EVENT STRUCTURES AND REFINEMENT FOR RELAXED MEMORY

Alan Jeffrey (Bell Labs)
Joint work with James Riely (DePaul U.)

Memory Model Meeting, Cambridge, September 2014

OVERVIEW

Context

Event structures

Thin air reads

Synchronization actions

Program refinement

To Do List

Conclusions

CONTEXT

import usual.chit.chat.JMM;

CONTEXT

import usual.chit.chat.JMM;

State of the art:

- Formal models of valid executions: JMM, C11, ARM/POWER, x86-TSO, … (Mile high view: partially ordered events labelled with R/W actions)
- DRF theorem.
- Lots of lovely tooling (mechanized models, theorems, test cases, …)

CONTEXT

import usual.chit.chat.JMM;

State of the art:

- Formal models of valid executions: JMM, C11, ARM/POWER, x86-TSO, … (Mile high view: partially ordered events labelled with R/W actions)
- DRF theorem.
- Lots of lovely tooling (mechanized models, theorems, test cases, …)

Problems with state of the art:

- Thin Air Read (TAR): C11 undefined behaviour, JMM complexity.
- Compiler optimizations not validated against model.

CONTEXT

import usual.chit.chat.JMM;

State of the art:

- Formal models of valid executions: JMM, C11, ARM/POWER, x86-TSO, … (Mile high view: partially ordered events labelled with R/W actions)
- DRF theorem.
- Lots of lovely tooling (mechanized models, theorems, test cases, …)

Problems with state of the art:

- Thin Air Read (TAR): C11 undefined behaviour, JMM complexity.
- Compiler optimizations not validated against model.

Can we use existing models of relaxed concurrency?

EVENT STRUCTURES

Event structures (Winskel 1980s) are a model of relaxed concurrency.

EVENT STRUCTURES

Event structures (Winskel 1980s) are a model of relaxed concurrency.

Fix an alphabet of actions Σ (e.g. read, write, init…)

A labelled partial order $(E, \leq, \lambda)$ consists of:

- A partial order $(E, \leq)$ (events with program order)
- A function $\lambda : E \to \Sigma$ (labelling)

EVENT STRUCTURES

Event structures (Winskel 1980s) are a model of relaxed concurrency.

Fix an alphabet of actions Σ (e.g. read, write, init…)

A labelled prime event structure $(E, \leq, \#, \lambda)$ consists of:

- A partial order $(E, \leq)$ (events with program order)
- A function $\lambda : E \to \Sigma$ (labelling)
- A binary relation $\#$ on E (conflict)
- If $d \# e$ then $d \neq e$, and if $c \# d \leq e$ then $c \# e$

EVENT STRUCTURES

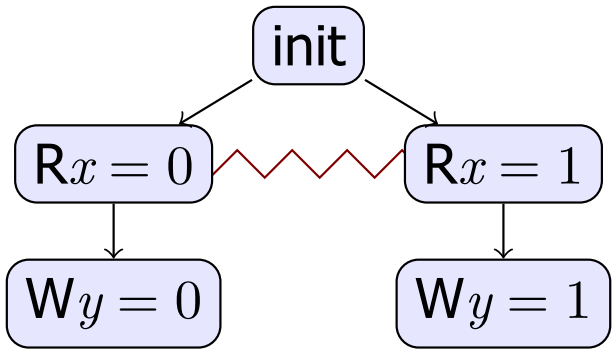
Event structures (Winskel 1980s) are a model of relaxed concurrency.

Fix an alphabet of actions Σ (e.g. read, write, init…)

A labelled prime event structure $(E, \leq, \#, \lambda)$ consists of:

- A partial order $(E, \leq)$ (events with program order)
- A function $\lambda : E \to \Sigma$ (labelling)
- A binary relation $\#$ on E (conflict)
- If $d \# e$ then $d \neq e$, and if $c \# d \leq e$ then $c \# e$

For example the event structure for `r=x; y=r` is:

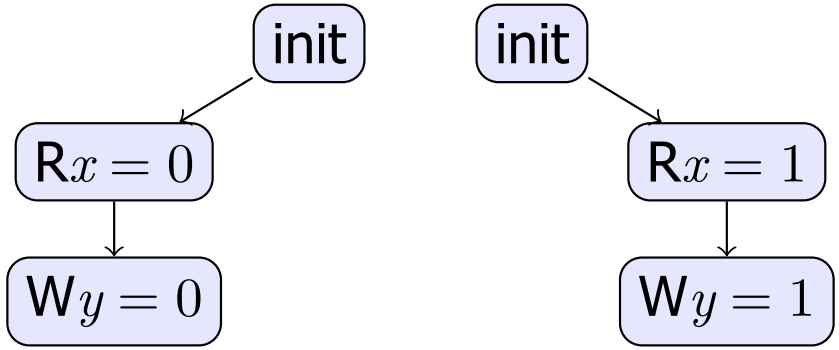


EVENT STRUCTURES

A **configuration** is a $\leq$-downclosed, #-free set of events.

