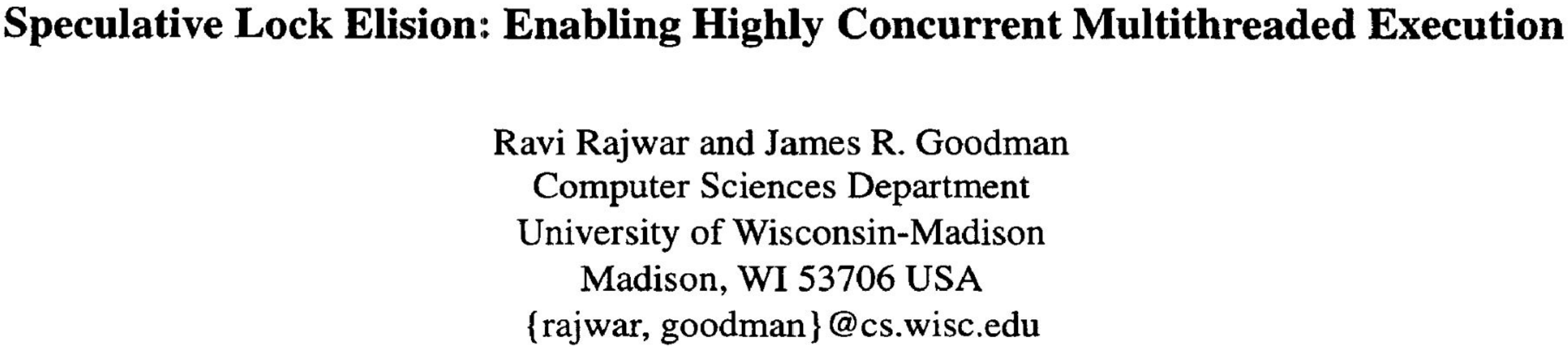
Speculative Lock Elision:

Enabling Highly Concurrent Multithreaded Execution

Ravi Rajwar and James R. Goodman
Computer Sciences Department
University of Wisconsin-Madison

MICRO 2001

Presented by: Andrea Lepori



Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution

Ravi Rajwar and James R. Goodman
Computer Sciences Department
University of Wisconsin-Madison
Madison, WI 53706 USA
{rajwar, goodman}@cs.wisc.edu

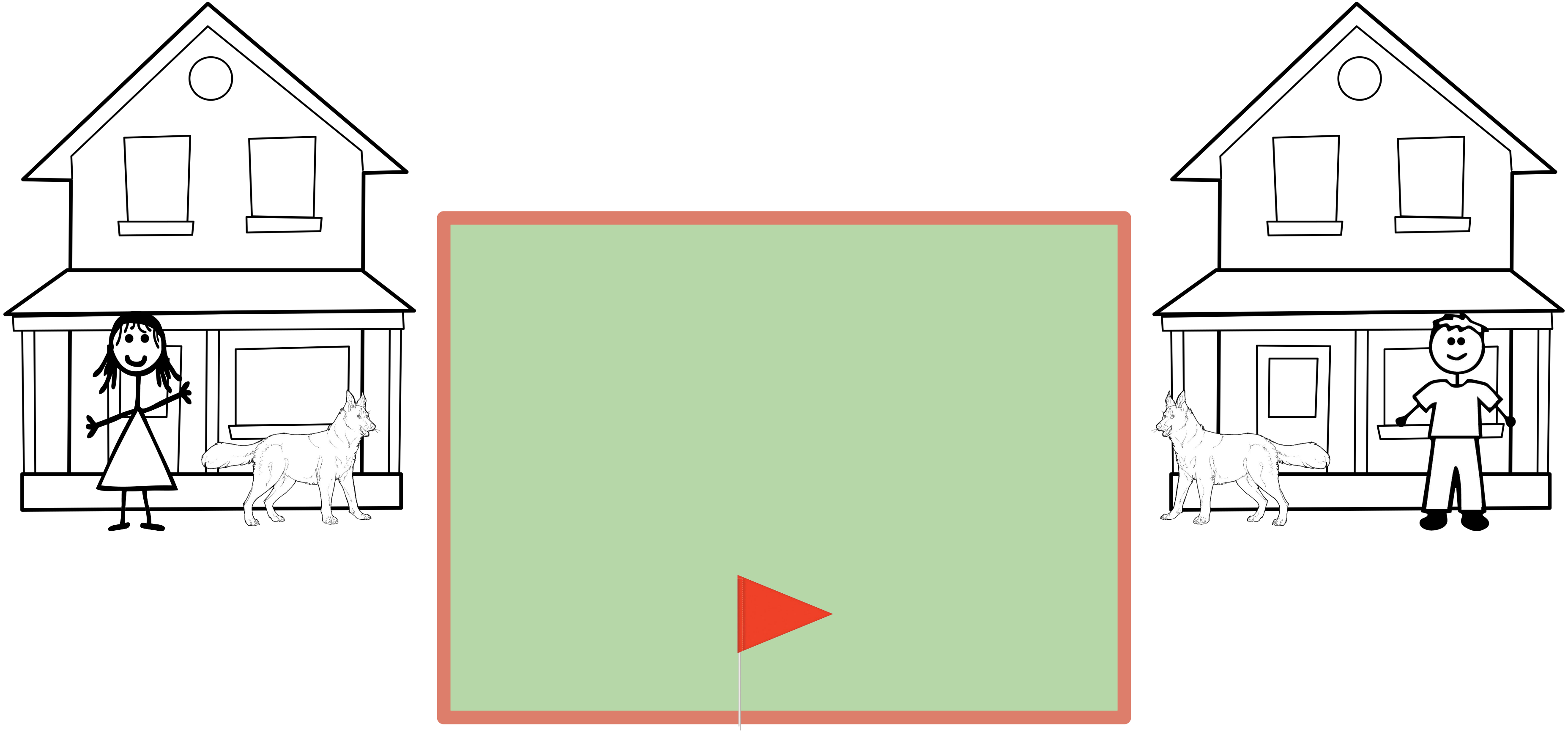
Table of contents

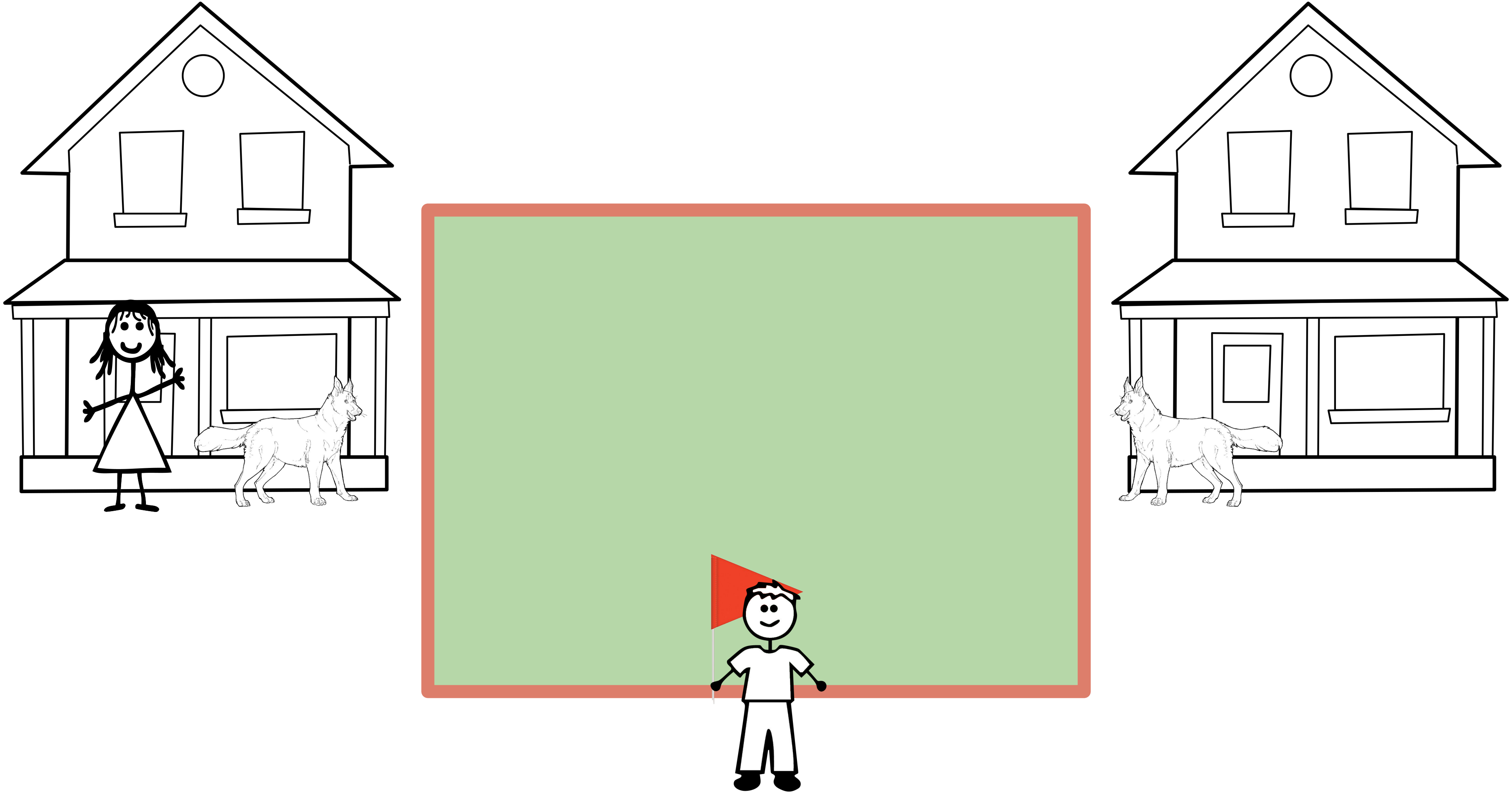
- Problem
- Goal of SLE
- Key Ideas
- SLE Algorithm (Mechanism)
- Implementation
- Methodology
- Strengths and weaknesses
- Thoughts and ideas

