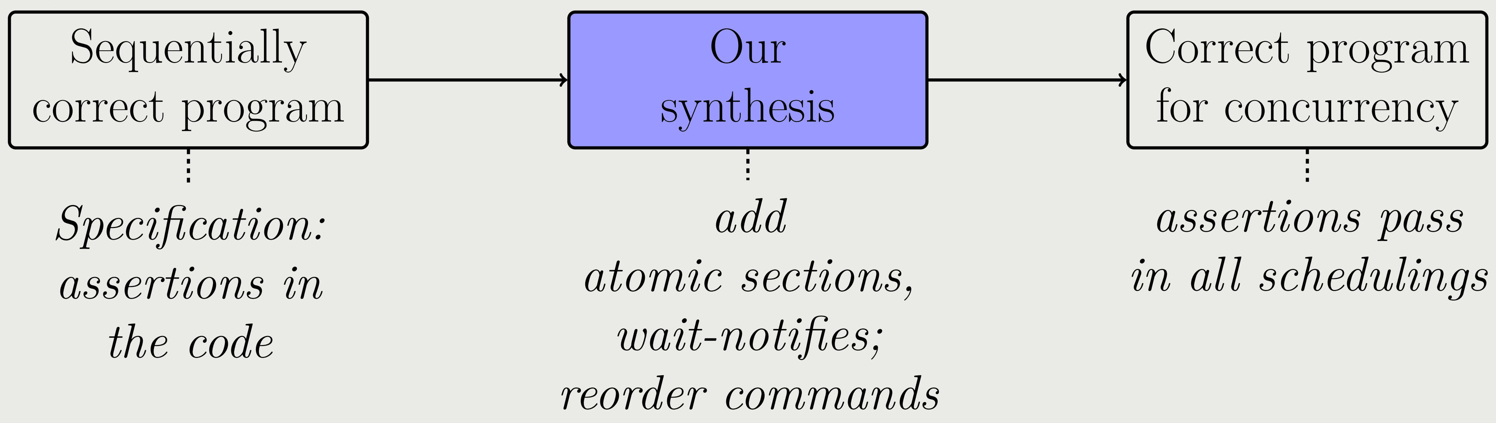
Synchronisation Synthesis for Concurrent Programs

Thorsten Tarrach (joint work with Pavol Černý, Ashutosh Gupta, Thomas A. Henzinger, Arjun Radhakrishna, Leonid Ryzhyk and Roopsha Samanta)

IST AUSTRIA
Institute of Science and Technology

Introduction: Concurrency bugs

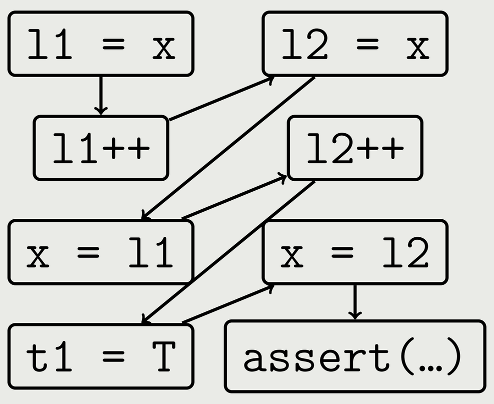
- Concurrency bugs are hard to find and fix
- We attempt to fix them automatically using synthesis
- **Specification:**



