Further Work On Thin Air Reads

Aarhus Concurrency Workshop

Simon Cooksey, Jon Sherry, Mark Batty
{sjc205,js942,mjb211}@kent.ac.uk

The University of Kent

May 2017

Weak Memory Behaviours

- Executions which don't correspond to strict interleavings of instructions are said to be weak.

```
x = y = 0;
```

T_0	T_1
r_1 = x;	r_2 = y;
y = 1;	x = 1;

- $r_1 = 1 \wedge r_2 = 1$ is allowed by the C11 memory model and observable on hardware.

The Thin Air Problem

```
x = y = 0;
```

T_0	T_1
r_1 = x;	r_2 = y;
y = r_1;	x = r_2;

- Here the result $r_1 = 1 \wedge r_2 = 1$ is allowed by the C11 memory mode, but is not observable on hardware.
- A flaw of current programming language memory models is that they permit behaviours which are **not** observable on hardware, or as a result of optimisations.
- The model allowing this behaviour breaks reasoning about how a program can execute.

The Thin Air Problem

```
x = y = 0;
```

T_0	T_1
r_1 = x;	r_2 = y;
y = r_1;	if (r_2 == 1) {
	x = 1;
	}

- It's more complicated than just ruling out executions with cycles of syntactic data dependency like the previous one
- An execution where $r_1 = 1 \wedge r_2 = 1$ is allowed in the C11 memory model, but is not observable in hardware or as a result of optimisations.

The Thin Air Problem

```
x = y = 0;
```

T_0	T_1
r_1 = x;	r_2 = y;
y = r_1;	if (r_2 == 1) {
	x = 1;
	} else {
	x = 1;
	}

- An execution where $r_1 = 1 \wedge r_2 = 1$ is allowed in C11, and is observable as a result of program optimisation in the compiler and on hardware.

Candidate Solutions

- Kang et. al: A Promising Semantics (POPL '17)
- Jeffrey and Riely: On Thin Air Reads (LICS '16)
- Pichon-Pharabod and Sewell: Bubbly Semantics (POPL '16)

Wish list

- Concise definition
- Easy to execute by hand
- Full Java Coverage
- Full C11 Coverage

Event Structures

```
x = y = 0;
```

T_0

```
r1 = A;
if (r1 == 1)
  B = 1;
```

T_1

```
r2 = B;
if(r2 == 1)
  A = 1;
else
  A = 1;
```

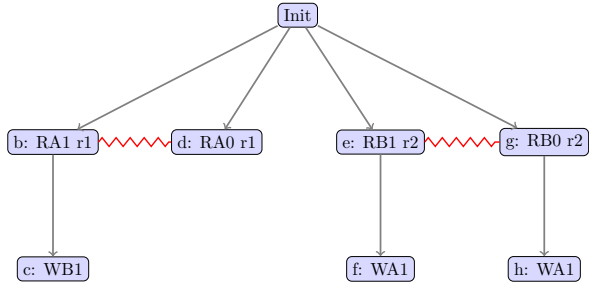


Figure: Java Causality Test Case 6

Conflict

Order

Conflict is symmetric but irreflexive, so not transitive.

Event Structures

```
x = y = 0;
```

T_0

```
r1 = A;
if (r1 == 1)
  B = 1;
```

T_1

```
r2 = B;
if(r2 == 1)
  A = 1;
else
  A = 1;
```

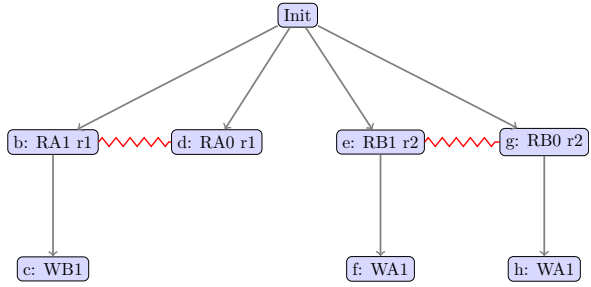


Figure: Java Causality Test Case 6

Conflict

Order

Conflict is symmetric but irreflexive, so not transitive.

Event Structures

```
x = y = 0;
```

T_0

```
r1 = A;
if (r1 == 1)
  B = 1;
```

T_1

```
r2 = B;
if(r2 == 1)
  A = 1;
else
  A = 1;
```

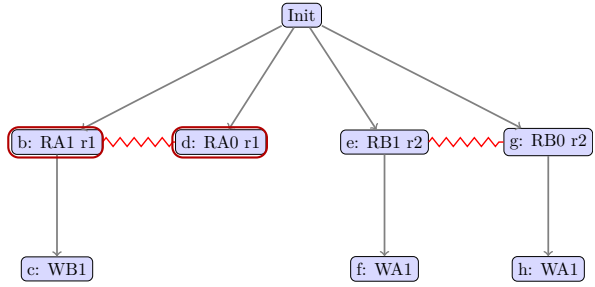


Figure: Java Causality Test Case 6

Conflict

Order

Conflict is symmetric but irreflexive, so not transitive.

Event Structures

```
x = y = 0;
```

T_0

```
r1 = A;
if (r1 == 1)
  B = 1;
```

T_1

```
r2 = B;
if(r2 == 1)
  A = 1;
else
  A = 1;
```

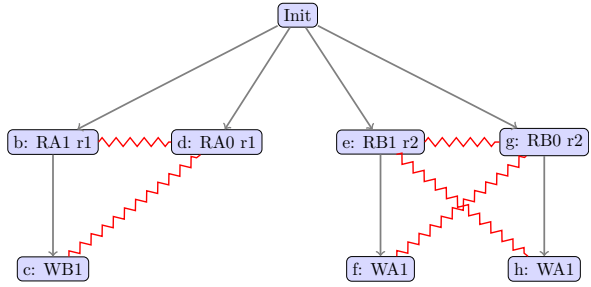


Figure: Java Causality Test Case 6

Conflict

Order

Conflict is symmetric but irreflexive, so not transitive.