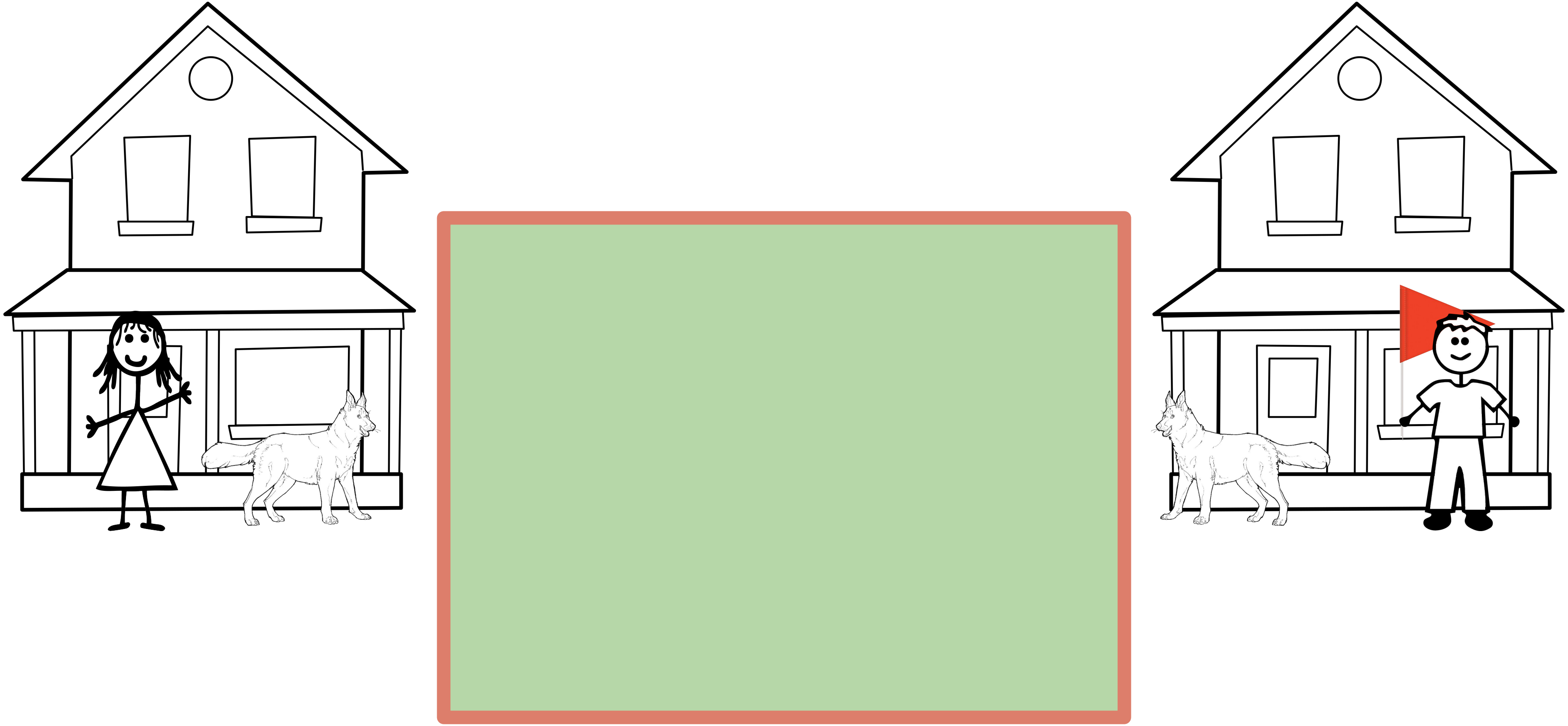
Executive summary

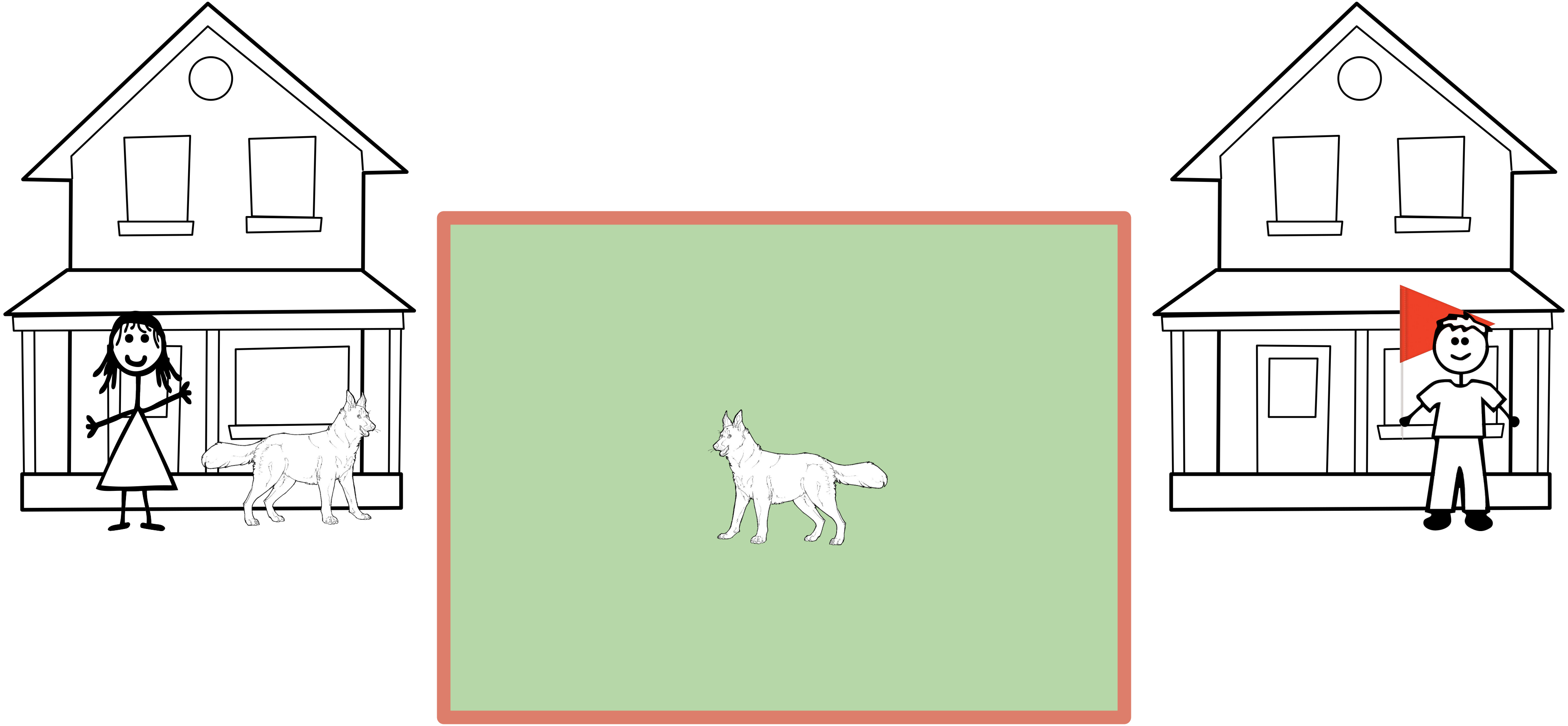
- **Problem:** Easier for the developers to do conservative locking but it comes with performance tradeoffs
- **Goal:** Enable highly concurrent multithreaded execution in a simple way
- **Key Approach**
  - Try to execute critical section without locks
  - If other threads interfere rollback
- **Evaluation:** Test SLE on different kind of multiprocessors
  - More than 50% of the locks were removed
  - Speedup on most workloads on most multiprocessors