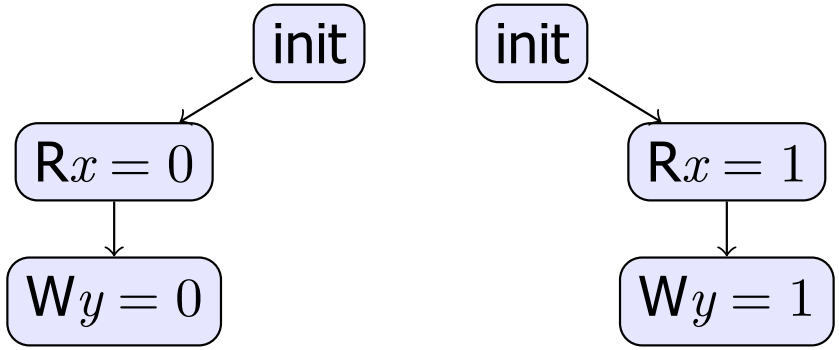
EVENT STRUCTURES

A **configuration** is a $\leq$-downclosed, $\#$-free set of events. For example:



EVENT STRUCTURES

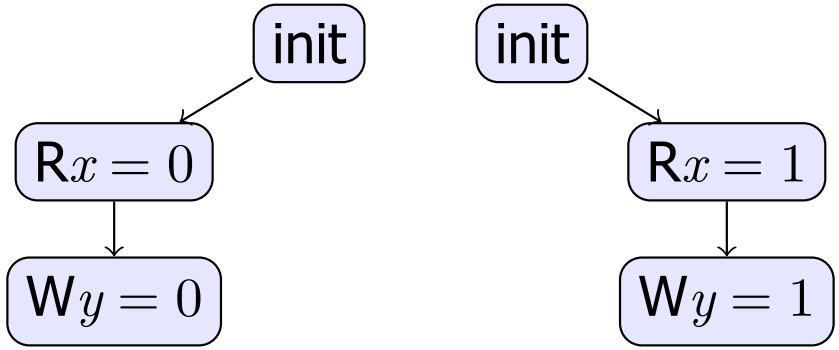
A **configuration** is a $\leq$-downclosed, $\#$-free set of events. For example:



First configuration is fine, but second is fishy, where did $x = 1$ come from?

EVENT STRUCTURES

A **configuration** is a $\leq$-downclosed, $\#$-free set of events. For example:

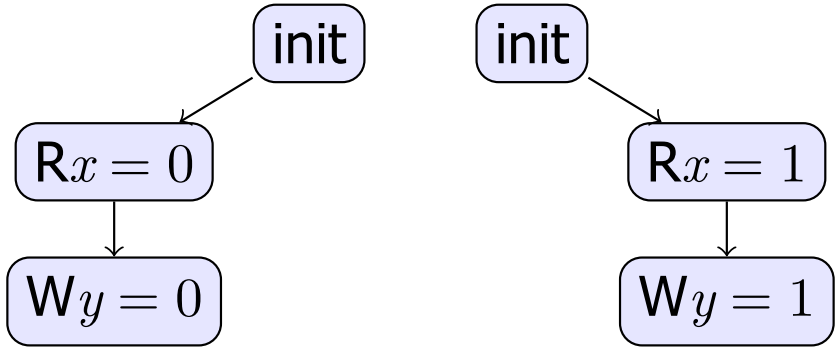


First configuration is fine, but second is fishy, where did $x = 1$ come from?

Assume relations RWJ $\subseteq$ RWC on Σ.
(E.g. RWC is r/w same location, RWJ is r/w same location+value).

EVENT STRUCTURES

A **configuration** is a $\leq$-downclosed, $\#$-free set of events. For example:



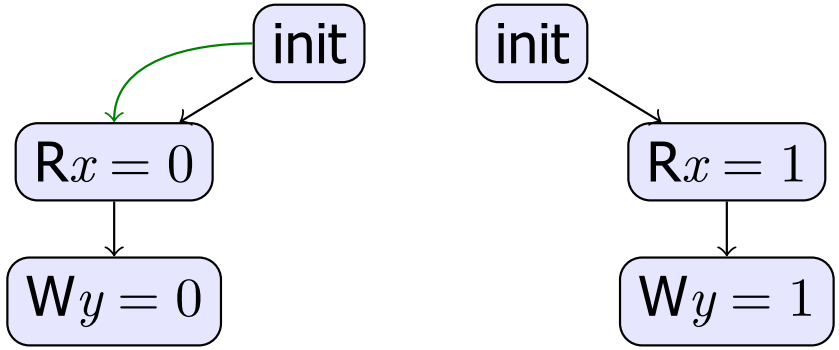
First configuration is fine, but second is fishy, where did $x = 1$ come from?

Assume relations RWJ $\subseteq$ RWC on Σ.
(E.g. RWC is r/w same location, RWJ is r/w same location+value).

Write $d \in \mathrm{RWC}(e)$ for $\lambda(d) \in \mathrm{RWC}(\lambda(e))$, $\neg(d = e)$ and $\neg(d \# e)$. Ditto RWJ.

EVENT STRUCTURES

A **configuration** is a $\leq$-downclosed, $\#$-free set of events. For example:



First configuration is fine, but second is fishy, where did $x = 1$ come from?

Assume relations RWJ $\subseteq$ RWC on Σ.
(E.g. RWC is r/w same location, RWJ is r/w same location+value).

Write $d \in \mathrm{RWC}(e)$ for $\lambda(d) \in \mathrm{RWC}(\lambda(e))$, $\neg(d = e)$ and $\neg(d \# e)$. Ditto RWJ.