Our Choice of Candidate

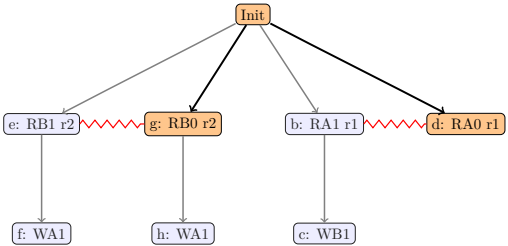
- We've chosen to extend the Jeffrey & Riely model.
- It has a concise definition ready for extra development.
- It has already got a formalisation in AGDA which has proofs for some of the java causality test cases.
- This gives us a good starting point for extension.
- It takes the shape of a 2 player game.

Definitions

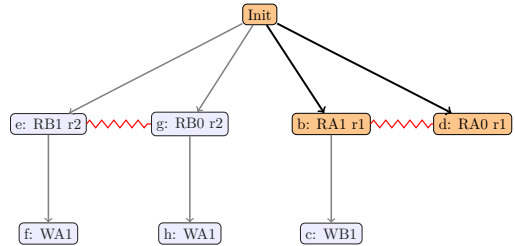
- A configuration is a set of events which is **conflict-free** and **down-closed**

Definitions

- A configuration is a set of events which is **conflict-free** and **down-closed**
- A set is **conflict-free** when no 2 events in a set are in conflict



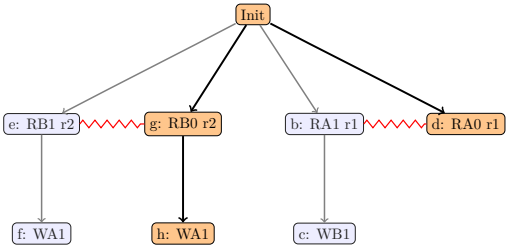
Permitted: $\{Init, g, d\}$



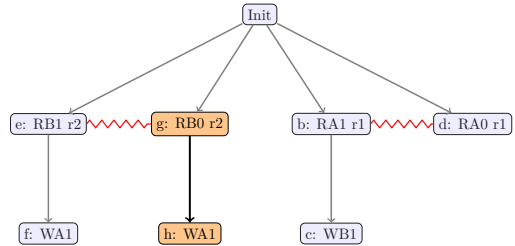
Forbidden: $\{Init, b, d\}$

Definitions

- A configuration is a set of events which is **conflict-free** and **down-closed**
- A set is **conflict-free** when no 2 events in a set are in conflict
- A set is **down-closed** when all the events in a set have their predecessors in the set too



Permitted: $\{Init, g, d, h\}$



Forbidden: $\{g, h\}$

Justification

- An event justifies another when it has the same value and location

- The Init event justifies all reads of 0

Higher Justification

- We can raise justification to a property of a given configuration

$$\text{justified } C \triangleq \forall x \in C.\ \text{is-read } x \implies \exists y \in C.\ y \xrightarrow{\textit{justifies}} x$$

- Or a relation between configurations

$$\text{justifies-config } A\ B \triangleq \forall x \in B.\ \text{is-read } x \implies \exists y \in A.\ y \xrightarrow{\textit{justifies}} x$$

The Game

- The game is a 2 player game with a player and an opponent
- The game is played in rounds
- The aim of the game is for the player to get to a goal configuration

$$\emptyset \sqsubseteq R_1 \sqsubseteq R_2 \sqsubseteq ... \sqsubseteq G$$

- If the player can get to a G then the behaviour is permitted by the model

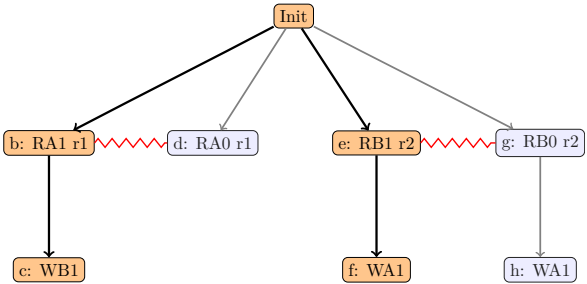
Structure of The Game

- The goal of a round of the game is to get closer to a goal configuration
- The game has a pattern of alternating quantification
- The opponent has the effect of exploring control flow
- The player has the effect of exploring all the valid assumptions left in the environment

1. The round starts in a starting position C
2. The opponent tries to pick super set of C: C' which prevents the player making progress
3. The player attempts to pick super set of C': C'' such that C'' justifies a *Round*.
   - $C \subseteq$ *Round* does not need to be a super set of the intermediate states

Example: Java Causality Test Case 6

Example: Java Causality Test Case 6



T_0

```
r1 = A;
if (r1 == 1)
  B = 1;
```

T_1

```
r2 = B;
if(r2 == 1)
  A = 1;
else
  A = 1;
```

Allowed: $r_1 = 1 \wedge r_2 = 1$.