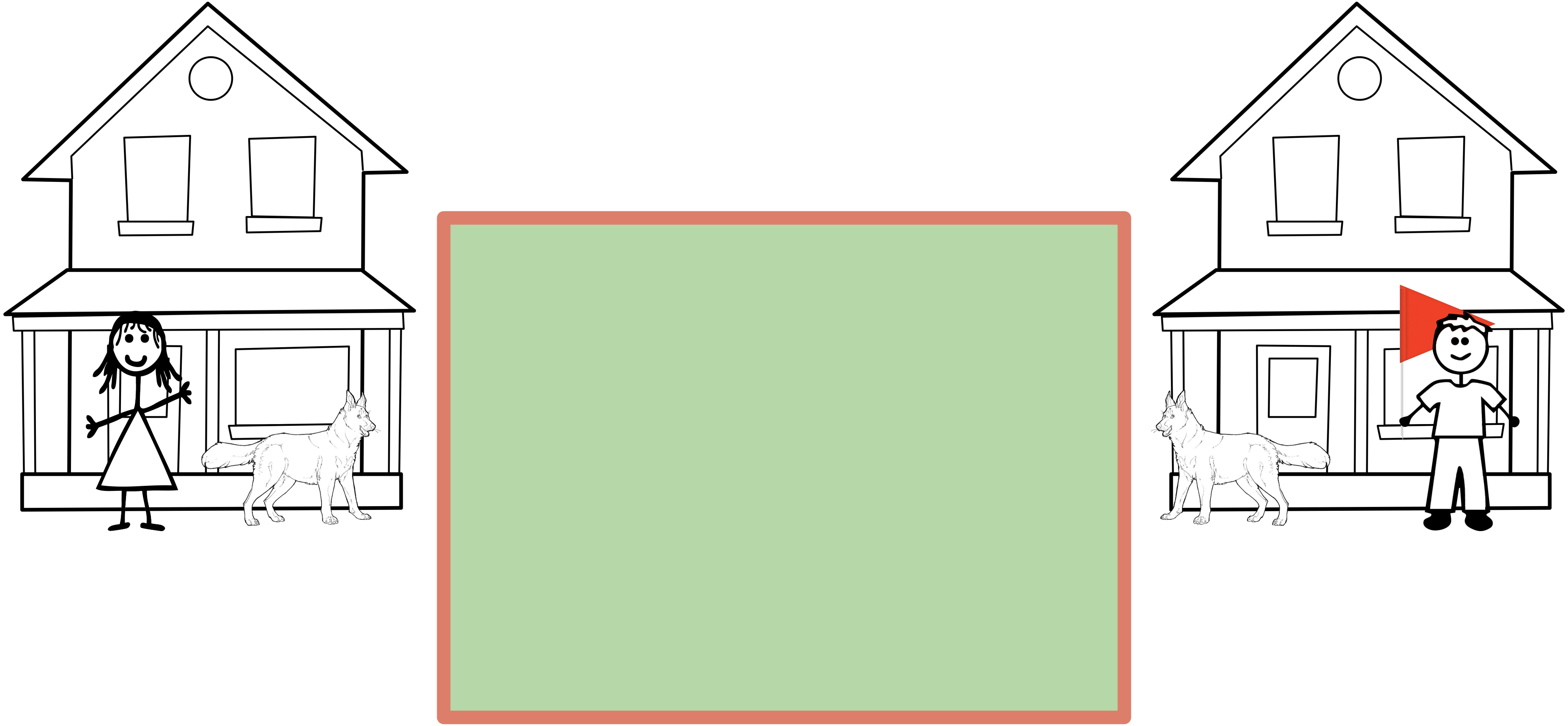
Problem

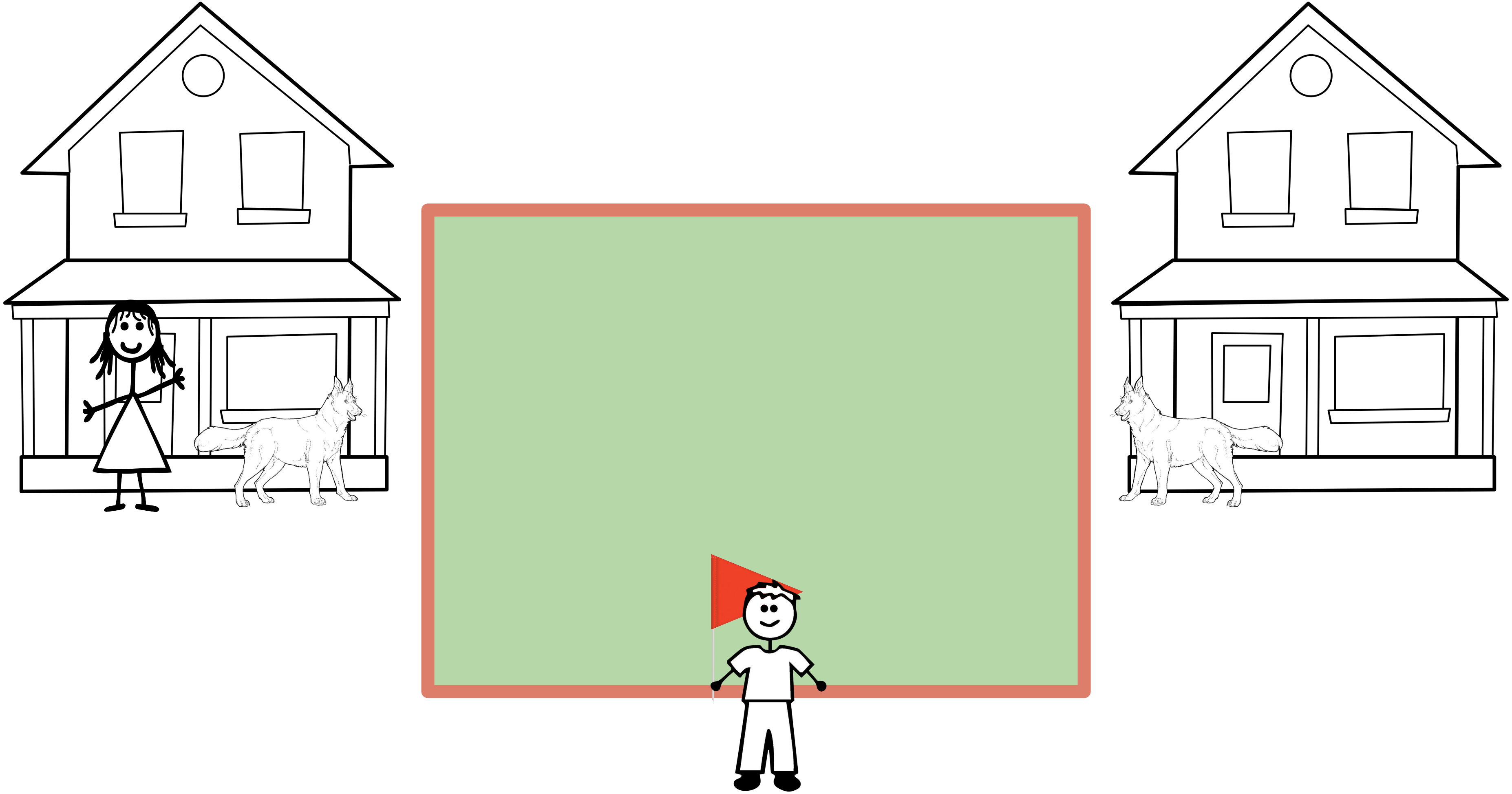


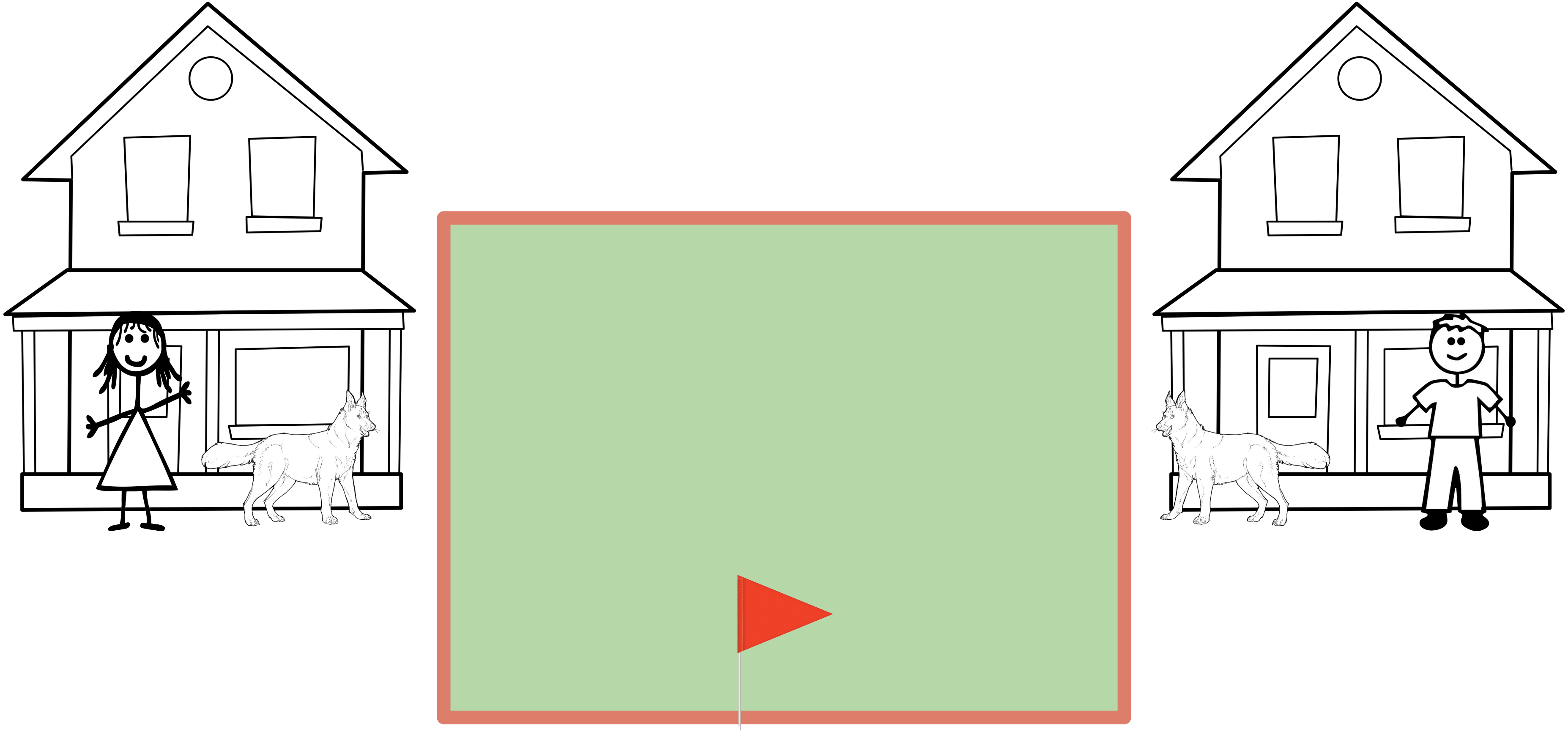