Define d is a **justifier** for e when $e \not\leq d$, $d \in \mathrm{RWJ}(e)$, and
there is no $d \leq c \leq e$ where $c \in \mathrm{RWC}(e)$.

A configuration is **justified** if all non-initial events have a justifier.

EVENT STRUCTURES

Event structures are a relaxed model, but some configurations correspond to sequential executions…

EVENT STRUCTURES

Event structures are a relaxed model, but some configurations correspond to sequential executions…

A configuration is **totally ordered** when there is a total order $\leq_{to}$ such that $d \leq e$ implies $d \leq_{to} e$.

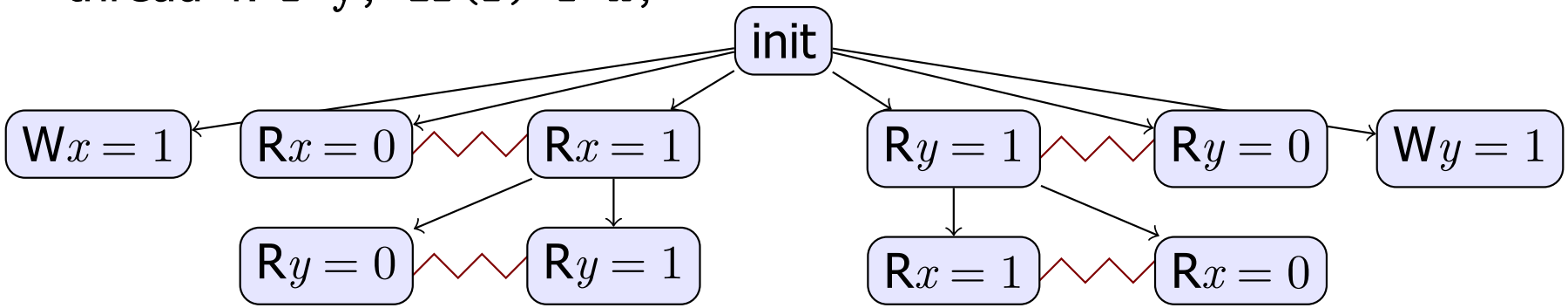
Define d is a **sequential justifier** for e when $d \leq_{to} e$, $d \in \text{RWJ}(e)$, and there is no $d \leq_{to} c \leq_{to} e$ where $c \in \text{RWC}(e)$.

A configuration is **sequentially consistent** if it can be totally ordered such that all non-initial events have a sequential justifier.

EVENT STRUCTURES

An IRIW example:

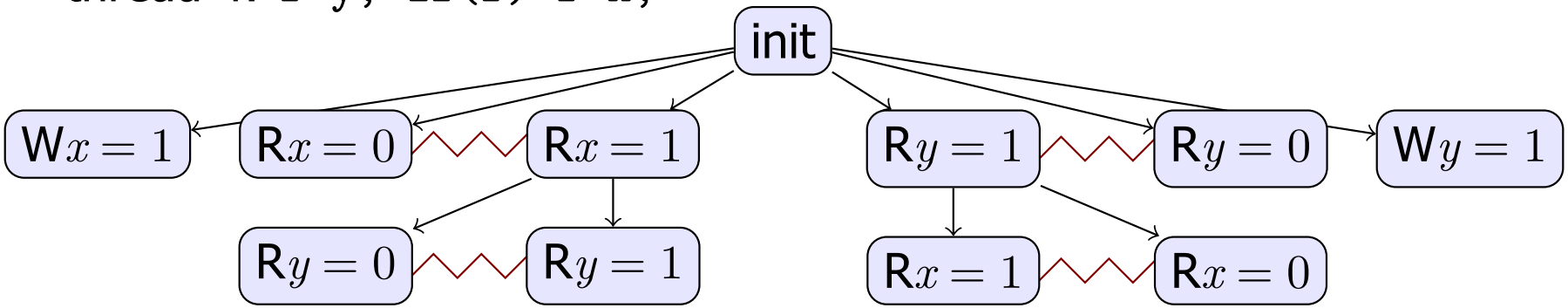
thread 1: `x=1;`
thread 2: `y=1;`
thread 3: `r=x; if(r) r=y;`
thread 4: `r=y; if(r) r=x;`



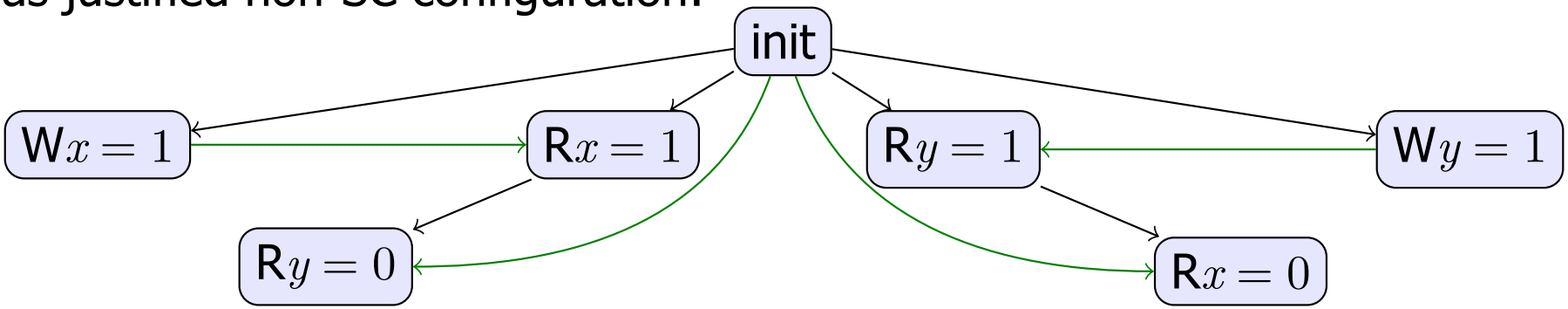
EVENT STRUCTURES

An IRIW example:

thread 1: `x=1;`
thread 2: `y=1;`
thread 3: `r=x; if(r) r=y;`
thread 4: `r=y; if(r) r=x;`