- The allowed execution would be a configuration of $\{Init, b, c, e, f\}$

Example: Java Causality Test Case 6 – Strategy

$$\emptyset \sqsubseteq R_1 \sqsubseteq R_2 \sqsubseteq ... \sqsubseteq G$$

- We're going to do this in 2 rounds
- We will first show $\emptyset \sqsubseteq \{Init, b, c, g, h\}$
- Then we show $\{Init, b, c, g, h\} \sqsubseteq \{Init, b, c, e, f\}$

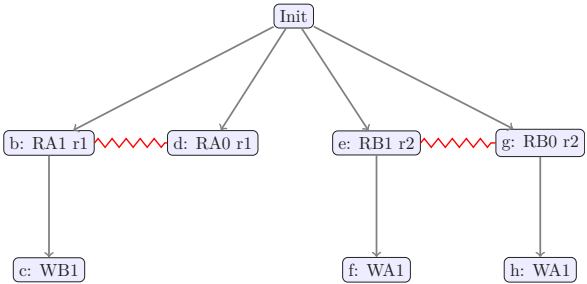
Example: Java Causality Test Case 6 – Strategy

$$\emptyset \sqsubseteq R_1 \sqsubseteq R_2 \sqsubseteq ... \sqsubseteq G$$

- We're going to do this in 2 rounds
- We will first show $\emptyset \sqsubseteq \{Init, b, c, g, h\}$
- Then we show $\{Init, b, c, g, h\} \sqsubseteq \{Init, b, c, e, f\}$

$$\emptyset \sqsubseteq \{Init, b, c, g, h\} \sqsubseteq \{Init, b, c, e, f\}$$

Example: Java Causality Test Case 6

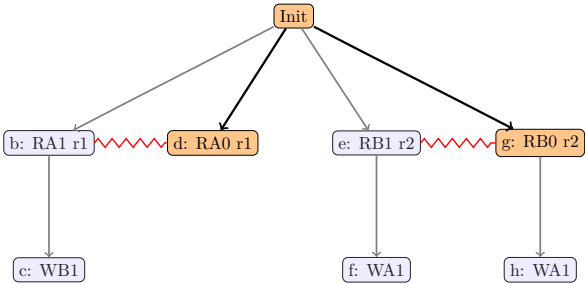


- This is the initial event structure, our $C = \emptyset$

$Goal = \{Init, b, c, e, f\}$

- $C = \emptyset$

Example: Java Causality Test Case 6

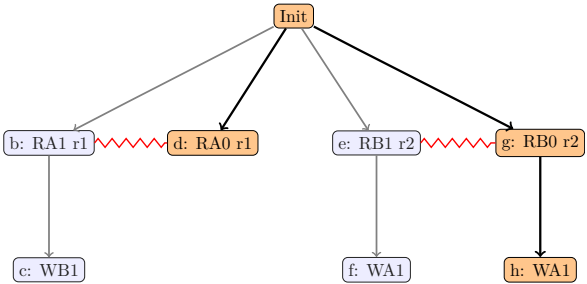


$Goal = \{Init, b, c, e, f\}$

- $C = \emptyset$
- $C' = \{Init, g, d\}$

- The opponent picks C' with as events conflicting with our Goal

Example: Java Causality Test Case 6

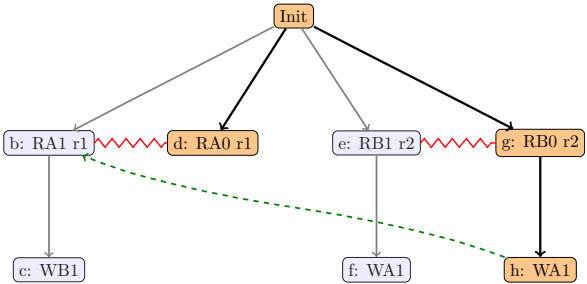


$Goal = \{Init, b, c, e, f\}$

- $C = \emptyset$
- $C' = \{Init, g, d\}$
- $C'' = \{Init, g, d, h\}$

- We extend C' with an extra write to get our C''

Example: Java Causality Test Case 6

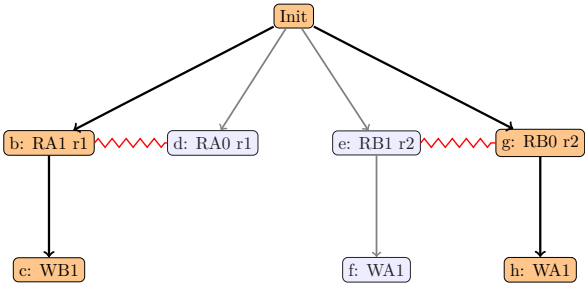


$Goal = \{Init, b, c, e, f\}$

- $C = \emptyset$
- $C' = \{Init, g, d\}$
- $C'' = \{Init, g, d, h\}$

- This extra write in C'' allows us to justify a R_1 with one of our Goal events.

Example: Java Causality Test Case 6

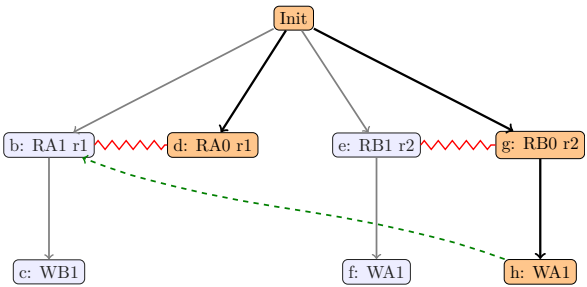


- So we win this round, and we can start the next!
- $\emptyset \sqsubseteq \{Init, b, c, g, h\}$

$Goal = \{Init, b, c, e, f\}$

- $C = \emptyset$
- $C' = \{Init, g, d\}$
- $C'' = \{Init, g, d, h\}$
- $R_1 = \{Init, b, c, g, h\}$

Example: Java Causality Test Case 6 – Intuition



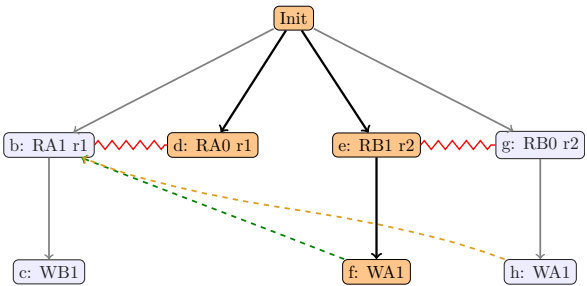
T_0

```
r1 = A;
if (r1 == 1)
  B = 1;
```