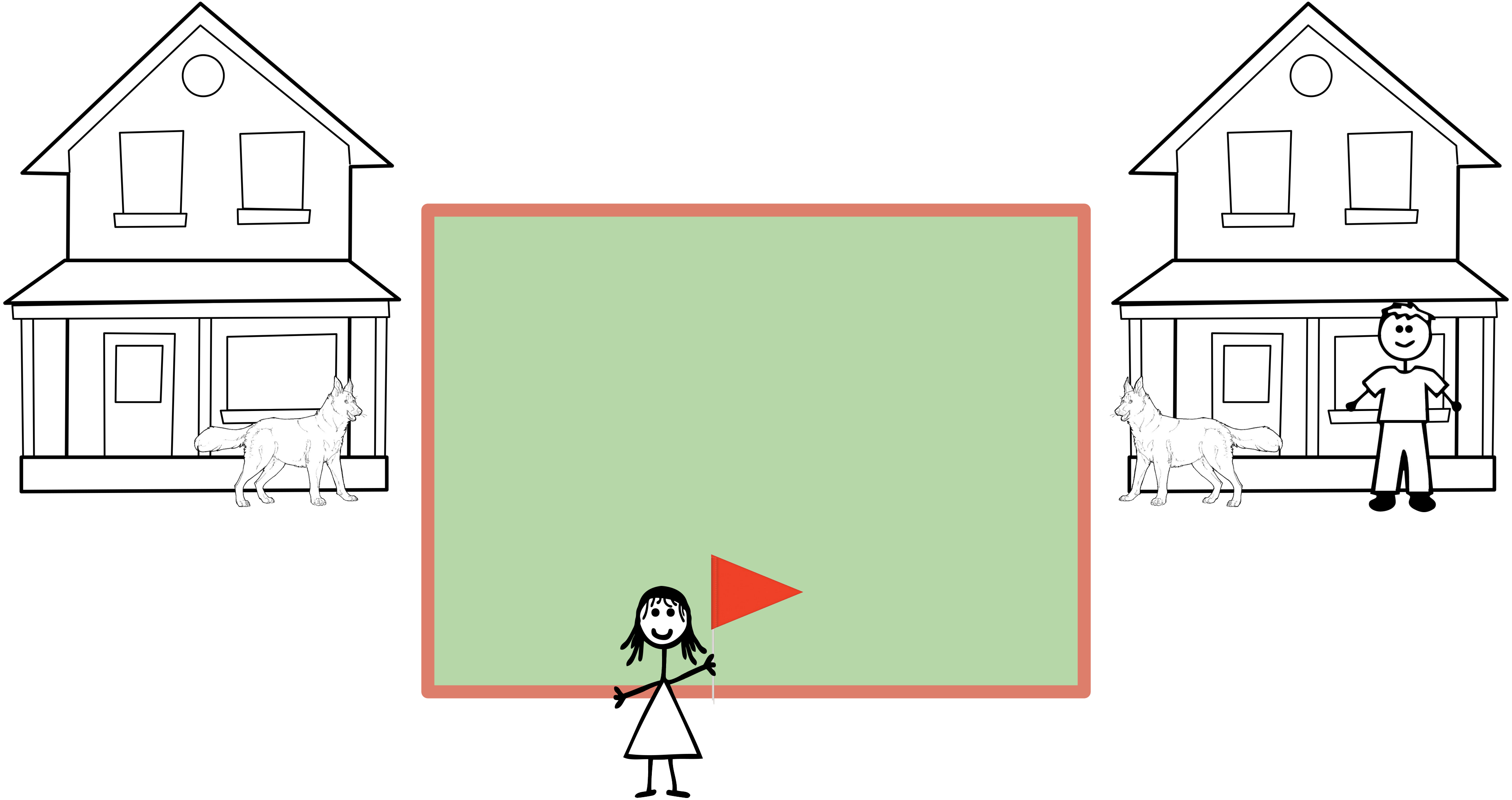


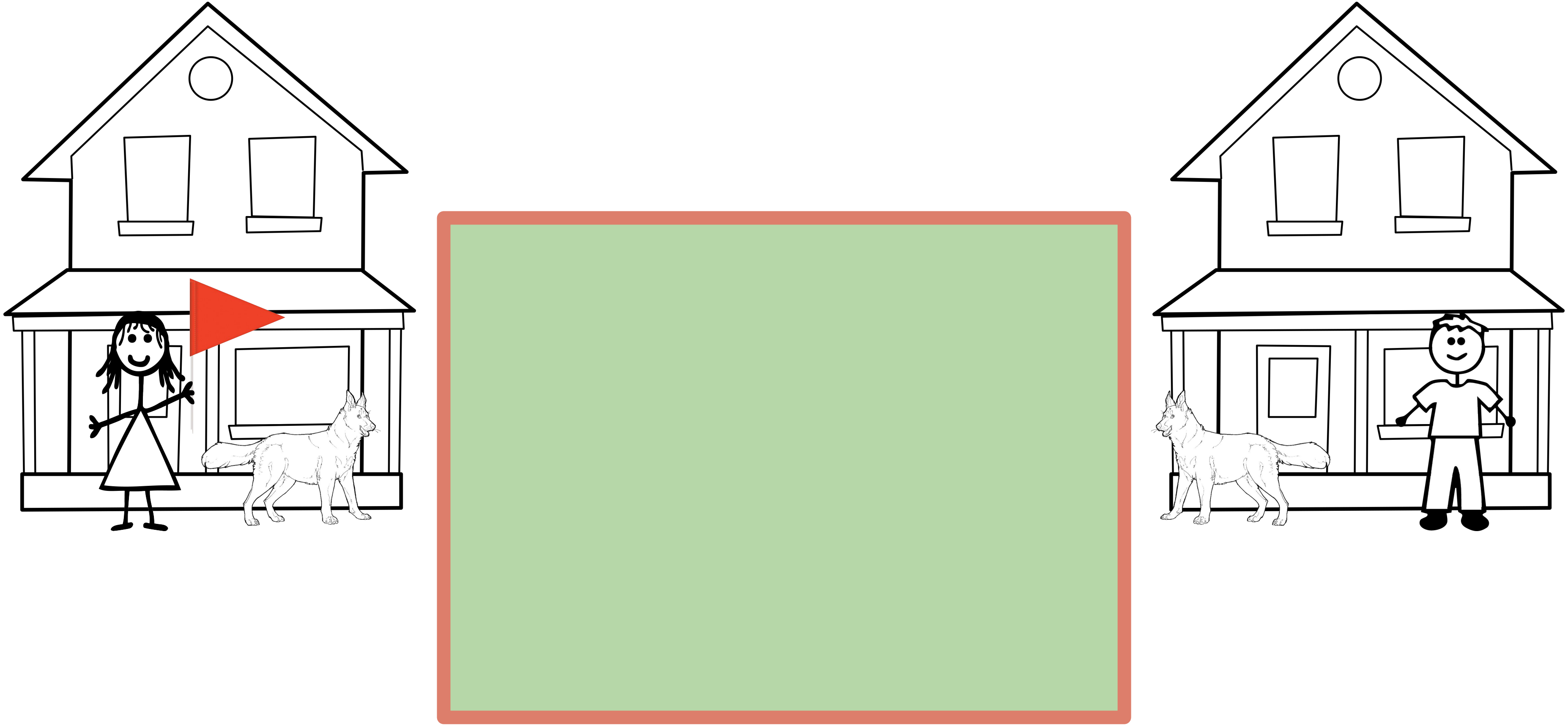


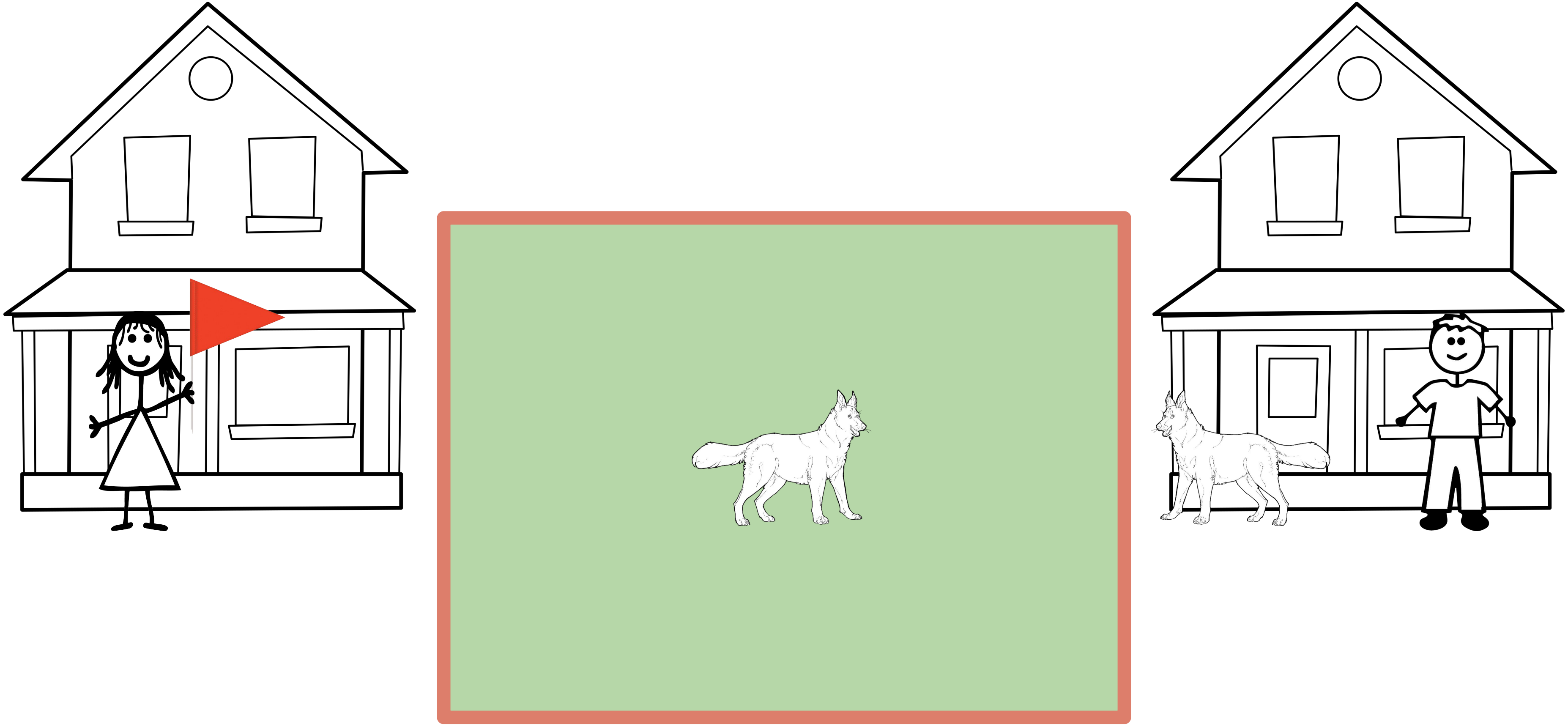