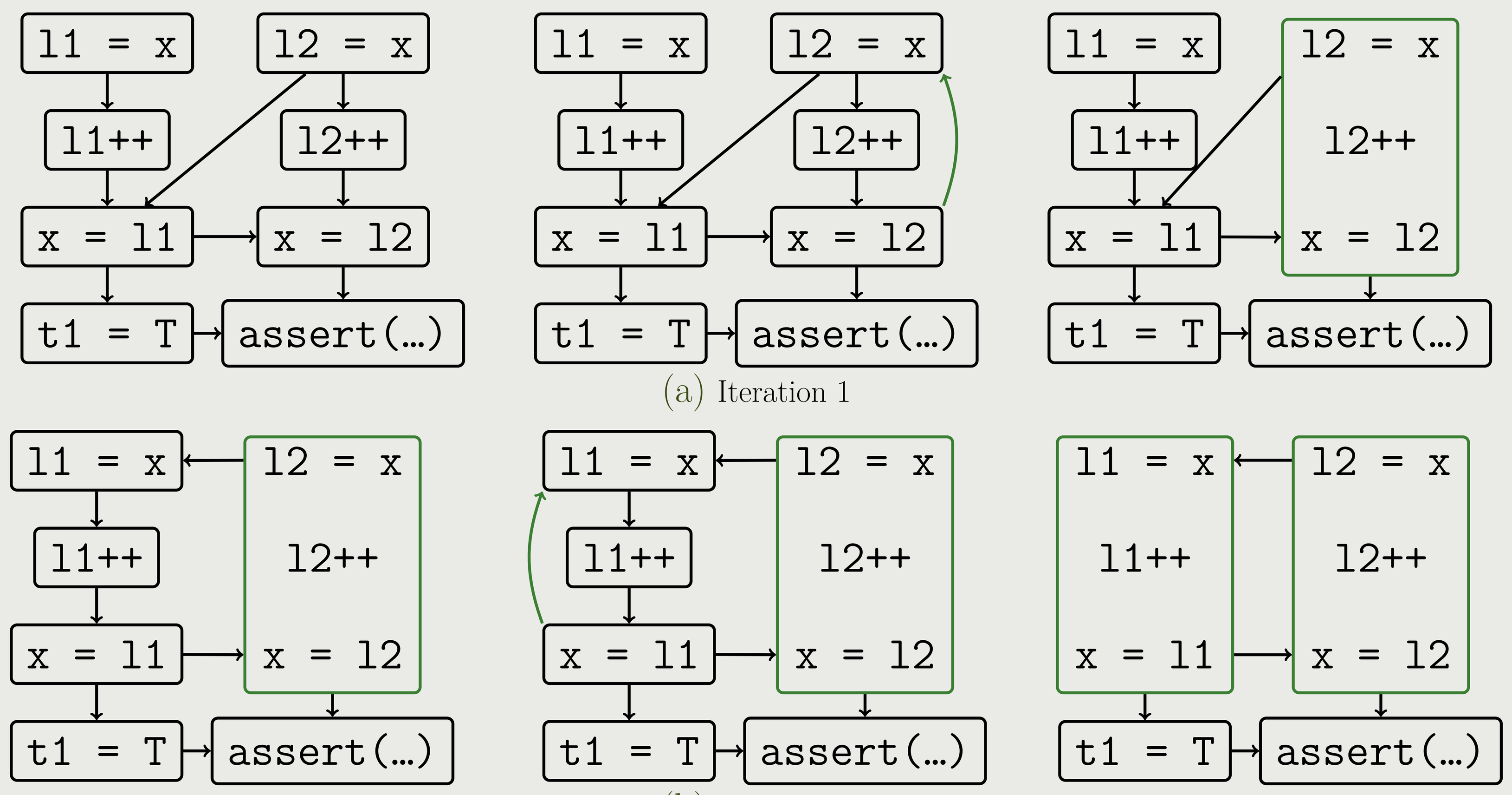
Atomic sections example

- This example requires two atomic sections to be fixed
- With a linear trace we cannot infer where to place atomic sections

```
init: x = 0; t1 = F
thread1      thread2
A: l1 = x    1: l2 = x
B: l1++      2: l2++
C: x = l1    3: x = l2
D: t1 = T    4: assert(¬t1∨x==2)
```



- Using a happens-before relationship we can infer atomic sections after two iterations
  - An atomic section is denoted by a loop inside a thread (it is created by adding an edge)