T_1

```
r2 = B;
if(r2 == 1)
  A = 1;
else
  A = 1;
```

Example: Java Causality Test Case 6 – Intuition



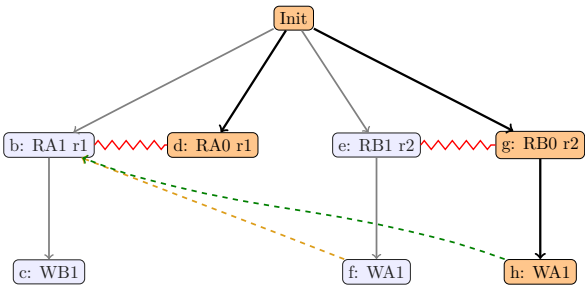
T_0

```
r1 = A;
if (r1 == 1)
  B = 1;
```

T_1

```
r2 = B;
if(r2 == 1)
  A = 1;
else
  A = 1;
```

Example: Java Causality Test Case 6 – Intuition



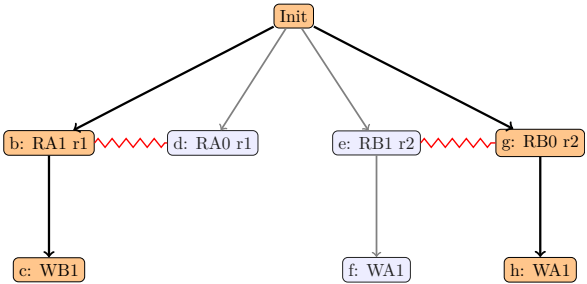
T_0

```
r1 = A;
if (r1 == 1)
  B = 1;
```

T_1

```
r2 = B;
if(r2 == 1)
  A = 1;
else
  A = 1;
```

Example: Java Causality Test Case 6

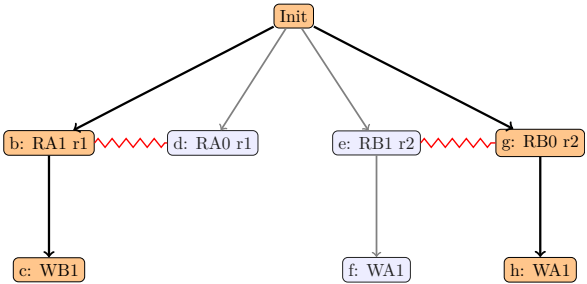


$Goal = \{Init, b, c, e, f\}$

- $C = \emptyset$
- $C' = \{Init, g, d\}$
- $C'' = \{Init, g, d, h\}$
- $R_1 = \{Init, b, c, g, h\}$

- This is the initial event structure, our $C = \emptyset$ The opponent picks C' with as events conflicting with our Goal We extend C' with an extra write to get our C'' This extra write in C'' allows us to justify a R_1 with one of our Goal events. So we win this round, and we can start the next!

Example: Java Causality Test Case 6



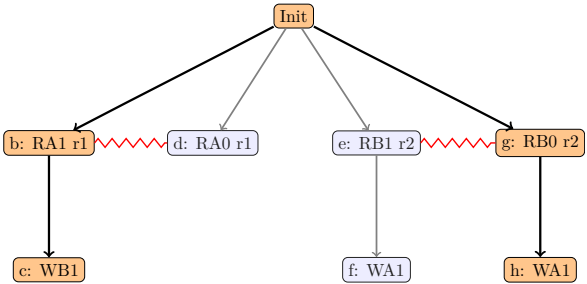
$Goal = \{Init, b, c, e, f\}$

- $C = \emptyset$
- $C' = \{Init, g, d\}$
- $C'' = \{Init, g, d, h\}$
- $R_1 = \{Init, b, c, g, h\}$

- $C = \{Init, b, c, g, h\}$

- We start with C as the previous R_1

Example: Java Causality Test Case 6



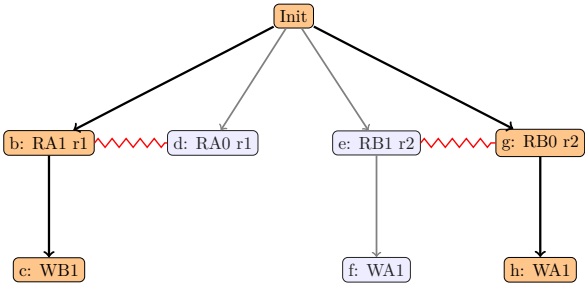
$Goal = \{Init, b, c, e, f\}$

- $C = \emptyset$
- $C' = \{Init, g, d\}$
- $C'' = \{Init, g, d, h\}$
- $R_1 = \{Init, b, c, g, h\}$

- $C = \{Init, b, c, g, h\}$
- $C' = \{Init, b, c, g, h\}$

- The opponent can't add anything to C' that isn't in conflict.

Example: Java Causality Test Case 6



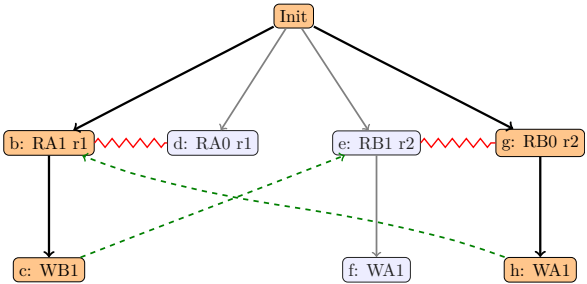
$Goal = \{Init, b, c, e, f\}$

- $C = \emptyset$
- $C' = \{Init, g, d\}$
- $C'' = \{Init, g, d, h\}$
- $R_1 = \{Init, b, c, g, h\}$

- $C = \{Init, b, c, g, h\}$
- $C' = \{Init, b, c, g, h\}$
- $C'' = \{Init, b, c, g, h\}$

- We're on a good track now, we can't add anything but we can win.