Has justified non-SC configuration:



EVENT STRUCTURES

The basic model is event structures with justified configurations.

Elegant model with pretty pictures (thanks Glynn!).

EVENT STRUCTURES

The basic model is event structures with justified configurations.

Elegant model with pretty pictures (thanks Glynn!).

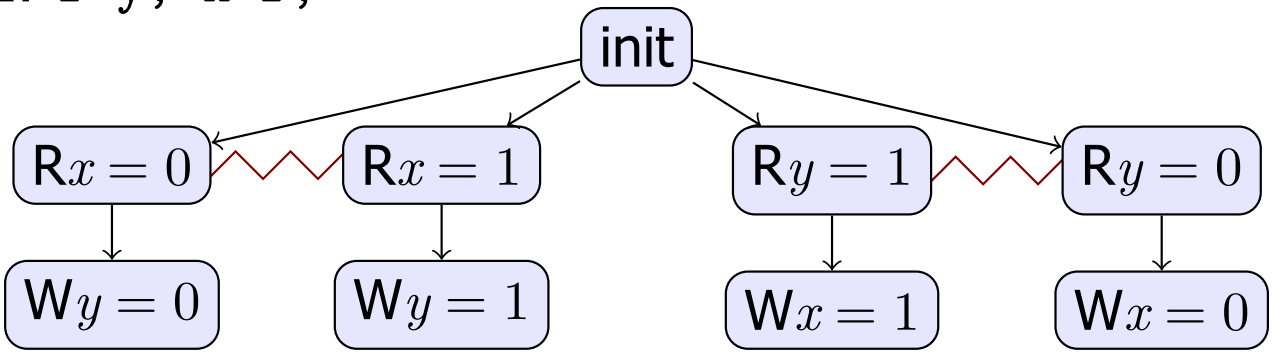
Basic model doesn't do everything:

- Thin Air Reads.
- Synchronization actions (volatile fields, locks, …).
- Refinement relation to validate compiler optimizations.

THIN AIR READS

Oh dear, the TAR pit:

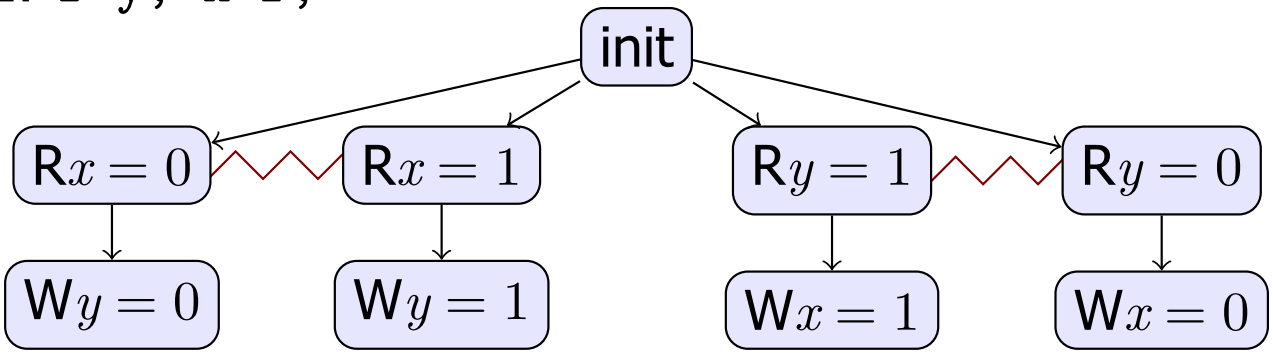
thread 1: `r=x; y=r;`
thread 2: `r=y; x=r;`



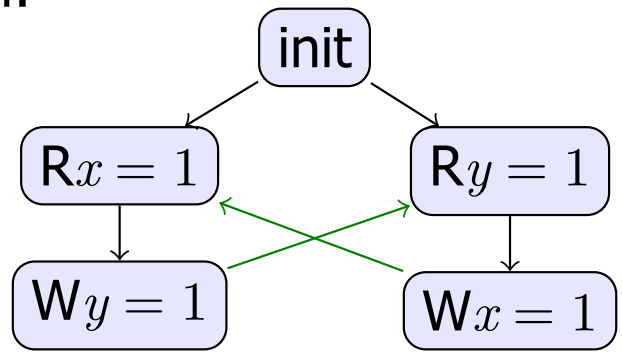
THIN AIR READS

Oh dear, the TAR pit:

thread 1: `r=x; y=r;`
thread 2: `r=y; x=r;`



Has justified configuration:

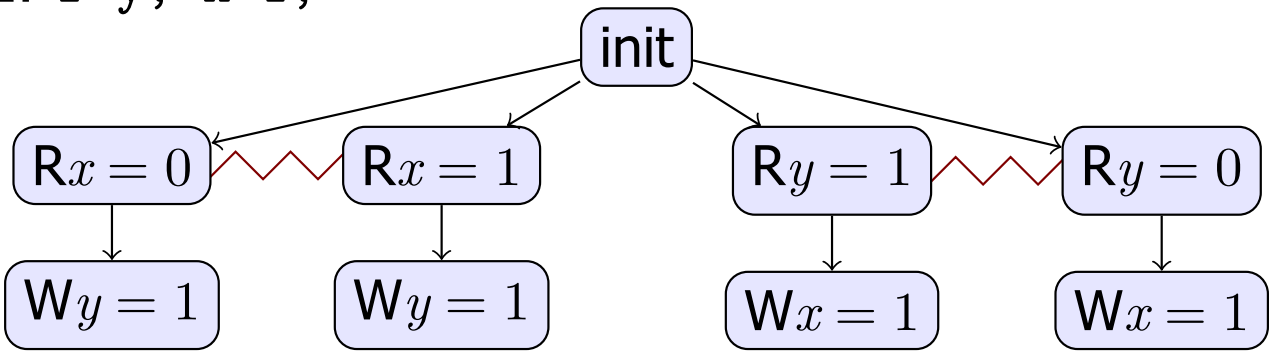


A TAR caused by a cycle in (justification + program-order).

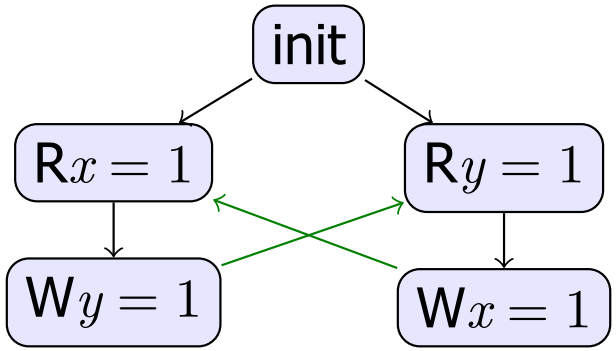
THIN AIR READS

Banning such cycles kills instruction reordering:

thread 1: `r=x; y=1;`
thread 2: `r=y; x=1;`



since the expected behaviour has exactly the same cycle:



THIN AIR READS

Handwave: allow conflict when avoiding TAR.

Analogous to JMM candidate executions.

THIN AIR READS

Handwave: allow conflict when avoiding TAR.

Analogous to JMM candidate executions.

A **configuration** is a $\leq$-downclosed, $\#$-free set of events.

A **pre-configuration** is a $\leq$-downclosed set of events.

THIN AIR READS

Handwave: allow conflict when avoiding TAR.

Analogous to JMM candidate executions.

A **configuration** is a $\leq$-downclosed, $\#$-free set of events.

A **pre-configuration** is a $\leq$-downclosed set of events.

A pre-configuration is **relaxed justified** when it has a total order such that any non-initial e has a justifier $j(e) \leq_{to} e$.

A configuration is **relaxed justified** when it is included in a relaxed justified pre-configuration.

THIN AIR READS

Handwave: allow conflict when avoiding TAR.

Analogous to JMM candidate executions.

A **configuration** is a $\leq$-downclosed, $\#$-free set of events.

A **pre-configuration** is a $\leq$-downclosed set of events.

A pre-configuration is **relaxed justified** when it has a total order such that any non-initial e has a justifier $j(e) \leq_{to} e$.

A configuration is **relaxed justified** when it is included in a relaxed justified pre-configuration.

Proposal: allowed executions are relaxed justified configurations.

THIN AIR READS

Handwave: allow conflict when avoiding TAR.

Analogous to JMM candidate executions.

A **configuration** is a $\leq$-downclosed, $\#$-free set of events.

A **pre-configuration** is a $\leq$-downclosed set of events.

A pre-configuration is **relaxed justified** when it has a total order such that any non-initial e has a justifier $j(e) \leq_{to} e$.

A configuration is **relaxed justified** when it is included in a relaxed justified pre-configuration.

Proposal: allowed executions are relaxed justified configurations.

DRF Theorem holds for relaxed justified configurations
(under some mild technical conditions).

THIN AIR READS

The stuff I just brushed under the carpet…

Properties of the alphabet:

- If $a \in \mathsf{RWC}(b)$ and $b \in \mathsf{RWC}(c)$ then $a \in \mathsf{RWC}(c)$.