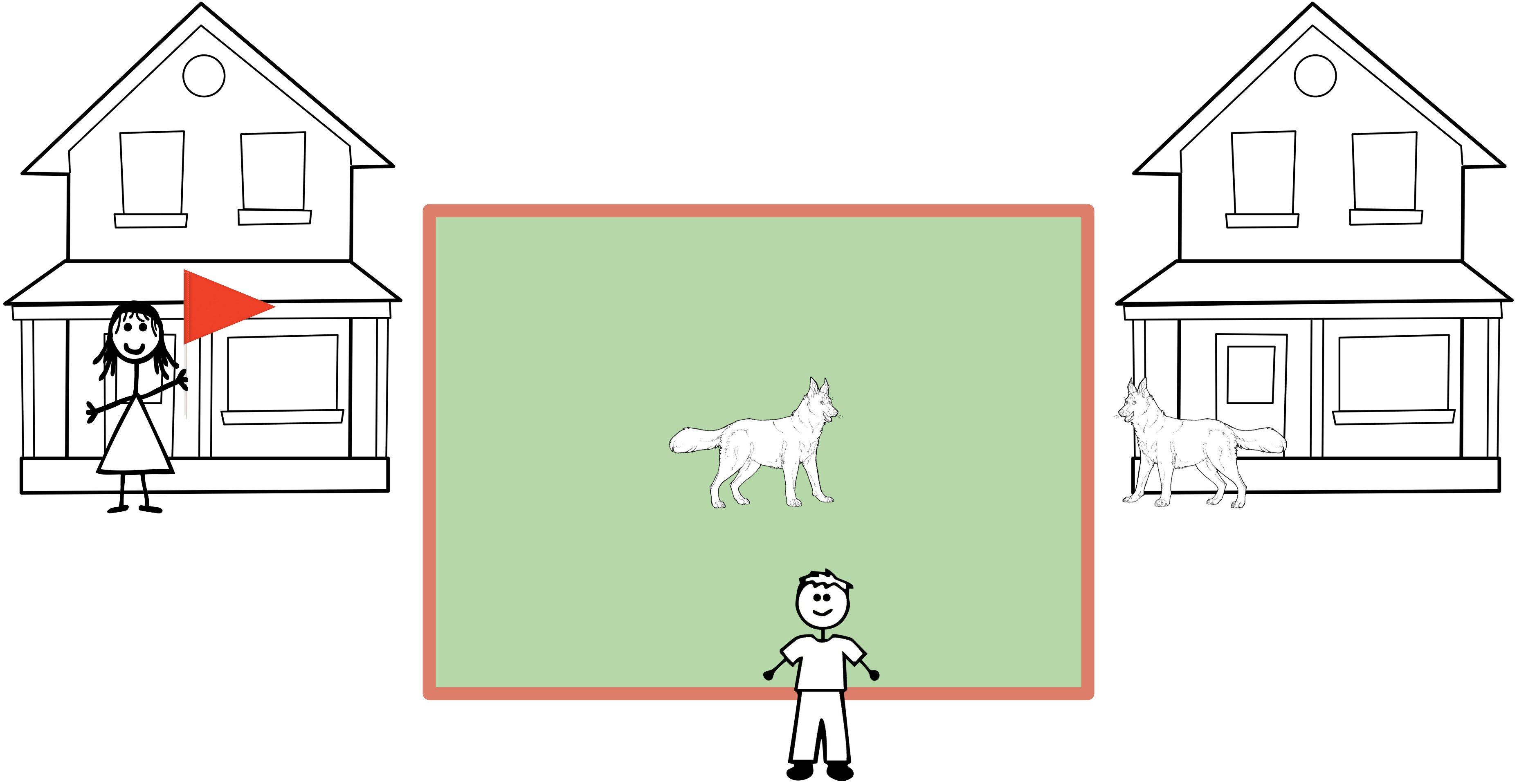


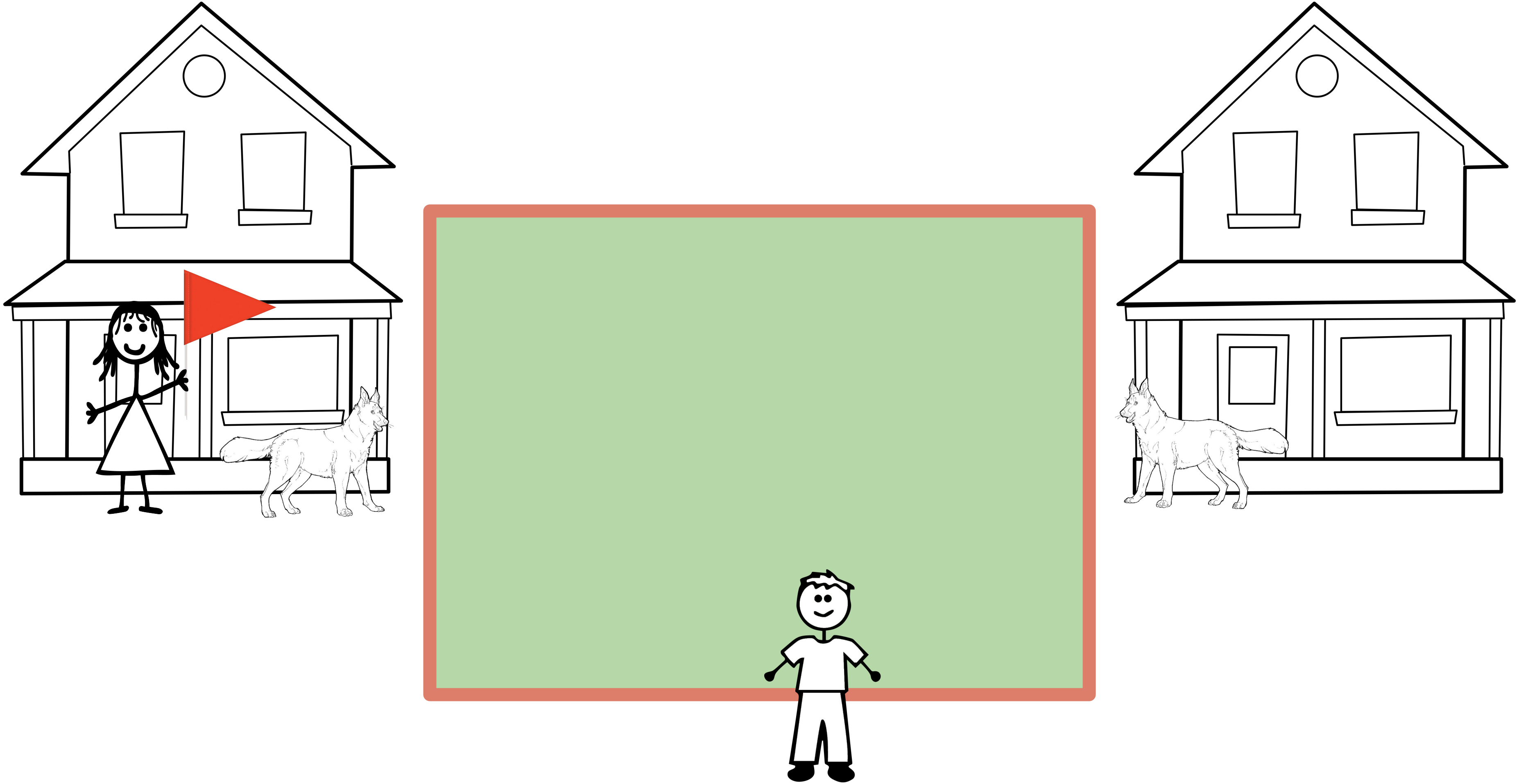


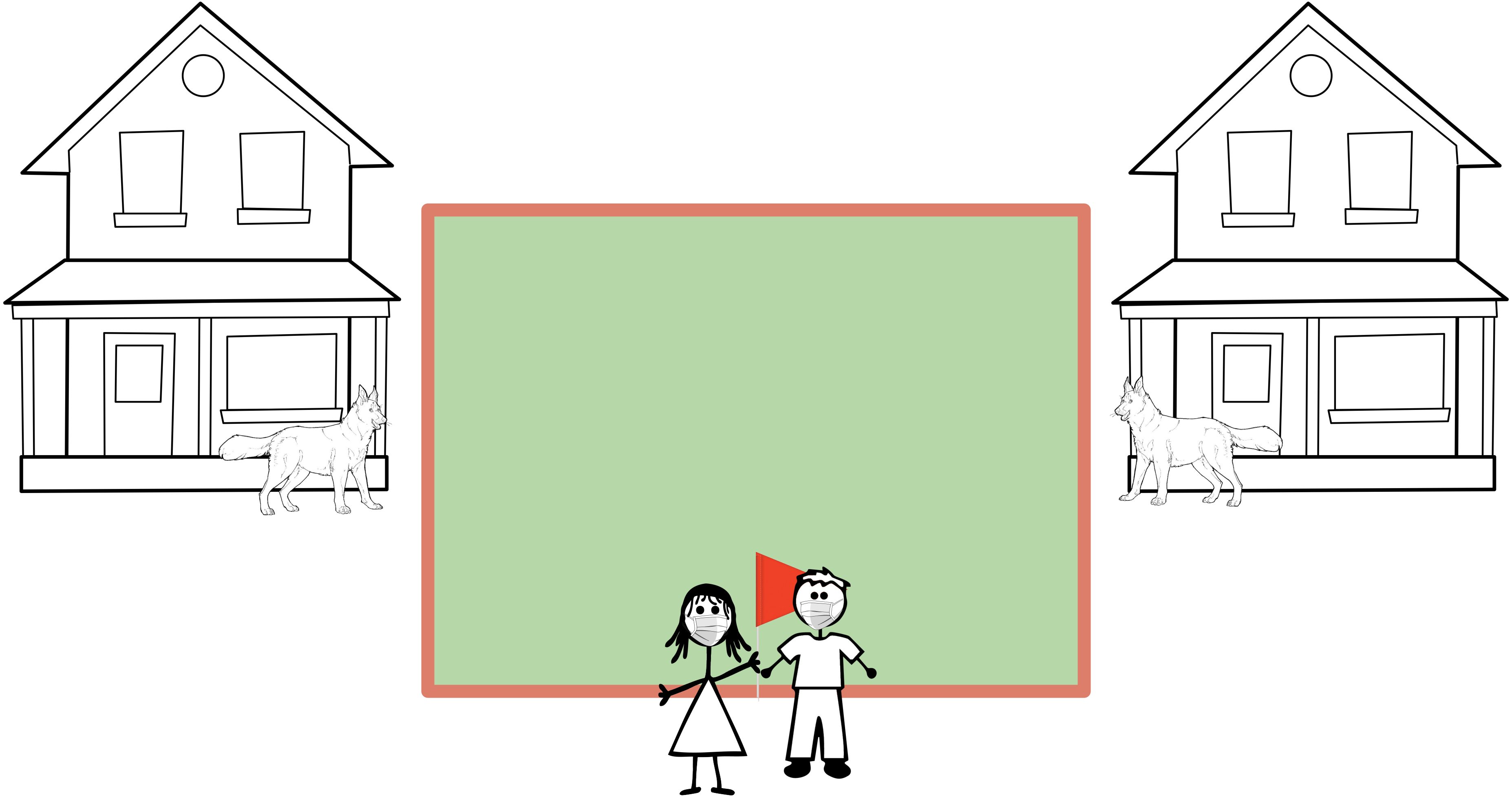