Example: Java Causality Test Case 6



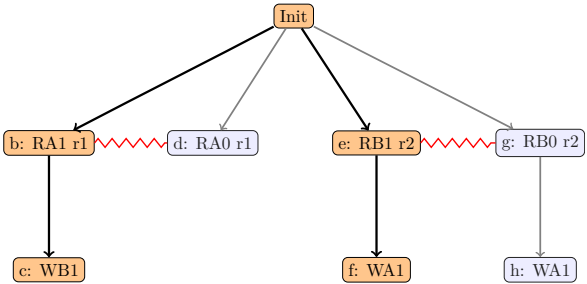
- We can justify e with c, and b with h

$Goal = \{Init, b, c, e, f\}$

- $C = \emptyset$
- $C' = \{Init, g, d\}$
- $C'' = \{Init, g, d, h\}$
- $R_1 = \{Init, b, c, g, h\}$

- $C = \{Init, b, c, g, h\}$
- $C' = \{Init, b, c, g, h\}$
- $C'' = \{Init, b, c, g, h\}$
- $G = \{Init, b, c, e, f\}$

Example: Java Causality Test Case 6



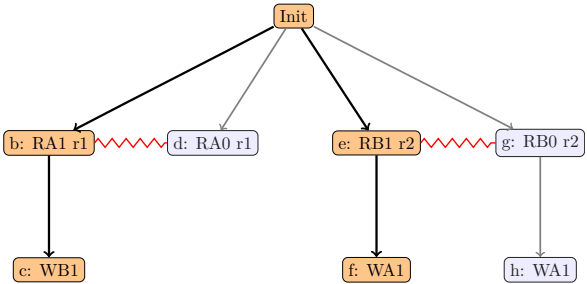
$Goal = \{Init, b, c, e, f\}$

- $C = \emptyset$
- $C' = \{Init, g, d\}$
- $C'' = \{Init, g, d, h\}$
- $R_1 = \{Init, b, c, g, h\}$

- $C = \{Init, b, c, g, h\}$
- $C' = \{Init, b, c, g, h\}$
- $C'' = \{Init, b, c, g, h\}$
- $G = \{Init, b, c, e, f\}$

- This lets us pick our final D configuration as the Goal configuration and we win the game!

Example: Java Causality Test Case 6



$Goal = \{Init, b, c, e, f\}$

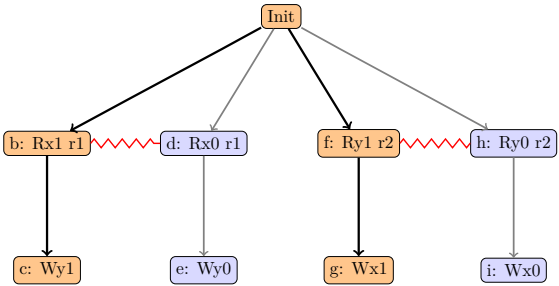
- $C = \emptyset$
- $C' = \{Init, g, d\}$
- $C'' = \{Init, g, d, h\}$
- $R_1 = \{Init, b, c, g, h\}$

- $C = \{Init, b, c, g, h\}$
- $C' = \{Init, b, c, g, h\}$
- $C'' = \{Init, b, c, g, h\}$
- $G = \{Init, b, c, e, f\}$

- This lets us pick our final D configuration as the Goal configuration and we win the game!
- $\emptyset \sqsubseteq \{Init, b, c\} \sqsubseteq \{Init, b, c, e, f\}$

Example: Java Causality Test Case 4

Example: Java Causality Test Case 4

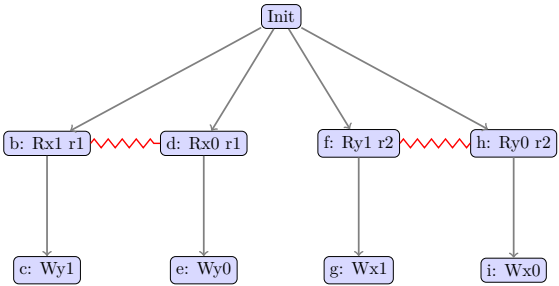


T_0	T_1
`r1 = x;`	`r2 = y;`
`y = r1;`	`x = r2;`

Forbidden: $r_1 = 1 \wedge r_2 = 1$.