Properties of the event structure:

- If $c < d \sim e$ then $c \leq e$.
- If $c \# d \sim e$ and $c \not\geq e$ then $c \# e$.
- If $d \sim e$ and $\lambda(d) \in \mathsf{RWC}(a)$ then $\lambda(e) \in \mathsf{RWC}(a)$.
- If $d \sim e$ and $a \in \mathsf{RWC}(\lambda(d))$ then $a \in \mathsf{RWC}(\lambda(e))$.
- If $d \sim e$ and $a \in \mathsf{RWJ}(\lambda(d))$ then $a \notin \mathsf{RWJ}(\lambda(e))$.
- For any $a \in \mathsf{RWC}(\lambda(e))$ there is a $d \sim e$ where $a \in \mathsf{RWJ}(\lambda(d))$.

where $d \sim e$ is the minimal conflict relation:

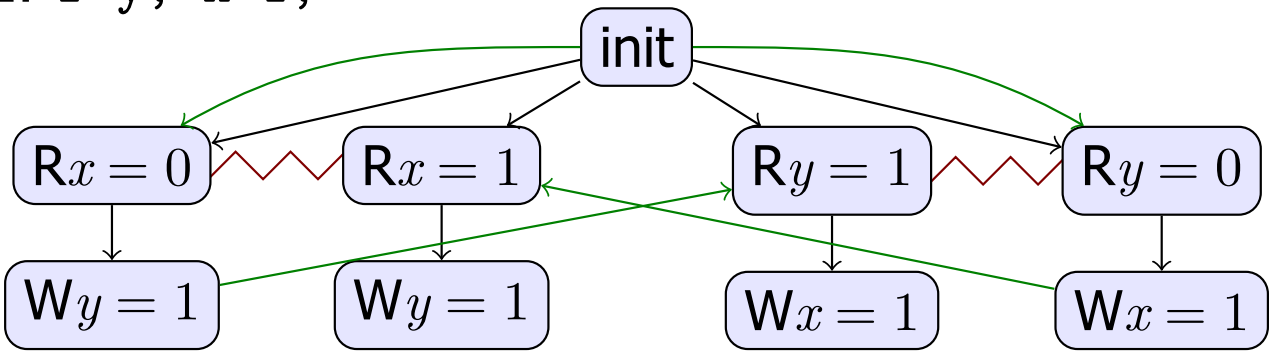
- $d \sim e$ whenever $d \# e$ and if $d \geq b \# c \leq e$ then $d = b \# c = e$.

In example programs, $d \sim e$ is generated by conflicting reads.

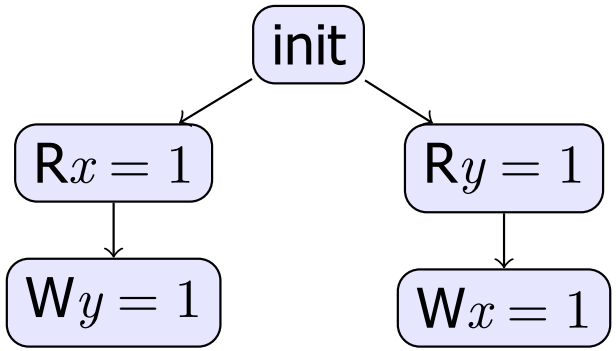
THIN AIR READS

Instruction reordering example:

thread 1: `r=x; y=1;`
thread 2: `r=y; x=1;`



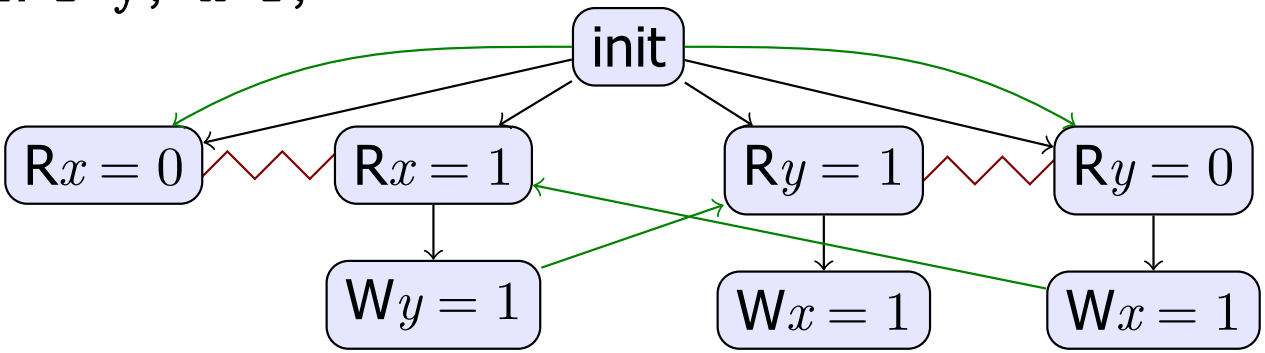
includes relaxed justified configuration:



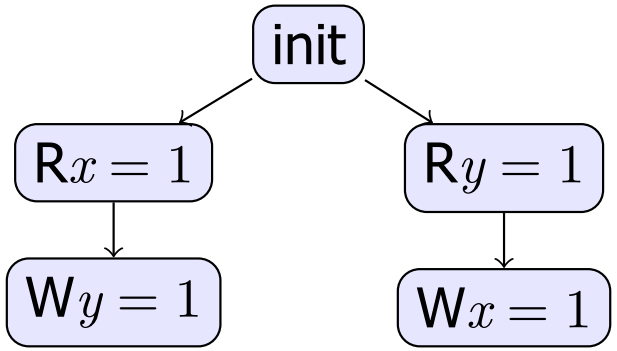
THIN AIR READS

Speculative read example:

