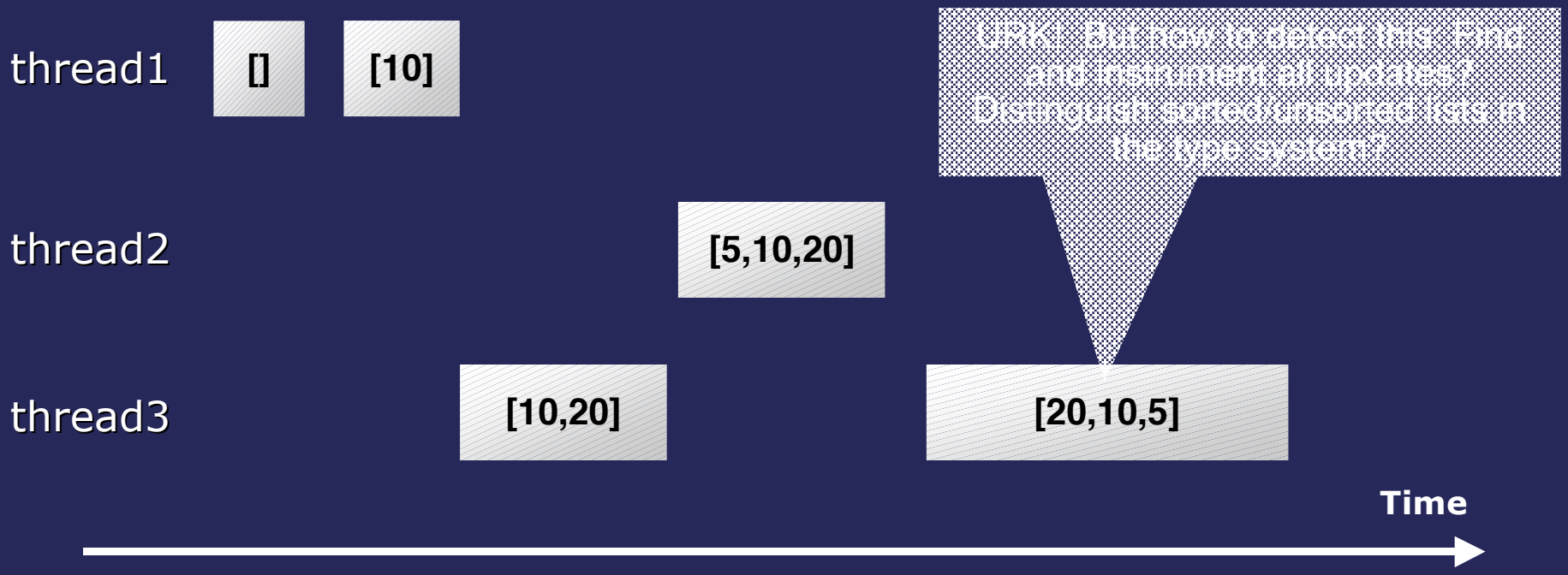
Transactional memory with data invariants: or "putting the C back in ACID"

Tim Harris, Simon Peyton Jones

Microsoft Research Cambridge

Introduction

- "List XYZ remains in sorted order"



Transactions to the rescue

- Write *executable invariants* to check the properties we're interested in
- Invariants fit in really nicely with transactions:
  - Of course, the DB folk knew this long ago :-)
  - *Every transaction must preserve every invariant*
  - Transactions define updates between *consistent* states
  - Invariants can (must!) be broken within transactions
  - It doesn't matter if an update is to the spine of the list or to a key that it contains (unlike much related work, e.g. Eiffel and Spec#)

A sorted list using invariants

Defining a function 'newSortedList' in Haskell-ish pseudo-code

```
newSortedList = do {
  r <- newList();
  check (isSorted r);
  return r;
}
```

Construct an ordinary mutable list as usual

Assert that the list is sorted now, and *every update to the list must keep it sorted*