- The forbidden execution would be a configuration of $\{Init, b, c, f, g\}$

Example: Java Causality Test Case 4

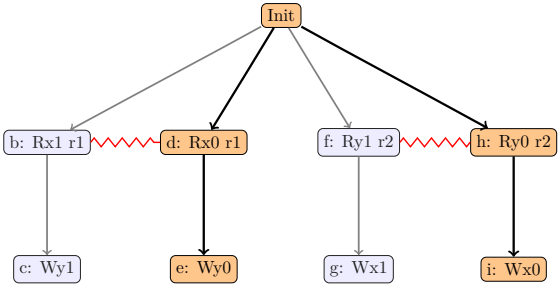


$Goal = \{Init, b, c, f, g\}$

- $C = \emptyset$

- This is the initial event structure, our $C = \emptyset$

Example: Java Causality Test Case 4

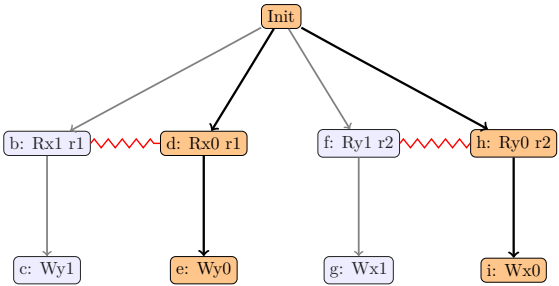


$Goal = \{Init, b, c, f, g\}$

- $C = \emptyset$
- $C' = \{Init, d, e, f, g\}$

- The opponent can pick a configuration C' which is blocking

Example: Java Causality Test Case 4

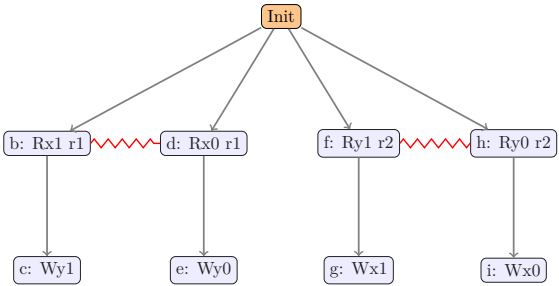


$Goal = \{Init, b, c, f, g\}$

- $C = \emptyset$
- $C' = \{Init, d, e, f, g\}$
- $C'' = \{Init, d, e, f, g\}$

- The player can't add anything to extend C''

Example: Java Causality Test Case 4

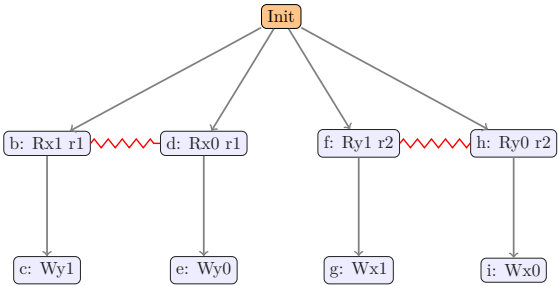


$Goal = \{Init, b, c, f, g\}$

- $C = \emptyset$
- $C' = \{Init, d, e, f, g\}$
- $C'' = \{Init, d, e, f, g\}$
- $D = \{Init\}$

- The largest D the player can pick to get to the goal is $\{Init\}$

Example: Java Causality Test Case 4



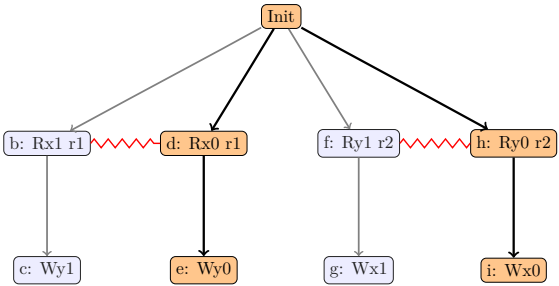
$Goal = \{Init, b, c, f, g\}$

- $C = \emptyset$
- $C' = \{Init, d, e, f, g\}$
- $C'' = \{Init, d, e, f, g\}$
- $D = \{Init\}$

- $C = \{Init\}$

- So the game starts again with $D = \{Init\}$

Example: Java Causality Test Case 4



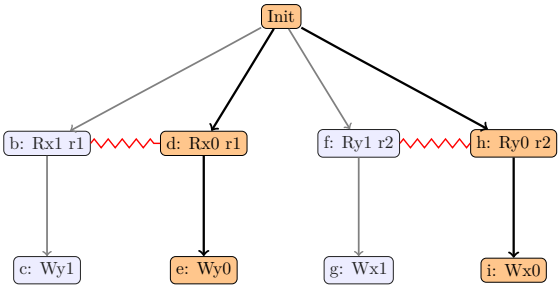
$Goal = \{Init, b, c, f, g\}$

- $C = \emptyset$
- $C' = \{Init, d, e, f, g\}$
- $C'' = \{Init, d, e, f, g\}$
- $D = \{Init\}$

- $C = \{Init\}$
- $C' = \{Init, d, e, f, g\}$

- The opponent again picks this *blocking configuration*

Example: Java Causality Test Case 4



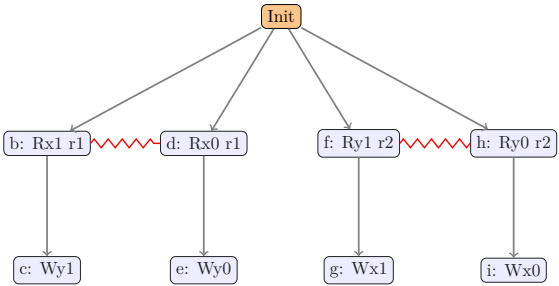
$Goal = \{Init, b, c, f, g\}$

- $C = \emptyset$
- $C' = \{Init, d, e, f, g\}$
- $C'' = \{Init, d, e, f, g\}$
- $D = \{Init\}$

- $C = \{Init\}$
- $C' = \{Init, d, e, f, g\}$
- $C'' = \{Init, d, e, f, g\}$

- The player can't add anything to expand it

Example: Java Causality Test Case 4



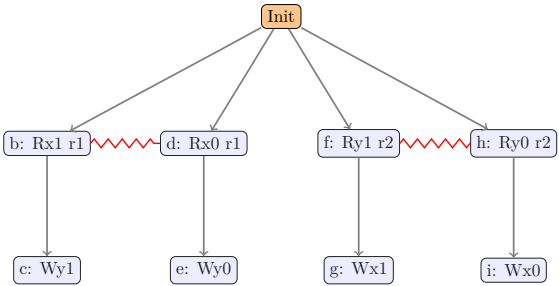
$Goal = \{Init, b, c, f, g\}$

- $C = \emptyset$
- $C' = \{Init, d, e, f, g\}$
- $C'' = \{Init, d, e, f, g\}$
- $D = \{Init\}$