thread 1: `r=x; if(r) y=1;`
thread 2: `r=y; x=1;`



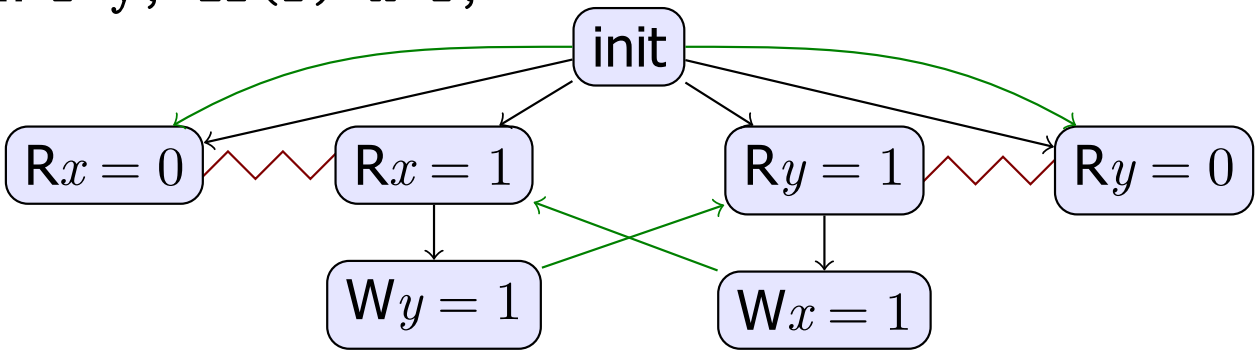
includes relaxed justified configuration:



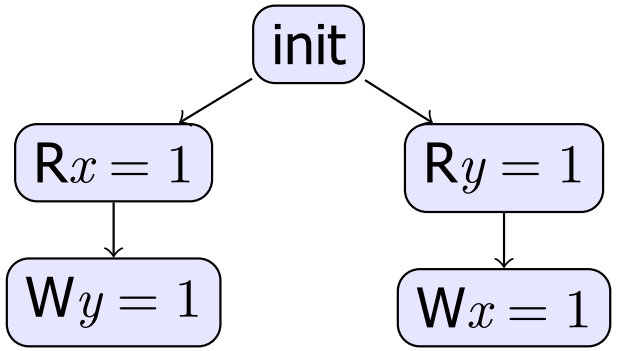
THIN AIR READS

Another TAR pit:

thread 1: `r=x; if(r) y=1;`
thread 2: `r=y; if(r) x=1;`



is cyclic, so this configuration is **not** relaxed justified:



SYNCHRONIZATION ACTIONS

Model so far is only for relaxed memory.

Let's add synchronization actions (motivating example: Java volatiles).

SYNCHRONIZATION ACTIONS

Model so far is only for relaxed memory.

Let's add synchronization actions (motivating example: Java volatiles).

Assume Sync $\subseteq \Sigma$.

In a totally ordered configuration, this introduces **synchronization** relation:

$$\frac{\lambda(d), \lambda(e) \in \mathsf{Sync} \quad d \leq_{to} e \quad \lambda(d) \in \mathsf{RWC}(\lambda(e))}{d \leq_{so} e}$$

and **happens before** order:

$$\frac{d \leq e}{d \leq_{hb} e} \qquad \frac{d \leq_{so} e}{d \leq_{hb} e} \qquad \frac{c \leq_{hb} d \leq_{hb} e}{c \leq_{hb} e}$$

(Note that if there are no synchronization actions then $\leq$ is the same as $\leq_{hb}$).

SYNCHRONIZATION ACTIONS

Recall that in the absence of synchronization actions, d is a justifier for e when $e \not\leq d$, $d \in \text{RWJ}(e)$, and there is no $d \leq c \leq e$ where $c \in \text{RWC}(e)$.

SYNCHRONIZATION ACTIONS

In the presence of synchronization actions, d is a justifier for e when $e \not\leq_{hb} d$, $d \in \text{RWJ}(e)$, and there is no $d \leq_{hb} c \leq_{hb} e$ where $c \in \text{RWC}(e)$.

SYNCHRONIZATION ACTIONS

In the presence of synchronization actions,
d is a justifier for e when $e \not\leq_{hb} d$, $d \in \text{RWJ}(e)$, and
there is no $d \leq_{hb} c \leq_{hb} e$ where $c \in \text{RWC}(e)$.

(These definitions coincide when there are no synchronization actions.)

SYNCHRONIZATION ACTIONS

In the presence of synchronization actions,
d is a justifier for e when $e \not\leq_{hb} d$, $d \in \mathrm{RWJ}(e)$, and
there is no $d \leq_{hb} c \leq_{hb} e$ where $c \in \mathrm{RWC}(e)$.

(These definitions coincide when there are no synchronization actions.)

Previous definitions and results go through, under some more technical requirements…

SYNCHRONIZATION ACTIONS

In the presence of synchronization actions,
d is a justifier for e when $e \not\leq_{hb} d$, $d \in \mathrm{RWJ}(e)$, and
there is no $d \leq_{hb} c \leq_{hb} e$ where $c \in \mathrm{RWC}(e)$.

(These definitions coincide when there are no synchronization actions.)

Previous definitions and results go through, under some more technical requirements…

- If $a \in \mathrm{RWC}(b)$ and $a \in \mathrm{Sync}$ then $b \in \mathrm{Sync}$ (this fails in C11)
- If $d \sim e$ and $d \in \mathrm{Sync}$ then $e \in \mathrm{Sync}$.

in a relaxed justified pre-configuration:

- If e is a synchronization event, then $j(e)$ sequentially justifies e.
- If $d \leq_{hb} j(e)$ then $\neg(d \# e)$.

and in a relaxed justified configuration C:

- If $j(e) \leq_{hb} \leq_{so} \leq_{hb} e$ and $e \in C$ then $j(e) \in C$.