Return the list we created

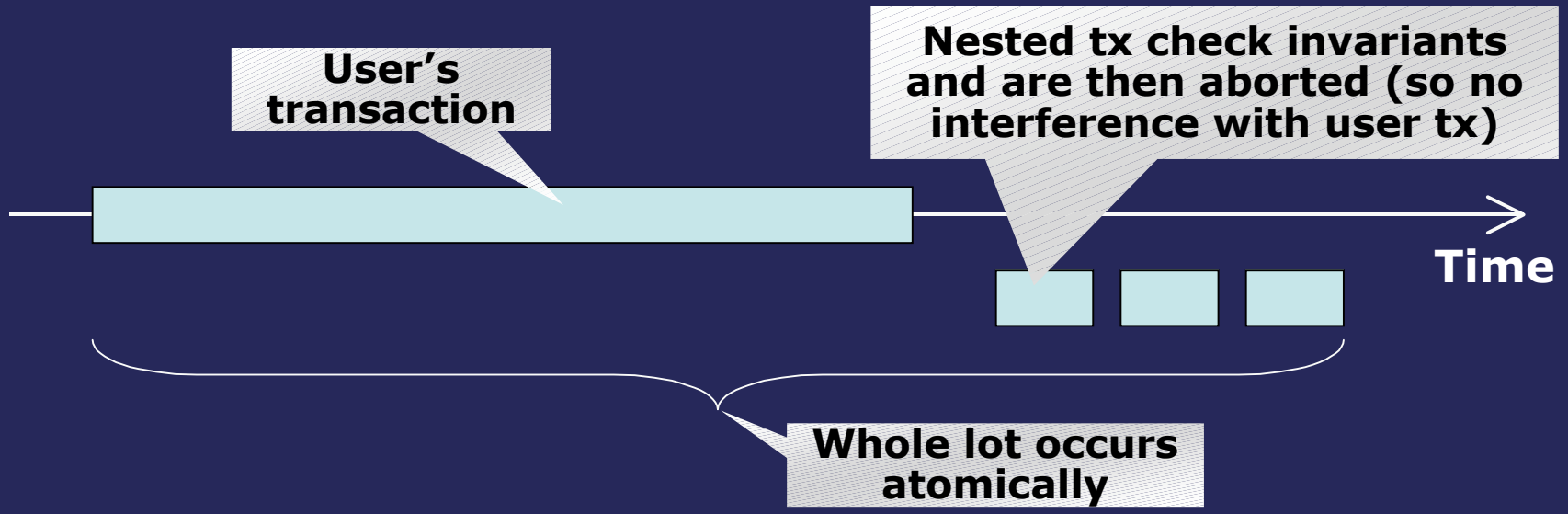
- The runtime system keeps track of the invariant and ensures transactions don't violate it – the type system treats it as any other list

Semantics

- The invariant is an STM action… so [in Haskell] it's guaranteed not to do I/O
  - But what if it loops?
  - What if it updates or allocates transactional state?
  - What if it calls *check* to add yet another invariant?
  - What if it uses condition synchronization and blocks the transaction?

First design choice: overview

- Run invariants in [closed] nested tx (so they see the tentative updates), check they succeed (don't throw an exception), then roll back each nested tx
- All this happens atomically with the user tx



First design choice: semantics

- The invariant is an arbitrary STM action…
  - What if it loops?
  - What if it updates trans state?
  - What if it calls *check*?
  - What if it blocks?

Your program loops…

Can use them internally, but updates discarded

Checked at that call, but then discarded

The user's tx blocks until the invariant can complete

STM Haskell

Transactional state is held in TVars

```
newTVar :: a -> STM (TVar a)

readTVar :: TVar a -> STM a

writeTVar :: TVar a -> a -> STM ()

atomic :: STM a -> IO a
```

The type system distinguishes transactional code (types like “STM a”) from general imperative code (types like “IO a”)

