next owner
- Comparison function?
At most one replica has rights to the mutual exclusion token

- An array \( V[R] \) to indicate who has the “lock”
- Escrow
  - Owner transfer the token to the next owner
- Comparison function?
  - Add a timestamp \( t \)
(Eg.) Mutual Exclusion CRDT

- At most one replica has rights to the mutual exclusion token
  - An array $V[R]$ to indicate who has the “lock”
  - Escrow
    - Owner transfer the token to the next owner
- Comparison function?
  - Add a timestamp $t$

```plaintext
transfer((t,V),ro):
    assert(V[r] = 1 \land (\forall t_o \neq t, t \geq t_o))
    t = t+1
    V[r_s] = 0 \# self
    V[r_o] = 1 \# other

merge((t,V),(t_o,V_o)):
    t = \max(t,t_o)
    v = (t_o < t)?V:V_o
```
\( \text{(Eg.) Mutual Exclusion CRDT} \)

\[
\text{transfer}((t, V), r_o) : \\
\quad \text{assert}(V[r] = 1 \land (\forall t_o \neq t, t \geq t_o)) \\
\quad t = t + 1 \\
\quad V[r_s] = 0 \ # \ self \\
\quad V[r_o] = 1 \ # \ other \\
\]

\[
\text{merge}((t, V), (t_o, V_o)) : \\
\quad t = \max(t, t_o) \\
\quad v = (t_o < t)?V:V_o
\]
(E.g.) Mutual Exclusion CRDT

\[
\begin{align*}
\text{transfer}((t,V), r_o): & \\
& \text{assert}(V[r] = 1 \land (\forall t_o \neq t, t \geq t_o)) \\
& t = t + 1 \\
& V[r_s] = 0 \ # \text{self} \\
& V[r_o] = 1 \ # \text{other}
\end{align*}
\]

\[
\begin{align*}
\text{merge}((t,V),(t_o,V_o)): & \\
& t = \max(t,t_o) \\
& v = (t_o < t)?V:V_o
\end{align*}
\]

- Order:
(Eg.) Mutual Exclusion CRDT

**transfer**((t,V),rₐ):
- assert(V[r] = 1 ∧ (∀tₒ ≠ t, t ≥ tₒ))
- t = t+1
- V[rₛ] = 0  # self
- V[rₒ] = 1  # other

**merge**((t,V),(tₒ,Vₒ)):
- t = max(t,tₒ)
- v = (tₒ < t)?V:Vₒ

- Order: 
  
  \[(t₀, V₀) ≤ (t₁, V₁) = t₀ ≤ t₁ ∧ (t₀ ⇒ t₁ ∧ V₀ ≤ V₁)\]
(Eg.) **Mutual Exclusion CRDT**

### transfer((t,V),r_o):
- `assert(V[r] = 1 ∧ (∀t_o ≠ t, t≥t_o)`
- `t = t+1`
- `V[r_s] = 0 # self`
- `V[r_o] = 1 # other`

### merge((t,V),(t_o,V_o)):
- `t = max(t,t_o)`
- `v = (t_o<t)?V:V_o`

- **Order:**
  
  \[(t_0,V_0) ≤ (t_1,V_1) = t_0 ≤ t_1 ∧ (t_0 ⇒ t_1 ∧ V_0 ≤ V_1)\]

- **Invariant:**
**Mutual Exclusion CRDT**

- **transfer** \(((t,V),r_o)\):
  - assert \((V[r] = 1 \land (\forall t_o \neq t, t \geq t_o))\)
  - \(t = t + 1\)
  - \(V[r_s] = 0\) # self
  - \(V[r_o] = 1\) # other

- **merge** \(((t,V),(t_o,V_o))\):
  - \(t = \max(t,t_o)\)
  - \(v = (t_o < t)?V:V_o\)

- **Order:** \((t_0,V_0) \leq (t_1,V_1) = t_0 \leq t_1 \land (t_0 \Rightarrow t_1 \land V_0 \leq V_1)\)

- **Invariant:** \(\text{Inv}(t,V) = \sum_r V[r] = 1\)
(Eg.) **Mutual Exclusion CRDT**

\[\text{transfer}((t,V),r_o):\]
- \(\text{assert}(V[r] = 1 \land (\forall t_o \neq t, t \geq t_o))\)
- \(t = t + 1\)
- \(V[r_s] = 0 \# \text{self}\)
- \(V[r_o] = 1 \# \text{other}\)

\[\text{merge}((t,V),(t_o,V_o)):\]
- \(t = \max(t,t_o)\)
- \(v = (t_o < t) ? V : V_o\)

- **Order:**
  \((t_0,V_0) \leq (t_1,V_1) = t_0 \leq t_1 \land (t_0 \Rightarrow t_1 \land V_0 \leq V_1)\)

- **Invariant:**
  \(\text{Inv}(t,V) = \sum_r V[r] = 1\)

- **Merge precondition:**
(Eg.) Mutual Exclusion CRDT

`transfer((t, V), r_o):`

assert(V[r] = 1 \land (\forall t_o \neq t, t \geq t_o))

\[ t = t + 1 \]

V[r_s] = 0 # self

V[r_o] = 1 # other

- **Order:**
  \[ (t_0, V_0) \leq (t_1, V_1) = t_0 \leq t_1 \land (t_0 \Rightarrow t_1 \land V_0 \leq V_1) \]

- **Invariant:**
  \[ \text{Inv}(t, V) = \sum_r V[r] = 1 \]

- **Merge precondition:**
  \[ \text{Pre}_{\text{merge}}((t_s, V_s), (t_o, V_o)) = \text{Inv}(t_s, V_s) \land \text{Inv}(t_o, V_o) \land (t_s = t_o \Rightarrow V_s = V_o) \land (V[r_s] = 1 \Rightarrow t_s \geq t_o) \]
TOOL SUPPORT

- Input
  - Definition of Order
  - Definition of merging function
  - Invariant
TOOLS SUPPORT

- Input
  - Definition of Order
  - Definition of merging function
  - Invariant
- Soteria (sister to CISE/CEC)
  - Implemented on top of Boogie
  - Performs the lattice and invariant checks
TOOL SUPPORT

- Input
  - Definition of Order
  - Definition of merging function
  - Invariant
- Soteria (sister to CISE/CEC)
  - Implemented on top of Boogie
  - Performs the lattice and invariant checks
- Work in progress
TOOL SUPPORT

- Input
  
  ```bash
  ~/r/c/c/s/s/soteria (origin/rewriting+) > python3 soteria.py specs/token.spec

  INFO : ************ token ************
  INFO : Checking the syntax
  INFO : Parsing the specification
  INFO : Checking the well-formedness of the specification
  INFO : Checking convergence
  INFO : Checking safety
  INFO : The specification is safe!!!
  ```

  - Implemented on top of Boogie
  - Performs the lattice and invariant checks
  - Work in progress
Research Opportunities

‣ Beyond Simple Invariants
  ‣ Pre/Post conditions of client programs using (1+) CRDTs
  ‣ Transactions + CRDTs

‣ Consistency Models: Eventual, Causal, Strong, …

‣ Synchronization?

‣ …