An introduction to RG-Sep

Viktor Vafeiadis
Example: lock-coupling list

There is one lock per node; threads acquire locks in a hand over hand fashion.

If a node is locked, we can insert a node just after it.

If two adjacent nodes are locked, we can remove the second.
There is one lock per node; threads acquire locks in a hand over hand fashion.

If a node is locked, we can insert a node just after it.

If two adjacent nodes are locked, we can remove the second.
Example: lock-coupling list

There is one lock per node; threads acquire locks in a hand over hand fashion.

If a node is locked, we can insert a node just after it.

If two adjacent nodes are locked, we can remove the second.
There is one lock per node; threads acquire locks in a hand over hand fashion.

If a node is locked, we can insert a node just after it.

If two adjacent nodes are locked, we can remove the second.
Example: lock-coupling list

There is one lock per node; threads acquire locks in a hand over hand fashion.

If a node is locked, we can insert a node just after it.

If two adjacent nodes are locked, we can remove the second.
There is one lock per node; threads acquire locks in a hand over hand fashion.

If a node is locked, we can insert a node just after it.

If two adjacent nodes are locked, we can remove the second.
Example: lock-coupling list

There is one lock per node; threads acquire locks in a hand over hand fashion.

If a node is locked, we can insert a node just after it.

If two adjacent nodes are locked, we can remove the second.
Example: lock-coupling list

There is one lock per node; threads acquire locks in a hand over hand fashion.

If a node is locked, we can insert a node just after it.

If two adjacent nodes are locked, we can remove the second.
Operations (actions)

Lock-coupling list

**Lock**

**Unlock**

**Add node**

**Delete node**
Part I. Basic concepts

- Local & shared state
- Actions
- Program & environment
- Program specifications
- Stability
The total state is logically divided into two components:

- **Shared**: accessible by all threads via synchronisation
- **Local**: accessible only by one thread, its owner

**Figure.** State of the lock-coupling list just before inserting a new node. The node to be added is local because other threads cannot yet access it.
Actions describe minimal atomic changes to the shared state.

Actions (1/3)

Lock

Unlock

An action allows any part of the *shared state* that satisfies the LHS to be changed to a part satisfying the RHS, but the rest of the shared state must not be changed.
Actions (2/3)

Actions can also adjust the boundary between local state and stared state. This is also known as *transfer of ownership*.

**Add node**

This node becomes shared.

**Delete node**

This node becomes local.
Example: Lock coupling list
Actions (3/3)

Example: Lock coupling list

Shared

Local

Add node
Actions (3/3)

Example: Lock coupling list

Shared

2 → 3 → 5 → 7 → 11 → 13

Local

Lock node

12 → 13

[Diagram showing a sequence of actions with a lock symbol.]
Actions (3/3)

Example: Lock coupling list

Shared

2 → 3 → 5 → 7 → 11 → 13

Local

Lock node

→

→
Actions (3/3)

Example: Lock coupling list

Shared

Local

Delete node
Actions (3/3)

Example: Lock coupling list

Shared

Local

Delete node
Now, the node is local; we can safely dispose it.
Now, the node is local; we can safely dispose it.
Program & environment

**Program:** the current thread being verified.

**Environment:** all other threads of the system that execute in parallel with the thread being verified.

**Interference.** The program interferes with the environment by modifying the shared state. Conversely, the environment interferes with the program by modifying the shared state.
The specification of a program consists of two assertions (precondition & postcondition), and two sets of actions:

- **Rely:** Describes the interference that the program can tolerate from the environment; i.e. specifies how the environment can change the shared state.

- **Guarantee:** Describes the interference that the program imposes on its environment; i.e. specifies how the program can change the shared state.
**Definition.** An assertion is stable if and only if it is preserved under interference by other threads.

**Example 1.** The following assertion is not stable.

For instance, another thread could remove node 3 or add a node after node 11.
**Definition.** An assertion is stable if and only if it is preserved under interference by other threads.