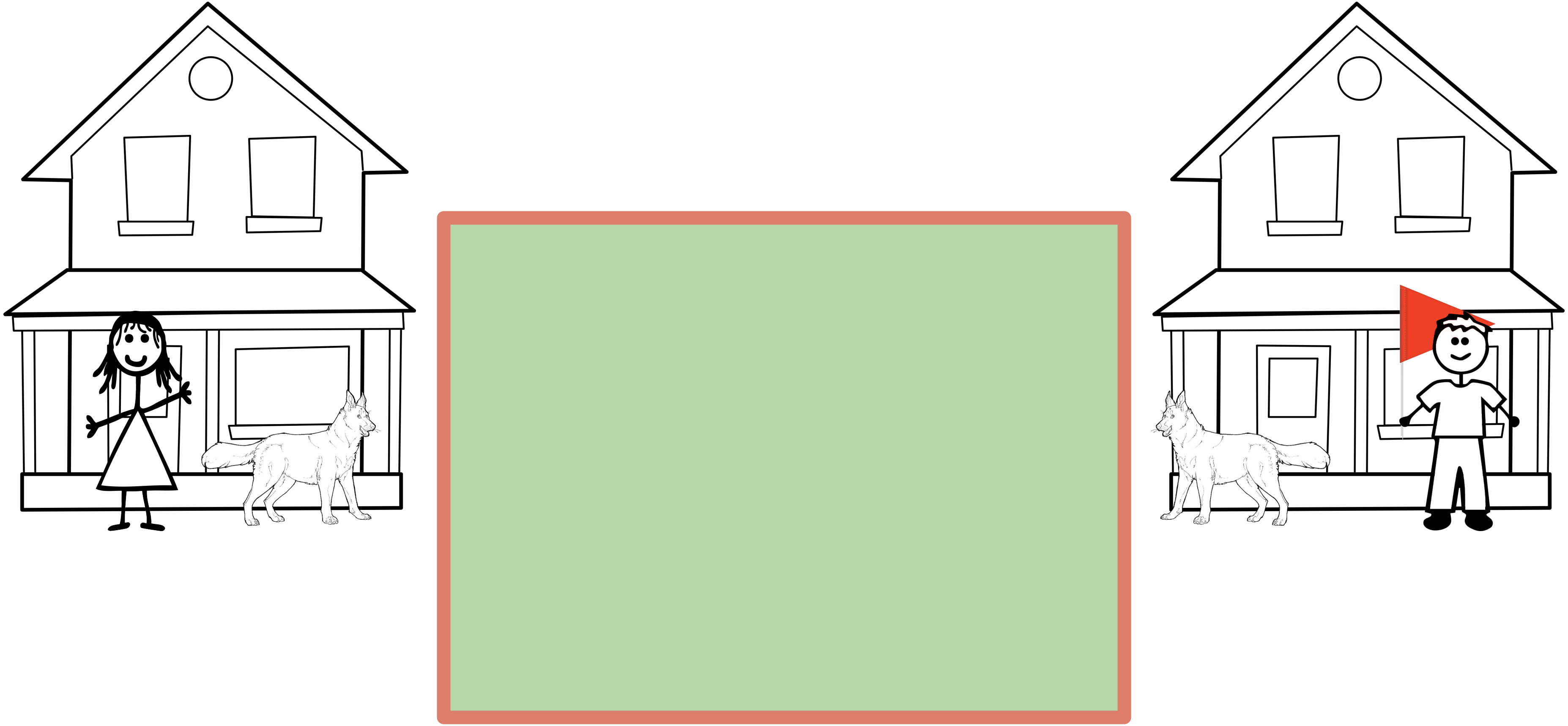


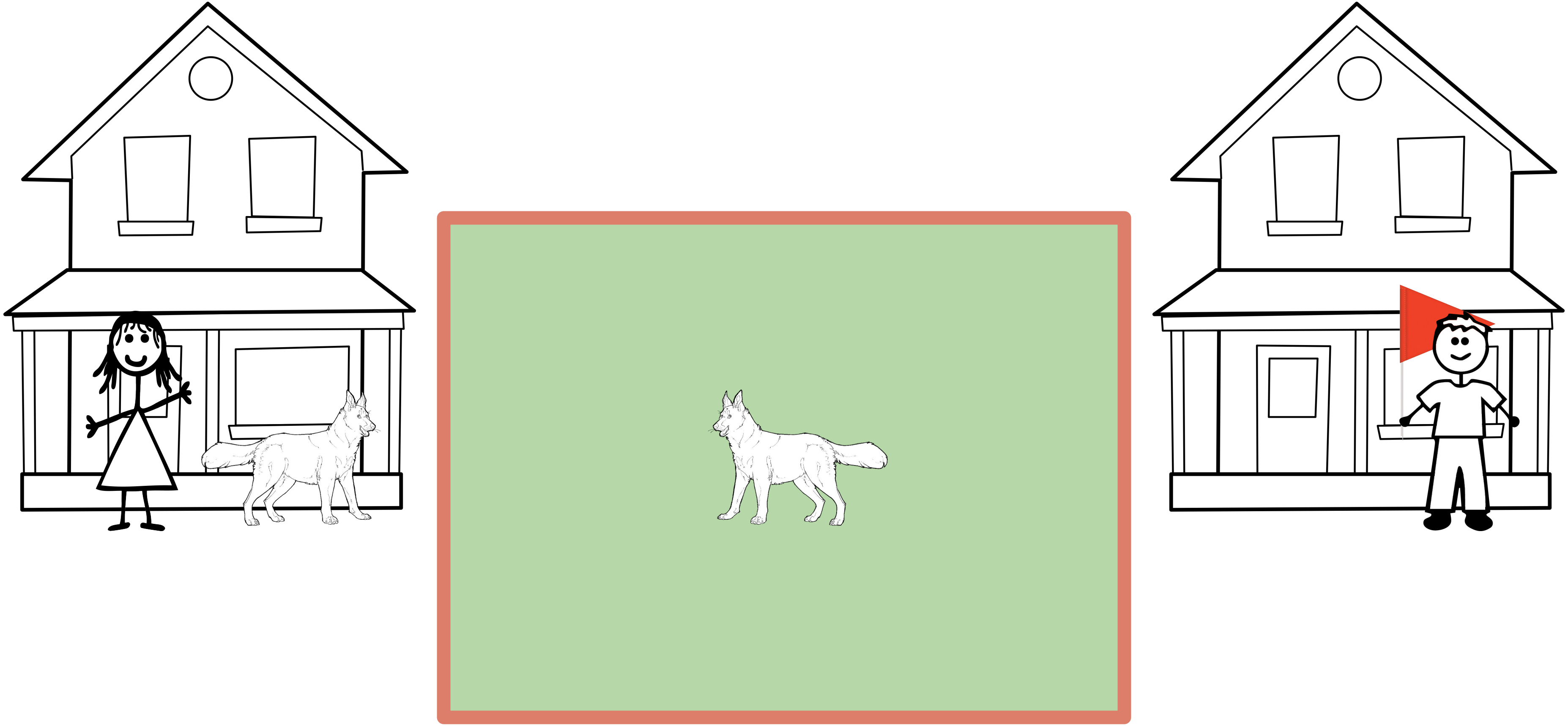


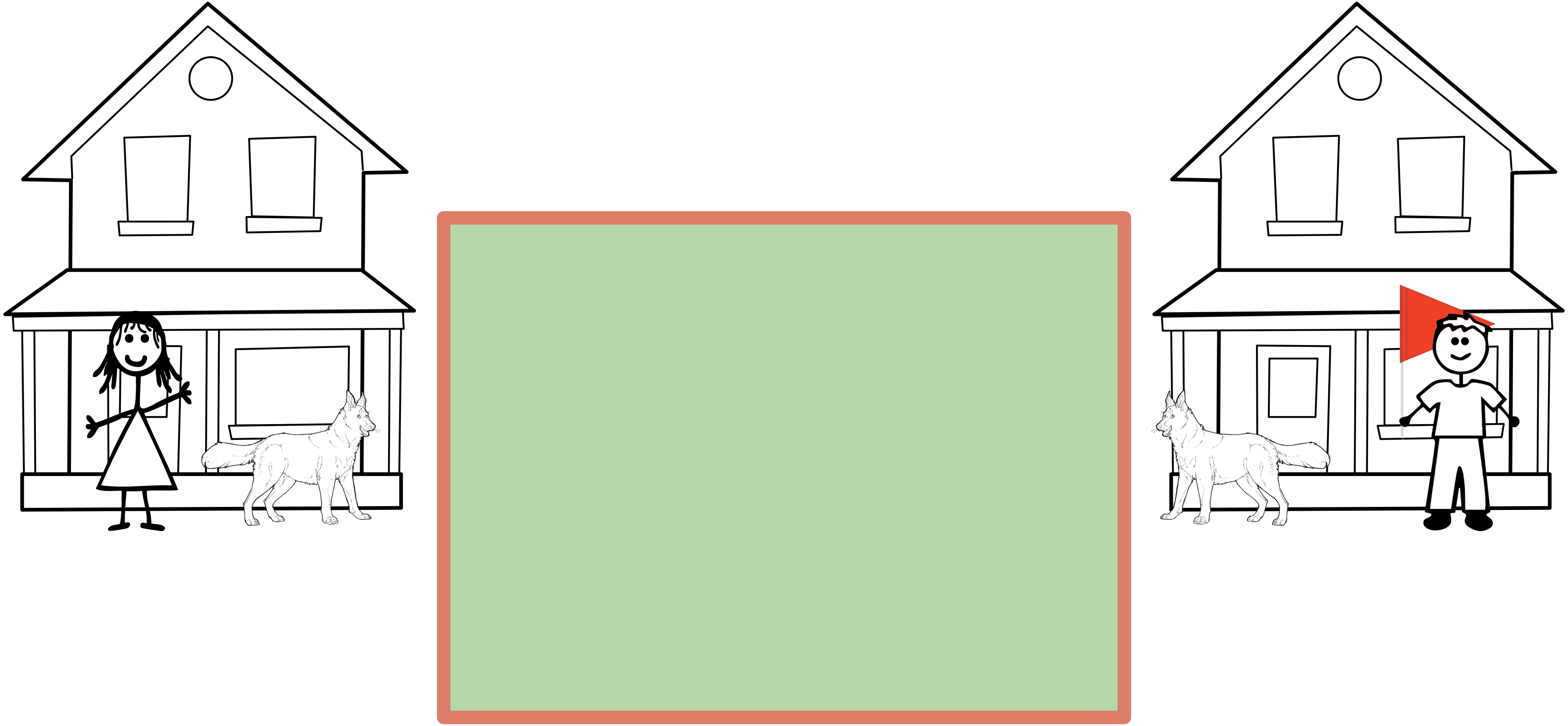


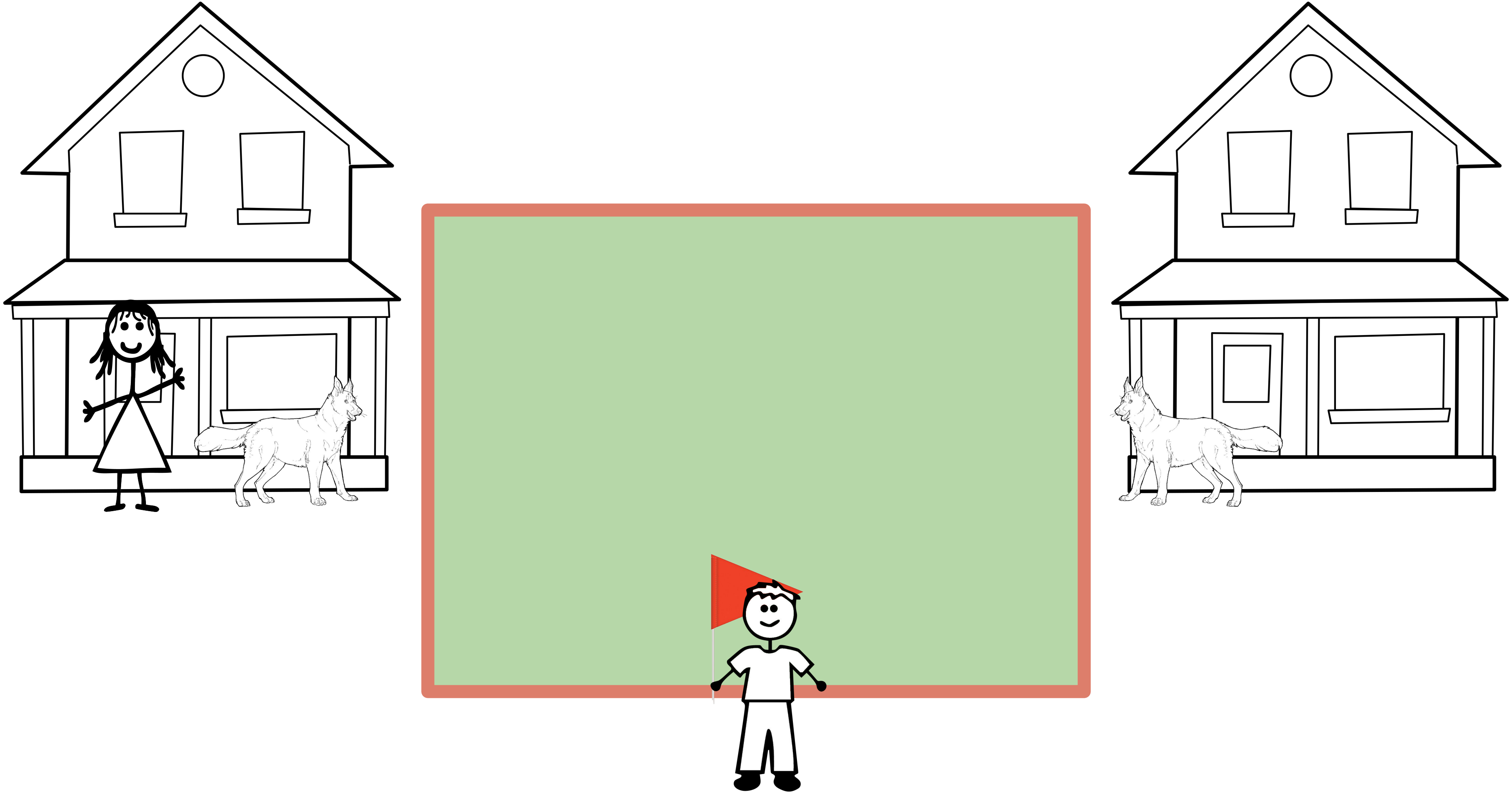