- $C = \{Init\}$
- $C' = \{Init, d, e, f, g\}$
- $C'' = \{Init, d, e, f, g\}$
- $D = \{Init\}$

- So the final D is still $\{Init\}$

Example: Java Causality Test Case 4



$Goal = \{Init, b, c, f, g\}$

- $C = \emptyset$
- $C' = \{Init, d, e, f, g\}$
- $C'' = \{Init, d, e, f, g\}$
- $D = \{Init\}$

- $C = \{Init\}$
- $C' = \{Init, d, e, f, g\}$
- $C'' = \{Init, d, e, f, g\}$
- $D = \{Init\}$

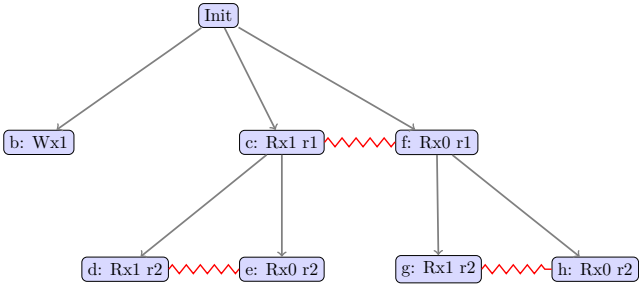
- So the final D is still $\{Init\}$
- The game diverges and the player can't win.

Coherence

Coherence

- This model doesn't enforce coherence
- It also doesn't pass all the Java Causality Test Cases
- Interestingly coherence isn't actually why

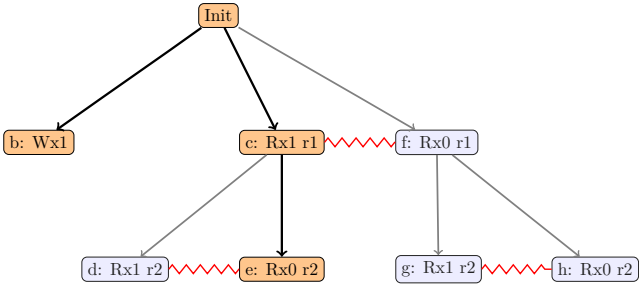
Coherence



T_0	T_1
`x = 1;`	r_1 `= x;`
	r_2 `= x;`

Incoherent: $r_1 = 1 \wedge r_2 = 0$.

Coherence

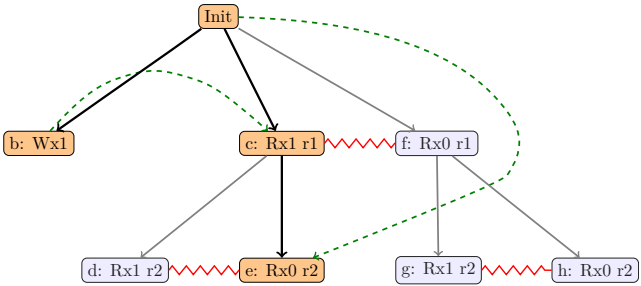


T_0	T_1
`x = 1;`	r_1 `= x;`
	r_2 `= x;`

Incoherent: $r_1 = 1 \wedge r_2 = 0$.

- The incoherent execution would be a configuration of $\{Init, b, c, e\}$

Coherence

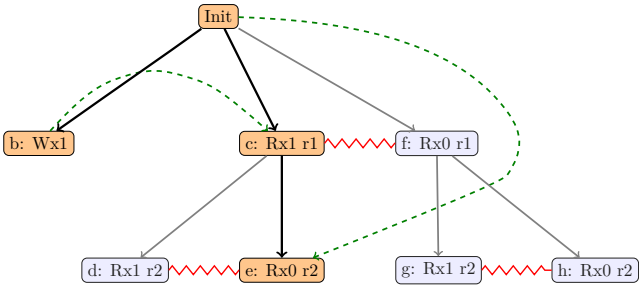


T_0	T_1
`x = 1;`	r_1 `= x;`
	r_2 `= x;`

Incoherent: $r_1 = 1 \wedge r_2 = 0$.

- The incoherent execution would be a configuration of $\{Init, b, c, e\}$
- There's no move the opponent can make to make the relevant writes unavailable to the player to justify this configuration.

Coherence



T_0	T_1
`x = 1;`	r_1 `= x;`
	r_2 `= x;`

Incoherent: $r_1 = 1 \wedge r_2 = 0$.

- The incoherent execution would be a configuration of $\{Init, b, c, e\}$
- There's no move the opponent can make to make the relevant writes unavailable to the player to justify this configuration.
- The incoherent execution is allowed by this model.

Coherence Conjecture

- It may be adequate to enforce coherence *ad hoc*
- We'd do this by adding a constraint at the end of the game's execution

$$\text{well-justified } C \triangleq \emptyset \sqsubseteq^* C \wedge \text{justified } C$$

Coherence Conjecture

- It may be adequate to enforce coherence *ad hoc*
- We'd do this by adding a constraint at the end of the game's execution

$$\text{well-justified } C \triangleq \emptyset \sqsubseteq^* C \wedge \text{justified } C \wedge \text{coherent } C$$

Coherence Conjecture

- It may be adequate to enforce coherence *ad hoc*
- We'd do this by adding a constraint at the end of the game's execution

$$\text{well-justified } C \triangleq \emptyset \sqsubseteq^* C \wedge \text{justified } C \wedge \text{coherent } C$$

- The coherence restriction could take the shape of a C11 memory model axiom
- The axiom forbids the "bad shape" of incoherent execution

$$\text{coherent } C \triangleq \forall x, y \in (C \cap W \cap \text{same-location}).\ \neg((\text{justifies}^{-1})^{=} \circ \text{order} \circ (\text{justifies}^{-1})^{=}\ x\ y)$$

Coherence Conjecture

- It may be adequate to enforce coherence *ad hoc*
- We'd do this by adding a constraint at the end of the game's execution

$$\text{well-justified } C \triangleq \emptyset \sqsubseteq^* C \wedge \text{justified } C \wedge \text{coherent } C$$

- The coherence restriction could take the shape of a C11 memory model axiom
- The axiom forbids the "bad shape" of incoherent execution

$$\text{coherent } C \triangleq \forall x, y \in (C \cap W \cap \text{same-location}).\ \neg((\text{justifies}^{-1})^{=} \circ \text{order} \circ (\text{justifies}^{-1})^{=}\ x\ y)$$

- It's not clear that an *ad hoc* application of this rule is quite enough.