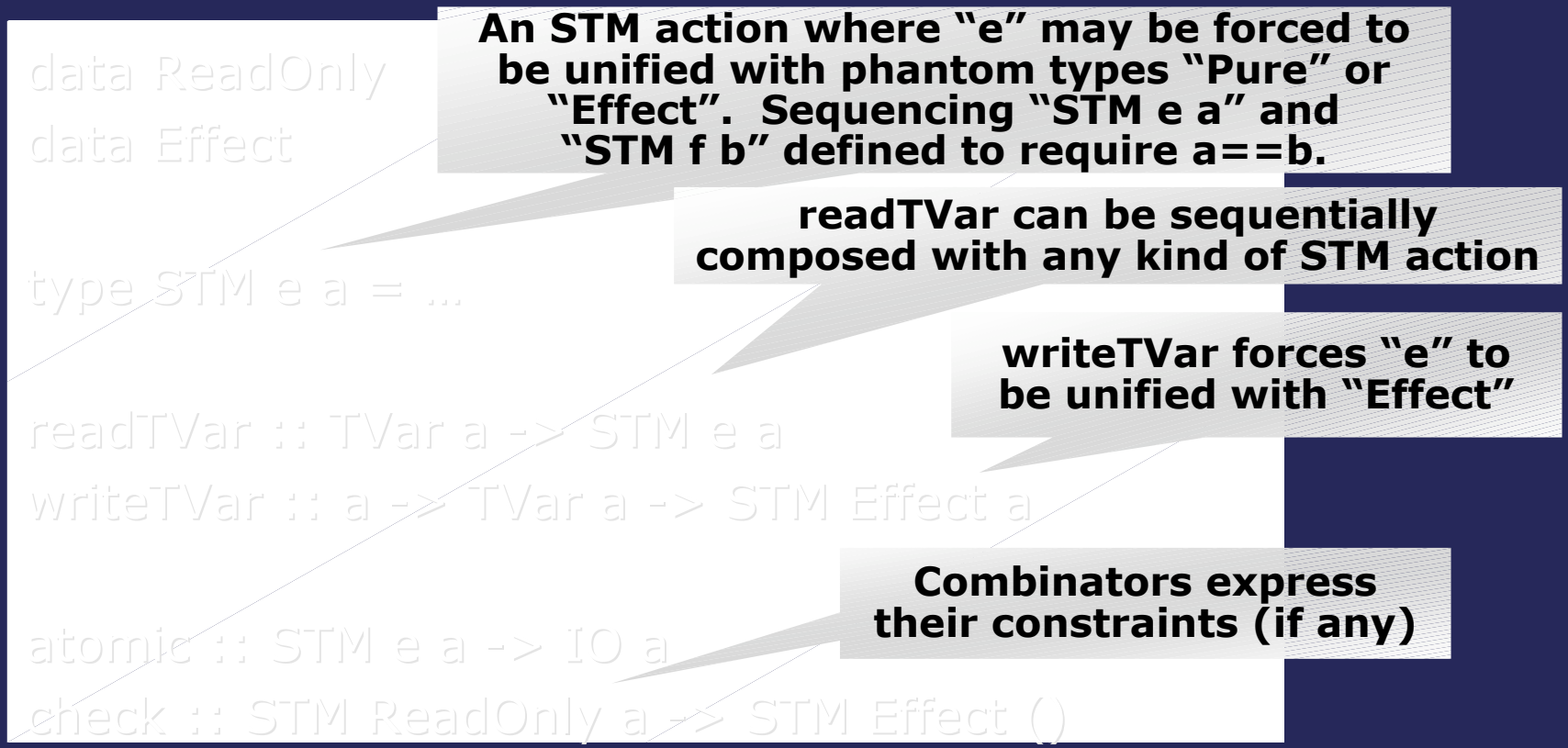
```
incT :: TVar Int -> STM ()

incT r = do { v <- readTVar r
  ; writeTVar r (v+1) }

main = do { r <- atomic (newTVar 0)
  ; fork (atomic (incT r))
  ; atomic (incT r)
  ; ... }
```

Sequential composition of STM actions

Microsoft Research Cambridge

Second design choice: restrict to reads

- We can distinguish between STM actions that just read from the heap from STM actions that perform updates / create TVars / block / add invariants



Microsoft Research Cambridge

Invariants over state pairs

- Exciting observation from the Spec# group
- Suppose we want invariants over state pairs rather than single states
  - "Value of x never decreases"
  - versus "Value of x is always positive"