**Example 1.** The following assertion is not stable.

For instance, another thread could remove node 3 or add a node after node 11.
**Definition.** An assertion is stable if and only if it is preserved under interference by other threads.

**Example 1.** The following assertion is not stable.

For instance, another thread could remove node 3 or add a node after node 11.
Example 2. The following assertion, however, is stable.
Example 2. The following assertion, however, is stable.

Add node

Delete node
Part II. Program logic

- Syntax & semantics of assertions
- Syntax & semantics of actions
- Syntax & semantics of judgements
- Some proof rules
- Checking stability
Assertion syntax

Separation Logic

\[ P, Q ::= \text{false} \mid \text{emp} \mid e = e' \mid e \leftrightarrow e' \mid \exists x. P \mid P \Rightarrow Q \mid P \ast Q \mid P \ominus Q \]

\[ h \models_{\text{SL}} P \ominus Q \iff h \models_{\text{SL}} \neg (P \ast \neg Q) \iff \exists h'. (h' \models_{\text{SL}} P) \land (h \uplus h' \models_{\text{SL}} Q) \]

Extended logic

\[ p, q ::= P \mid \overline{P} \mid p \ast q \mid p \land q \mid p \lor q \mid \exists x. p \mid \forall x. p \]

local \hspace{1cm} \text{shared}
Assertion semantics

\[ l, s \models P \iff l \models_{SL} P \]
\[ l, s \models \overline{P} \iff l = \emptyset \land (s \models_{SL} P) \]
\[ l, s \models p_1 \ast p_2 \iff \exists l_1, l_2. \ (l = l_1 \cup l_2) \land (l_1, s \models p_1) \land (l_2, s \models p_2) \]

Split local state;
share global state.
Actions

\[ x \mapsto 0, v, t \quad \leadsto \quad x \mapsto \text{tid}, v, t \]
\[ x \mapsto \text{tid}, v, t \quad \leadsto \quad x \mapsto 0, v, t \]
\[ x \mapsto \text{tid}, v, t \quad \leadsto \quad x \mapsto \text{tid}, v, y \]
\[ * y \mapsto 0, v', t \]
\[ x \mapsto \text{tid}, v, y \quad \leadsto \quad x \mapsto \text{tid}, v, t \]
\[ * y \mapsto \text{tid}, v', t \]
Judgements

\[ \vdash C \text{ sat } (p, R, G, q) \]

(precondition, rely, guarantee, postcondition)
Parallel rule

⊢ \( C_1 \) sat \( (p_1, R \cup G_2, G_1, q_1) \)

⊢ \( C_2 \) sat \( (p_2, R \cup G_1, G_2, q_2) \)

\( \vdash (C_1 \parallel C_2) \) sat \( (p_1 \ast p_2, R, G_1 \cup G_2, q_1 \ast q_2) \)

Splits local state;
Shares global state.
Atomic commands

\[ p, q \text{ stable under } R \]
\[ \vdash (\text{atomic } C) \text{ sat } (p, \emptyset, G, q) \]
\[ \vdash (\text{atomic } C) \text{ sat } (p, R, G, q) \]

\[ P_2, Q_2 \text{ precise } \quad (P_2 \rightsquigarrow Q_2) \in G \]
\[ \vdash C \text{ sat } (P_1 \ast P_2, \emptyset, \emptyset, Q_1 \ast Q_2) \]
\[ \vdash (\text{atomic } C) \text{ sat } \left( P_1 \ast \boxed{P_2 \ast F}, \emptyset, G, Q_1 \ast \boxed{Q_2 \ast F} \right) \]
Stability

\[ S \text{ stable under } P \leadsto Q \]

iff

\[ (((P \ominus S) \ast Q) \Rightarrow S) \]

\[ \Rightarrow S \]
The End

Further topics:

- Automation (SmallfootRG)
- Local guards & provided clauses
- Modular reasoning about memory allocators
- Proving linearisability of concurrent algorithms