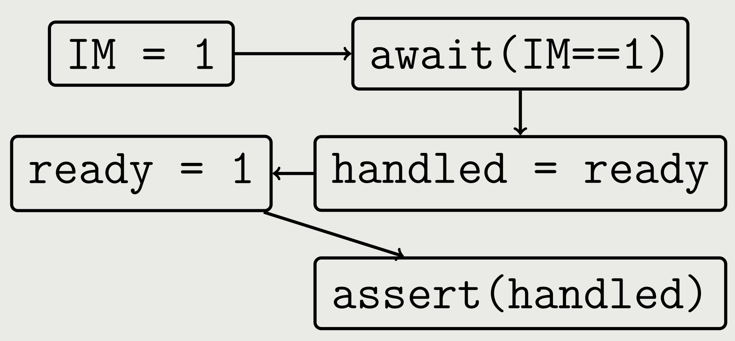


(a) Iteration 1

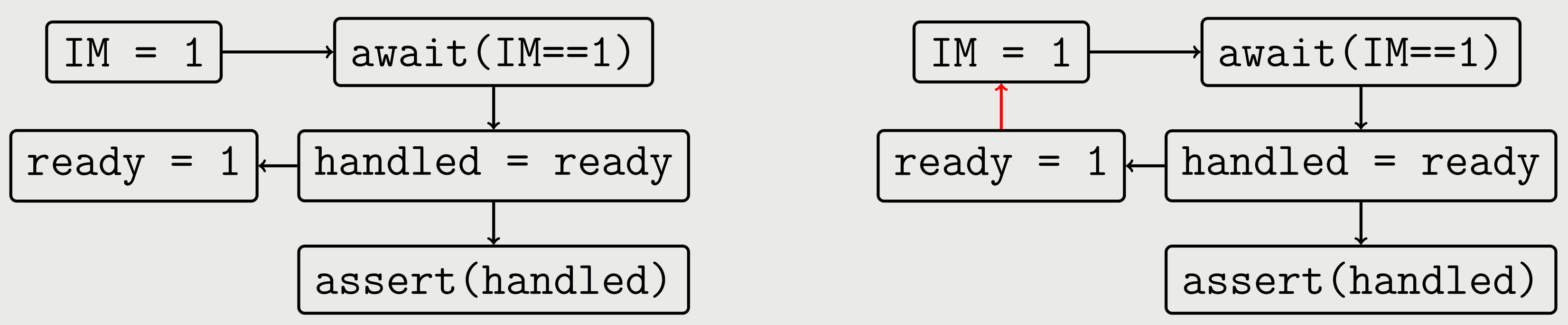
(b) Iteration 2

Reordering example

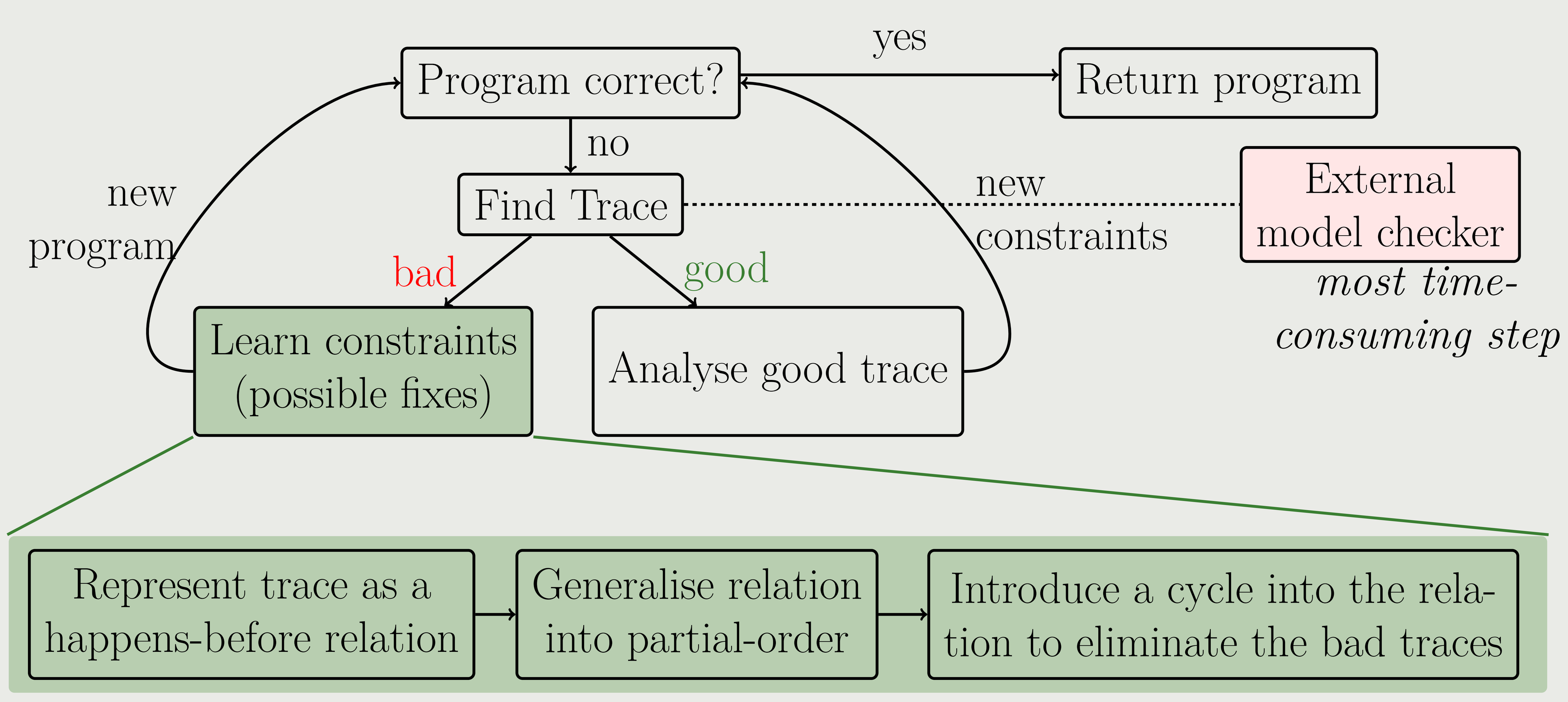
```
init: IntrMask=0; ready=0; handled=0
init_thread       intr_thread
M: IntrMask = 1   R: await(IntrMask==1)
N: ready = 1      S: handled = ready
                  T: assert(handled)