Take an STM action and run it in the *pre-transactional* state

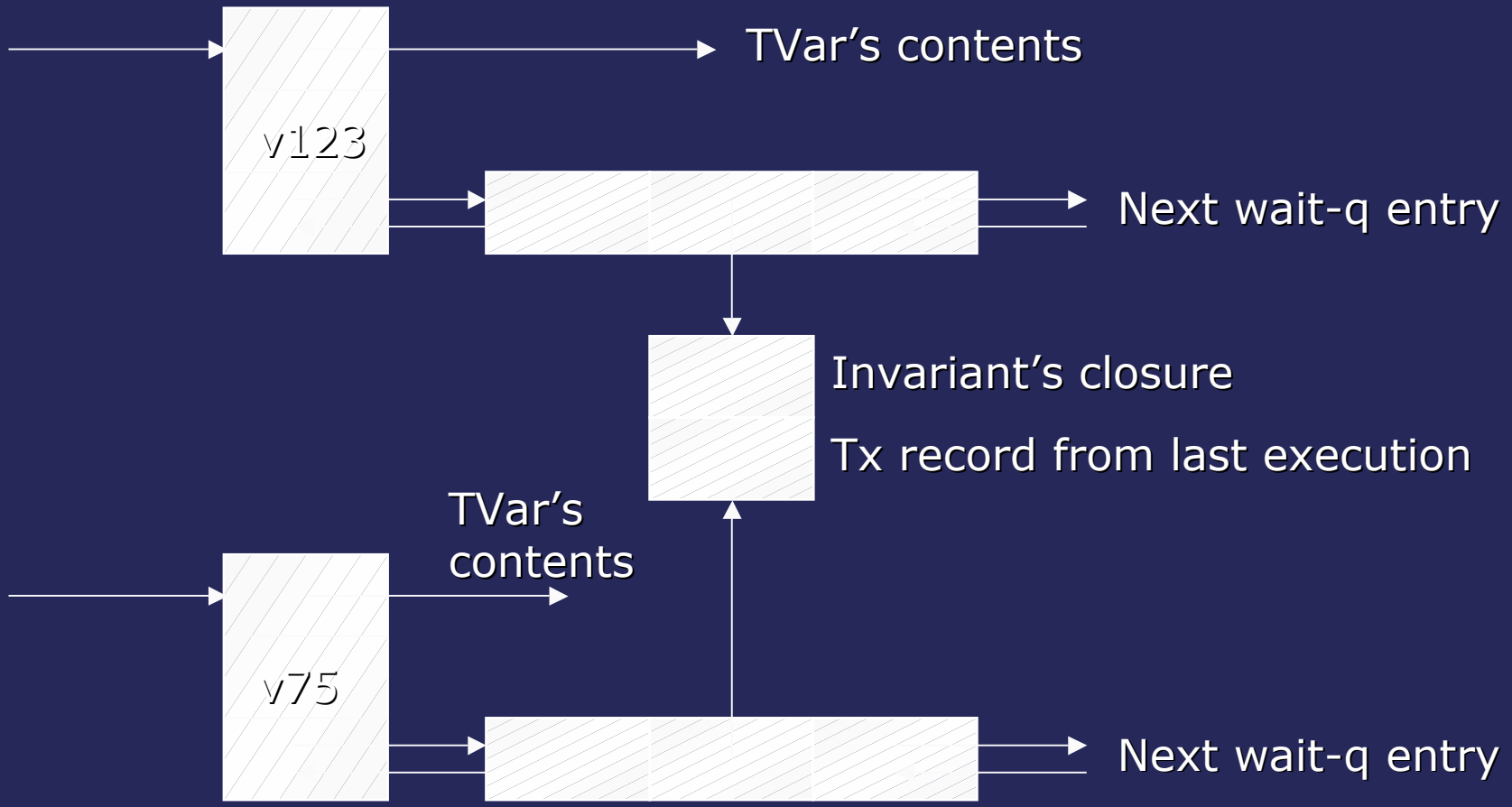
```
old :: STM ReadOnly a -> STM e a

check (do { cur <- readTVar x;
            prev <- old (readTVar x);
            assert (prev <= cur); } )
```

Microsoft Research Cambridge

Implementation

- Isn't it slow checking every invariant upon every transaction?
  - It would be if we actually checked them all
  - When invariant checking is enabled we dynamically track dependencies from TVars to invariants that depend on them
  - "Pay for play" (same wake-up mechanism we use for condition synchronization)
  - These are the only references to the data structures used to represent invariants: the GC reclaims these structures when they are unreachable
  - (But note that the extra links may extend the lifetime of individual objects – the invariant and everything reachable from it will be retained while at least one TVar it depends on is reachable)

Implementation (2)



Conclusions

- First system to integrate invariants with transactional memory
- "Putting the C back in ACID"
- Many of the ideas have a long history; both semantically and in the implementation
- Do we want some kind of "trigger"-like construct too?
- Workloads workloads workloads

Microsoft Research Cambridge

Papers

- "Transactional memory with data invariants" TRANSACT 2006
- "Lock-free data structures using STM in Haskell" FLOPS 2006 – larger examples and SMP performance eval
- "Composable memory transactions" PPoPP 2005 – software transactions in Haskell
- "Haskell on a shared-memory multiprocessor" Haskell 2005 – parallel thunk evaluation
- "Exceptions and side-effects in atomic blocks" CSJP 2004 – integration of transactional memory and external resource managers
- "Optimizing memory transactions" PLDI 2006 – C# work with MSR Redmond
- "Concurrent programming without locks" under submission – algorithmic gore for scalable STM