Limitations

- This model currently enforce coherence, so accepts incoherent executions
- The model doesn't currently target all of C11's features
- It doesn't model java fully yet either
- The model isn't compositional

Direction From Here

- We are replicating the causality test case proofs
- An extension to fully cover all 20 java causality test cases
- Extending the model to support full C11 semantics
- Making the model executable using a SAT solver like QBF
  - We looked at Alloy, but it didn't work out
- We have a tool in OCaml for interpreting programs into event structures
  - There are back-ends for the test alloy model, and for Isabelle/HOL
  - A QBF back-end seems like a good thing to write

Options for C11 extensions

- Adding axioms to the shape of configurations (like conflict-free)
- Axiomatic restrictions after the game completes
  - Filtering the valid pre-executions going into the C11 model should eliminate the thin air problem
- We need support for more types of atomic and non-atomics.
- A denotation of this memory model which allows composition.

Conclusion

- We've discussed how Jeffrey et al. have a model which solves the thin air problem
- We've presented the problems with the model
- We've positioned our direction for extending the model to support all of Java's causality test cases, and C11's semantics.

Cheers!

Questions, comments, suggestions?

- We've discussed how Jeffrey et al. have a model which solves the thin air problem
- We've presented the problems with the model
- We've positioned our direction for extending the model to support all of Java's causality test cases, and C11's semantics.

Thanks to collaborators, and to Alan Jeffrey and James Riely. This work has been partly funded by the EPSRC under grant number EP/M508068/1.

Appendix A: Event Structure Constructors

$$
\begin{aligned}
E &= E_1 \cup E_2 \\
\leq &= \leq_1 \cup \leq_2 \\
\# &= \#_1 \cup \#_2 \\
\lambda &= \lambda_1 \cup \lambda_2
\end{aligned}
$$

(Product ($\times$))

$$
\begin{aligned}
E &= E_1 \cup E_2 \\
\leq &= \leq_1 \cup \leq_2 \\
\# &= \#_1 \cup \#_2 \cup (E_1 \times E_2) \cup (E_2 \times E_1) \\
\lambda &= \lambda_1 \cup \lambda_2
\end{aligned}
$$

(Sum ($+$))

$$
\begin{aligned}
E &= E_0 \cup \{\bot\} \\
\leq &= \leq_0 \cup (\{\bot\} \times E) \\
\# &= \#_0 \\
\lambda &= \lambda_0 \cup \{(\bot, \sigma)\}
\end{aligned}
$$

(Composition ($\bullet$))

Appendix B: Program Interpretation

$$\mathscr{T}[\![r = x; T]\!]\rho \triangleq \Sigma_{v \in V}(\mathrm{R}\ x\ v) \bullet \mathscr{T}[\![T]\!](\rho[r \mapsto v]) \qquad \text{(Read)}$$

$$\mathscr{T}[\![x = M; T]\!]\rho \triangleq (\mathrm{W}\ x\ \mathscr{M}[\![M]\!]\rho) \bullet \mathscr{T}[\![T]\!](\rho[r \mapsto v]) \qquad \text{(Write)}$$

$$\mathscr{T}[\![\texttt{done}]\!]\rho \triangleq 0 \qquad \text{(Done)}$$

$$\mathscr{T}[\![\texttt{if}\ M\ T_1\ \texttt{else}\ T_0]\!]\rho \triangleq \begin{cases} \mathscr{T}[\![T_0]\!]\rho & \text{if} \mathscr{M}[\![M]\!]\rho = 0 \\ \mathscr{T}[\![T_1]\!]\rho & \text{otherwise} \end{cases} \qquad \text{(If - Then - Else)}$$

$$\mathscr{P}[\![T_1||...||T_n]\!] \triangleq \mathrm{init} \bullet (\mathscr{T}[\![T_1]\!]\rho_0 \times ... \times \mathscr{T}[\![T_n]\!]\rho_0) \qquad \text{(Program)}$$

Appendix C: Game Definitions

$$\text{conflict-free } C \triangleq \forall x, y. x \in C \wedge y \in C \implies \neg(\text{conflict } x\ y)$$

$$\text{down-closed } C \triangleq \forall x, y. x \in C \wedge y \preceq x \implies y \in C$$

$$\text{justifies-config } C\ D \triangleq \forall x \in D.\text{is-read } x \implies (\exists y \in C.\text{justifies-event } (\mathit{label}\ y)\ (\mathit{label}\ x))$$

$$C \lesssim D \triangleq \text{justifies-config } C\ D \wedge C \subseteq D$$

$$\text{ae-justifies } C\ D \triangleq \forall C'. C \lesssim^* C' \implies \exists C''. C' \lesssim^* C'' \wedge \text{justifies-config } C''\ D$$

$$C \sqsubseteq D \triangleq \text{ae-justifies } C\ D \wedge C \subseteq D$$

$$\text{well-justified } C \triangleq \emptyset \sqsubseteq^* C \wedge \text{justified } C$$