SYNCHRONIZATION ACTIONS

In the presence of synchronization actions,
d is a justifier for e when $e \not\leq_{hb} d$, $d \in \mathrm{RWJ}(e)$, and
there is no $d \leq_{hb} c \leq_{hb} e$ where $c \in \mathrm{RWC}(e)$.

(These definitions coincide when there are no synchronization actions.)

Previous definitions and results go through, under some more technical requirements…

- If $a \in \mathrm{RWC}(b)$ and $a \in \mathrm{Sync}$ then $b \in \mathrm{Sync}$ (this fails in C11)
- If $d \sim e$ and $d \in \mathrm{Sync}$ then $e \in \mathrm{Sync}$.

in a relaxed justified pre-configuration:

- If e is a synchronization event, then $j(e)$ sequentially justifies e.
- If $d \leq_{hb} j(e)$ then $\neg(d \# e)$.

and in a relaxed justified configuration C:

- If $j(e) \leq_{hb} \leq_{so} \leq_{hb} e$ and $e \in C$ then $j(e) \in C$.

DRF Theorem still holds.

PROGRAM REFINEMENT

Looking for a definition of refinement $\sqsubseteq$ between event structures which is:

- is a preorder
- is compositional: if $P \sqsubseteq Q$ then $C[P] \sqsubseteq C[Q]$ for any program context C
- validates common compiler optimizations
  (roach motel, variable reordering and thread inlining)

PROGRAM REFINEMENT

Looking for a definition of refinement $\sqsubseteq$ between event structures which is:

- is a preorder
- is compositional: if $P \sqsubseteq Q$ then $C[P] \sqsubseteq C[Q]$ for any program context C
- validates common compiler optimizations
  (roach motel, variable reordering and thread inlining)

We have one.

PROGRAM REFINEMENT

Define $ES_1 \sqsubseteq ES_2$ whenever

- there is a binary relation $R \subseteq E_1 \times E_2$,
- for every $e_2 \in E_2$ there is $e_1 \in E_1$ such that $(e_1, e_2) \in R$,
- for every $(e_1, e_2) \in R$, we have $\lambda_1(e_1) = \lambda_2(e_2)$,
- for every $(d_1, d_2) \in R$ and $(e_1, e_2) \in R$, we have $d_1 \#_1 e_1$ iff $d_2 \#_2 e_2$,
- for every synchronized write d_2 with $(d_1, d_2) \in R$ and $d_2 <_2 e_2$, there exists $d_1 <_1 e_1$ such that $(e_1, e_2) \in R$,
- for every synchronized read e_2 with $(e_1, e_2) \in R$ and $d_2 <_2 e_2$, there exists $d_1 <_1 e_1$ such that $(d_1, d_2) \in R$, and
- for every synchronized read e_2 with $(e_1, e_2) \in R$ and $d_2 \ll_2 e_2$, there exists $d_1 \ll_1 e_1$ such that $(d_1, d_2) \in R$.

where $d \ll e$ whenever $d < c < e$ for some $c \in \mathrm{WWC}(d)$.

PROGRAM REFINEMENT

$\sqsubseteq$ is a **preorder.**

PROGRAM REFINEMENT

$\sqsubseteq$ is a **preorder**.

$\sqsubseteq$ is **compositional**, in that it respects the following operations:

- Prefixing ($a.ES$ adds a new event labelled a to the beginning of ES).
- Parallel composition ($ES_1 \mid ES_2$ is the disjoint union of ES_1 and ES_2).
- Sum ($ES_1 + ES_2$ is $ES_1 \mid ES_2$, but with conflict between E_1 and E_2).

Enough to give semantics for a simple shared-memory concurrent language.

PROGRAM REFINEMENT

$\sqsubseteq$ is a **preorder**.

$\sqsubseteq$ is **compositional**.

$\sqsubseteq$ validates **roach motel** and **variable reordering**, since $a.b.ES \sqsubseteq b.a.ES$ whenever:

- $a \notin \text{WWC}(b)$,
- a is not a synchronized read, and
- b is not a synchronized write.

PROGRAM REFINEMENT

$\sqsubseteq$ is a **preorder**.

$\sqsubseteq$ is **compositional**.

$\sqsubseteq$ validates **roach motel** and **variable reordering**.

$\sqsubseteq$ validates **thread inlining**, since $ES_1 \sqsubseteq ES_2$ whenever:

$$E_1 = E_2 \quad \leq_1 \subseteq \leq_2 \quad \#_1 = \#_2 \quad \lambda_1 = \lambda_2$$

that is, more things are related by program order in ES_2 than in ES_1.

TO DO LIST

Tooling.

Conjecture: program refinement respects relaxed justified configurations.

Per-location SC (aka coherence).

Richer alphabets of synchronization actions.

Object creation (c.f. Lochlieber).

Reasoning by invariants (e.g. type safety).

Specification language for APIs to describe synchronization (currently done by English, e.g. in java collections API).

CONCLUSIONS

Event structures provide a model for relaxed memory.

Basic model is event structures with justified configurations.

Scales up (at cost of complexity) to TAR and synchronization actions.

Supports program refinement which handles common optimizations (roach motel, variable reordering and thread inlining).