```



- We remove edges from the partial order if M; N ≡ N; M
  - If such an edge is readded to create a cycle it means the two corresponding statements will be swapped in the program



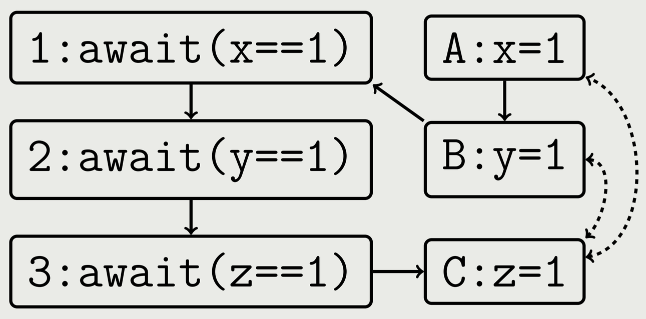
Synthesis algorithm outline



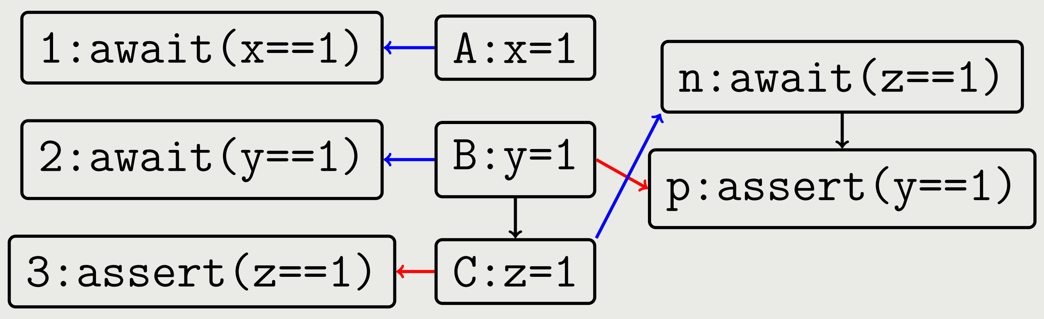
Preventing regressions by using good traces

