Deny-Guarantee Reasoning

Intuition
Rely-guarantee is the best approach to reasoning about concurrency. However, it only deals with parallel composition, not fork and join.
We propose deny-guarantee, a new logic that deals naturally with fork and join by dynamically splitting interference.

Fork and Join
Concurrency theorists have mostly dealt with parallel composition.
\[ C_1 \parallel C_2 \]
However, real programs use fork and join.
\[
\begin{align*}
\text{fork}(C_1) &; \text{join}(C_1) \; ; \\
\text{fork}(x := 1) &; \text{join} \; t_1; \\
x &:= 2;
\end{align*}
\]
Start a thread \( C_1 \) with fork, and continue execution. Collect thread with join.

Simple fork-join example:
\[
\begin{align*}
t_1 &:= \text{fork} \; (x := 1); \\
t_2 &:= \text{fork} \; (x := 2); \\
\text{join} \; t_1; \\
x &:= 2;
\end{align*}
\]
Program ensures that \( x=2 \) at termination, but this is difficult to prove.

Proving the Example
Suppose we allow interference to be split and joined.
We start with full permission. Full permission on a particular rewrite means no other thread can do it. Then we split it as follows.
\[
\text{full} \rightarrow A_1 \ast A_2 \ast K
\]
Here \( A_1 \) gives full permission to update \( x \) to 1, \( A_2 \) gives the same permission for \( x \) to 2, and \( K \) is the ‘remainder’ permission.

We split the full permission \( A_1 \) to give permission \( G_1 \), a partial permission to write 1 into \( x \).
\[
A_1 \rightarrow G_1 \ast G_1
\]
Partial permissions mean other threads may be able to do the rewrite.

Then we can prove the program as follows.
\[
\begin{align*}
\{G_1 \ast G_1 \ast G_2 \ast G_2 \ast K\} \\
\{G_1 \ast G_2 \ast G_2 \ast K \ast \text{Thread}(t_1, G_1)\} \\
\{G_1 \ast G_2 \ast K \ast \text{Thread}(t_1, G_1) \ast \text{Thread}(t_2, G_2)\} \\
\text{join} \; t_1; \\
\{G_1 \ast G_1 \ast G_2 \ast K \ast \text{Thread}(t_2, G_2)\} \\
x &:= 2; \\
\{G_1 \ast G_1 \ast G_2 \ast K \ast \text{Thread}(t_2, G_2) \ast x = 2\}
\end{align*}
\]
Post-condition \( x=2 \) is stable because \( G_1 \ast G_1 \ast K \) together give full permission on all actions, except writing 2 into \( x \).
That is, the only permitted interference is writing 2 to \( x \).

The Problem with Rely-guarantee
Rely-guarantee models interference as two relations over states.
• A rely \( R_c \), the interference from the environment
• A guarantee \( G \), the actions permitted for the program

Rely-guarantee rule for parallel composition:
\[
\begin{align*}
R_1, G_1 \vdash (P_1 \ast C_1 \ast Q_1) \\
R_2, G_2 \vdash (P_2 \ast C_2 \ast Q_2) \\
\vdash (P_1 \ast P_2 \ast C_1 \parallel C_2 \ast (Q_1 \land Q_2))
\end{align*}
\]
Note that the interference is statically scoped - the same before and after the parallel composition. This can’t cope with fork-join!

Deny-guarantee
For deny-guarantee, we split interference dynamically.
Deny-guarantee defines unified permissions that combine both the rely and guarantee of

Define a set of permissions \( \text{PermDG} \).
\[
\text{PermDG} = ((\text{guar}) \times (0,1)) \cup ((\text{deny}) \times (0,1)) \cup (0) \cup \{1\}
\]

Permission \( pr \) map actions in \( \text{State} \times \text{State} \) to permissions.
\[
pr : \text{State} \times \text{State} \rightarrow \text{PermDG}
\]

Permissions record interference. Given an action \( a \):
• If \( pr(a) = (\text{guar}, n) \) or 1, program can do action \( a \)
• If \( pr(a) = (\text{guar}, 0) \) or 0, environment can do action \( a \)
• A deny \( pr(a) = (\text{deny}, n) \) records that action \( a \) cannot occur.

Reasoning About Fork and Join
We can define a separation logic star-operator over a \( pr \).

Define a separation logic for programs with fork and join.
\[
P, Q ::= B \mid pr \mid \text{false} \mid \text{Thread}(E, P) \mid P \rightarrow Q \mid P = Q \mid \exists X \cdot P
\]

Assertions define both the state and the permitted interference.
Fork and join rules (simplified).
\[
\begin{align*}
\{P_1 \ast C \ast P_2\} \ast \text{Thread}(x, P_3) &\rightarrow P_4 \\
\{P_1 \ast P_3\} \ast x := C \ast (P_4)
\end{align*}
\]

\[
\{P \ast \text{Thread}(E, P')\} \ast \text{join} \; E \ast \{P \ast P'\}
\]

Deny-guarantee permissions allow us to prove our example.

Reasoning About Fork and Join
We can define a separation logic star-operator over a \( pr \).

Define a separation logic for programs with fork and join.
\[
P, Q ::= B \mid pr \mid \text{false} \mid \text{Thread}(E, P) \mid P \rightarrow Q \mid P = Q \mid \exists X \cdot P
\]

Assertions define both the state and the permitted interference.
Fork and join rules (simplified).
\[
\begin{align*}
\{P_1 \ast C \ast P_2\} \ast \text{Thread}(x, P_3) &\rightarrow P_4 \\
\{P_1 \ast P_3\} \ast x := C \ast (P_4)
\end{align*}
\]

\[
\{P \ast \text{Thread}(E, P')\} \ast \text{join} \; E \ast \{P \ast P'\}
\]

Deny-guarantee permissions allow us to prove our example.

Reasoning About Fork and Join
We can define a separation logic star-operator over a \( pr \).

Define a separation logic for programs with fork and join.
\[
P, Q ::= B \mid pr \mid \text{false} \mid \text{Thread}(E, P) \mid P \rightarrow Q \mid P = Q \mid \exists X \cdot P
\]

Assertions define both the state and the permitted interference.
Fork and join rules (simplified).
\[
\begin{align*}
\{P_1 \ast C \ast P_2\} \ast \text{Thread}(x, P_3) &\rightarrow P_4 \\
\{P_1 \ast P_3\} \ast x := C \ast (P_4)
\end{align*}
\]

\[
\{P \ast \text{Thread}(E, P')\} \ast \text{join} \; E \ast \{P \ast P'\}
\]

Deny-guarantee permissions allow us to prove our example.

Reasoning About Fork and Join
We can define a separation logic star-operator over a \( pr \).

Define a separation logic for programs with fork and join.
\[
P, Q ::= B \mid pr \mid \text{false} \mid \text{Thread}(E, P) \mid P \rightarrow Q \mid P = Q \mid \exists X \cdot P
\]

Assertions define both the state and the permitted interference.
Fork and join rules (simplified).
\[
\begin{align*}
\{P_1 \ast C \ast P_2\} \ast \text{Thread}(x, P_3) &\rightarrow P_4 \\
\{P_1 \ast P_3\} \ast x := C \ast (P_4)
\end{align*}
\]

\[
\{P \ast \text{Thread}(E, P')\} \ast \text{join} \; E \ast \{P \ast P'\}
\]

Deny-guarantee permissions allow us to prove our example.

People
• Mike Dodds, University of Cambridge
• Xinyu Feng, Toyota Technology Institute, Chicago
• Matthew Parkinson, University of Cambridge
• Viktor Vafeiadis, Microsoft Research Cambridge