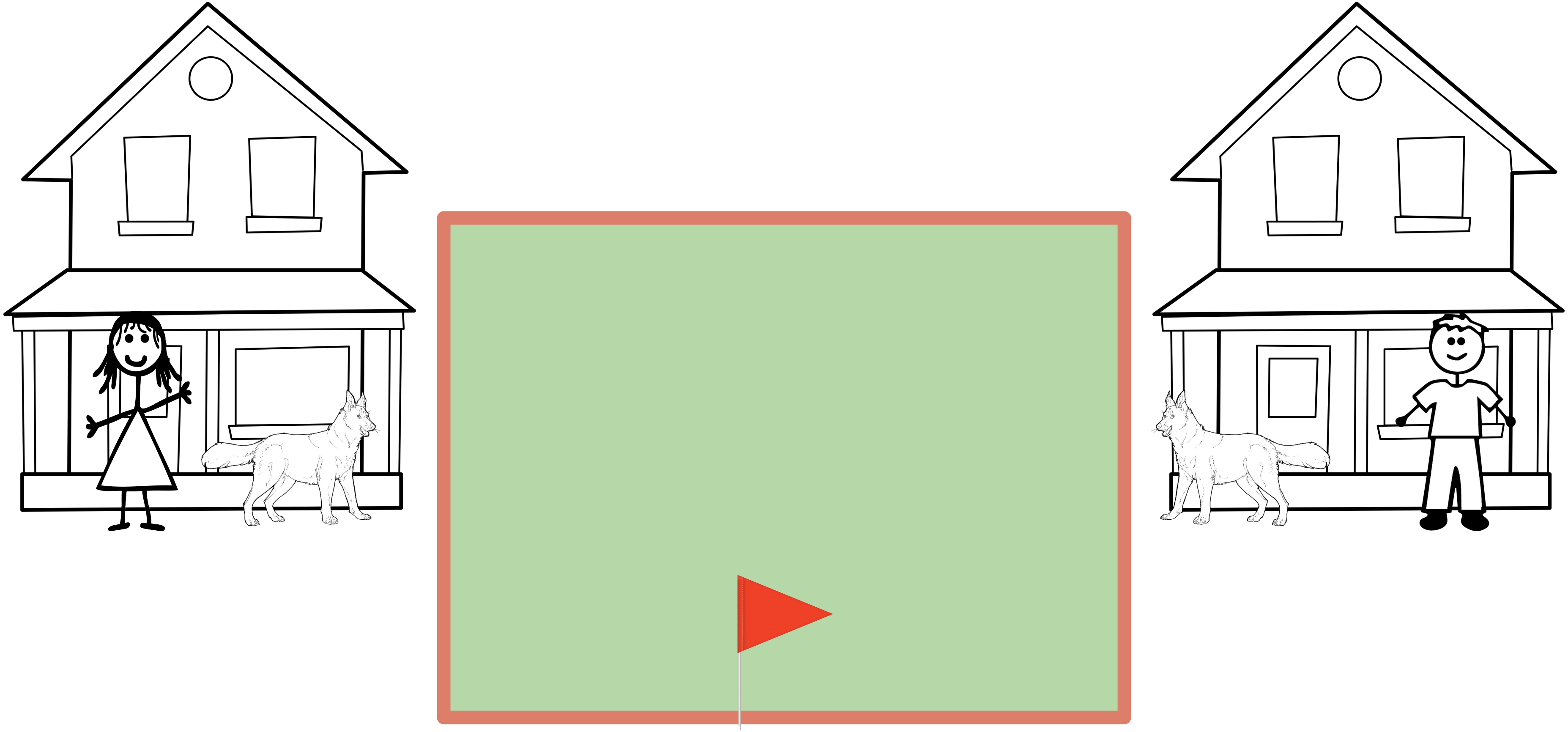


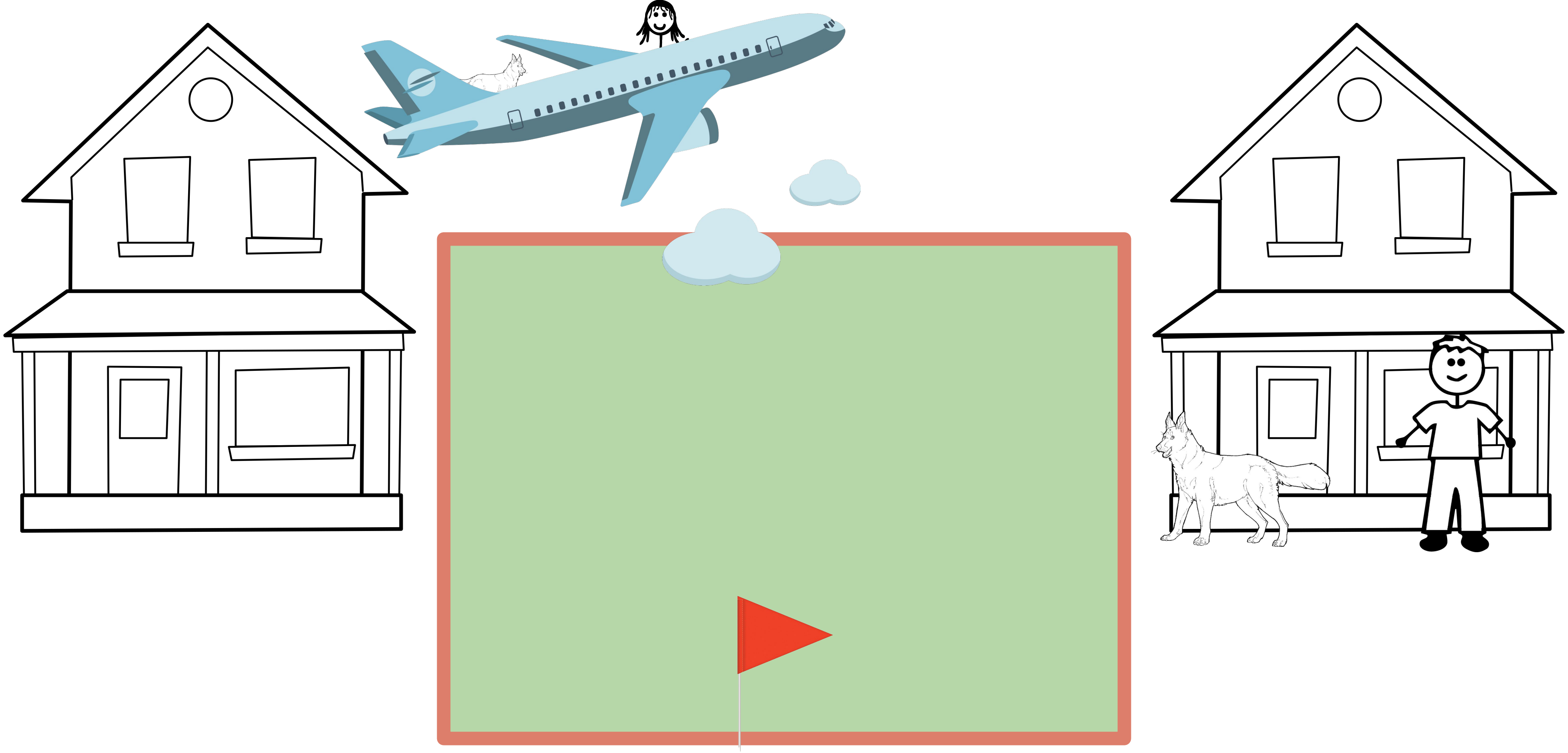


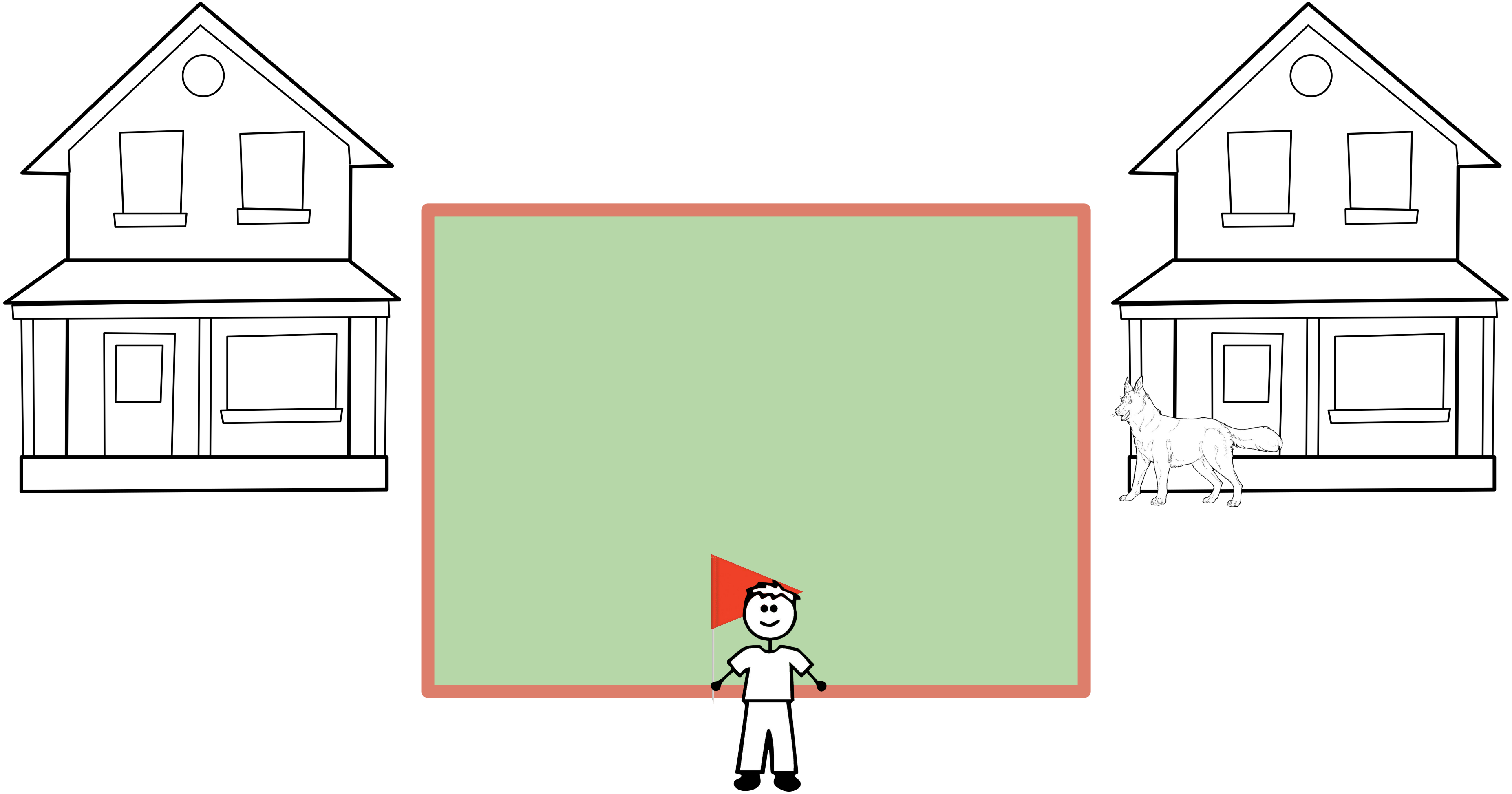