- Reordering can cause regressions
  - By analysing a good trace we can identify possible regressions before reordering instructions

```
init: x = 0; y = 0; z = 0
thread1           thread2   thread3
1: await(x==1)    A: x=1    n: await(z==1)
2: await(y==1)    B: y=1    p: assert(y==1)
3: assert(z==1)   C: z=1
```



- A;B;1;2;3 causes assertion 3 to fail
- 2 possible fixes: swap B ↔ C or swap A ↔ C
- Swapping B ↔ C can lead to assertion p failing



- We analyse good trace A;B;C;1;2;3;n;p
- Blue edges indicate data-flow dependencies of awaits, red of asserts
- We learn not to reorder B;C and n;p to protect the data-flow into assertion p
- After good trace analysis only the correct fix A ↔ C remains

Conclusion

- We consider reorderings as fixes
- We generalise the counter-example trace to capture the cause of the error
- We pervent regressions by analysing good traces

Recent: Better trace generalisation

- Trace generalisation is crucial to the success of the synthesis
- Trace generalisation should capture the core of the bug
- Idea: Represent traces as a Boolean formula over happens-before constraints

```
global: x, withdrawal, deposit, balance, deposited, withdrawn
init: x = balance; deposited = 0; withdrawn = 0

π:
thread_withdraw:
localvars: temp
W1: temp = balance
W2: balance = temp - withdrawal
W3: withdrawn = 1

thread_deposit:
localvars: temp
D1: temp = balance
D2: balance = temp + deposit
D3: deposited = 1

thread_checkresult:
C1: assume (deposited == 1 ∧ withdrawn == 1)
C2: assert (balance == x + deposit - withdrawal)
```

Original Trace: $\pi = W_1, D_1, W_2, W_3, D_2, D_3, C_1, C_2$

Representation of bad interleavings of π:

$\mathcal{N}^b_\pi = hb(W_1, D_2) \wedge hb(D_1, W_2)$

Representation of good interleavings of π:

$\mathcal{N}^g_\pi = hb(D_2, W_1) \vee hb(W_2, D_1)$

- We introduce rewrite rules on $\mathcal{N}^g_\pi$ for synthesis, e.g.

$$\frac{hb(X_j, Y_k) \vee hb(Y_\ell, X_i) \vee \psi \quad i \leq j \quad k \leq \ell}{lock(X_{[i,j]}, Y_{[k,\ell]}) \vee \psi}\ \text{ADD.LOCK}$$

The ADD.LOCK rewriting rule yields $lock(W_{[1,2]}, D_{[1,2]})$

References

[1] A. Griesmayer, R. Bloem, and B. Cook. Repair of Boolean Programs with an Application to C. In *CAV*, 2006.

[2] A. Gupta, T. Henzinger, A. Radhakrishna, S. Roopsha, and T. Tarrach. Succinct Representation of Concurrent Trace Sets. In *POPL*, 2015.

[3] B. Jobstmann, A. Griesmayer, and R. Bloem. Program Repair as a Game. In *CAV*, 2005.

[4] R. Samanta, J. Deshmukh, and A. Emerson. Automatic Generation of Local Repairs for Boolean Programs. In *FMCAD*, 2008.

[5] A. Solar-Lezama, C. Jones, and R. Bodík. Sketching concurrent data structures. In *PLDI*, 2008.

[6] M. Vechev, E. Yahav, and G. Yorsh. Abstraction-guided synthesis of synchronization. In *POPL*, 2010.

[7] P. Černý, K. Chatterjee, T. Henzinger, A. Radhakrishna, and R. Singh. Quantitative synthesis for concurrent programs. In *CAV*, 2011.

[8] P. Černý, T. Henzinger, A. Radhakrishna, L. Ryzhyk, and T. Tarrach. Efficient Synthesis for Concurrency by Semantics-Preserving Transformations. In *CAV*, 2013.

[9] P. Černý, T. Henzinger, A. Radhakrishna, L. Ryzhyk, and T. Tarrach. Regression-free Synthesis for Concurrency. In *CAV*, 2014.