Backup slides

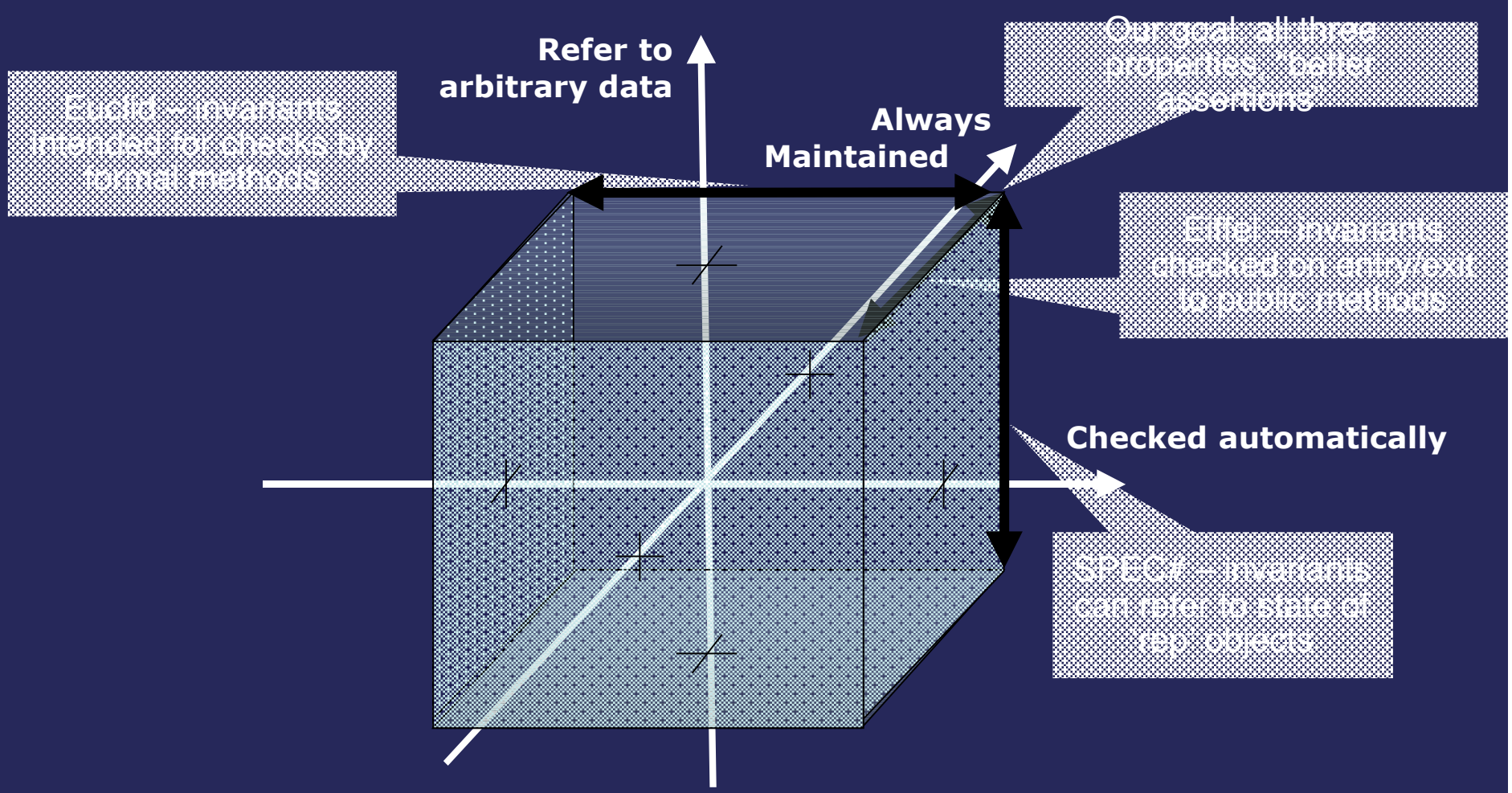
STM Haskell

```
retry :: STM a

tryDecT :: TVar Int -> STM ()


tryDecT r = do {
  v <- readTVar r;
  if (v == 0)
    retry
  else
    writeTVar r (v-1); }
```

- "retry" for condition synchronization
- Semantics: abort the tx (all the way out), re-execute it
- Implementation: smarter, block the thread until it's worth trying re-execution

Microsoft Research Cambridge

(Some axes of) the design space



- YMMV: e.g. another axis would be "could be checked statically"

Microsoft Research Cambridge