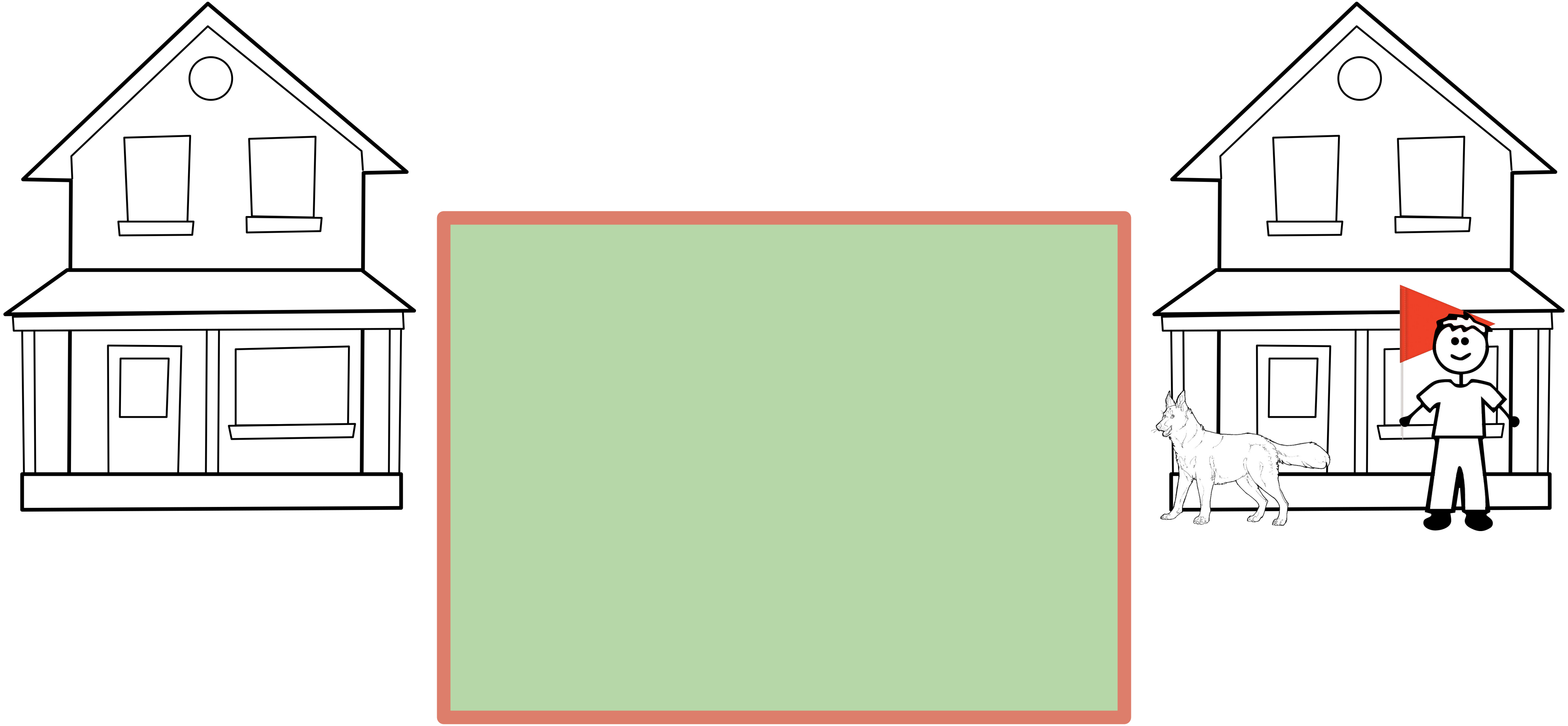


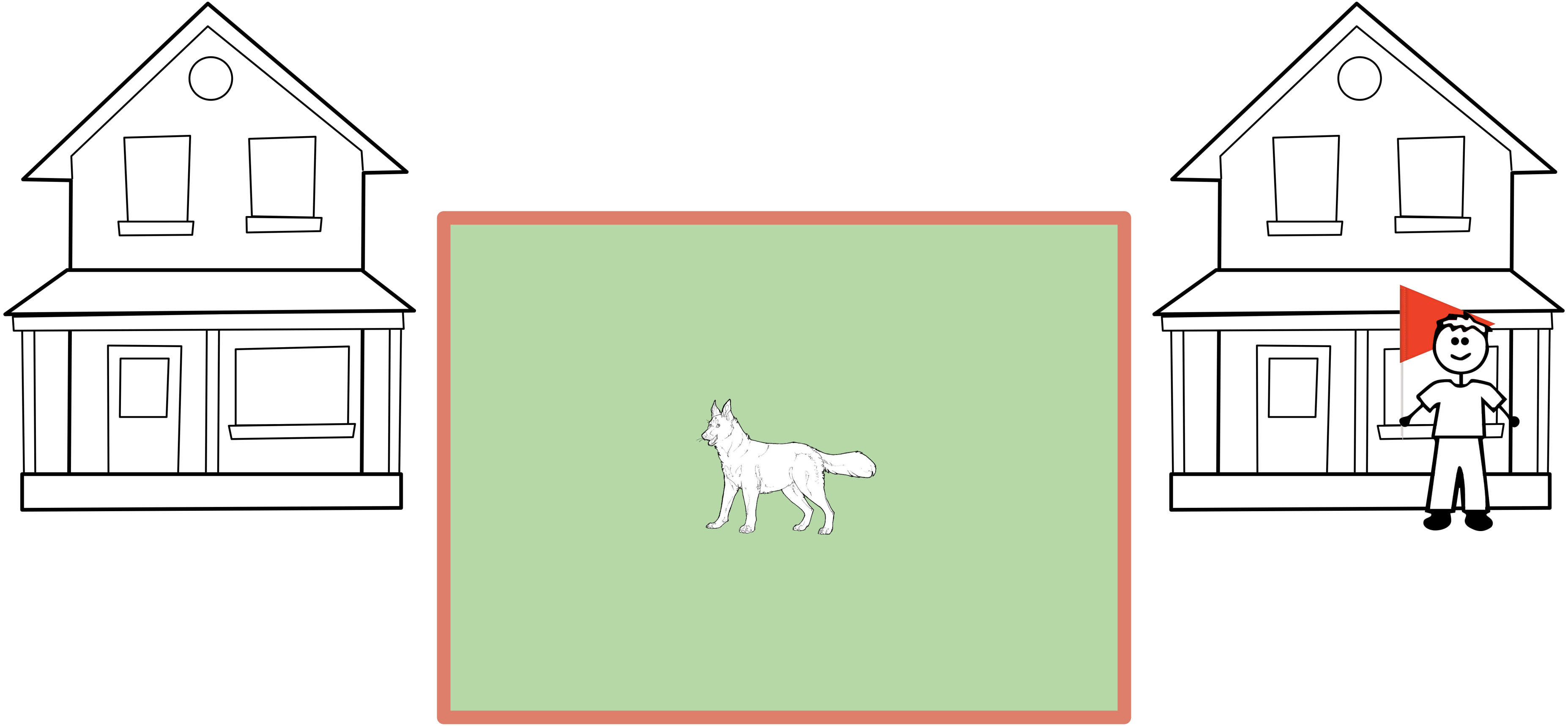


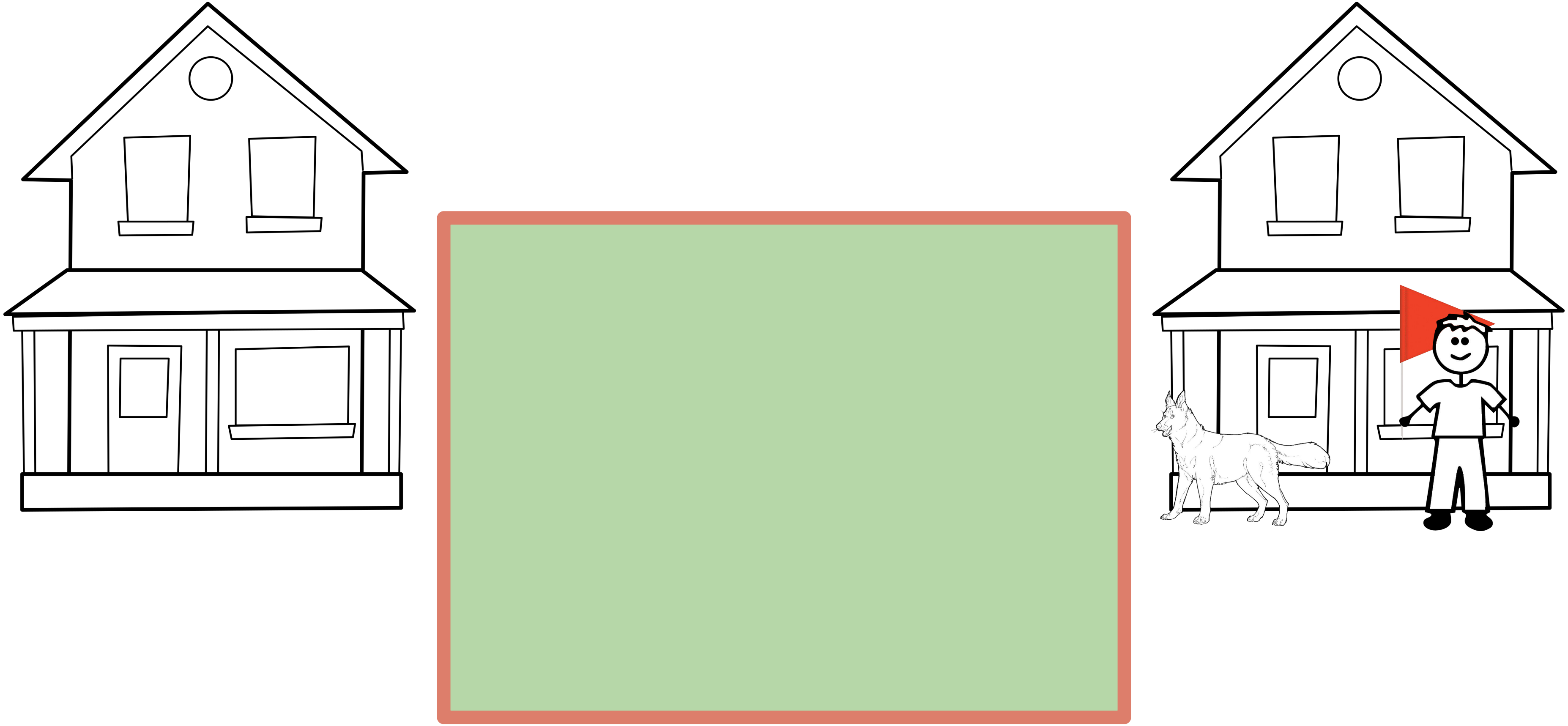


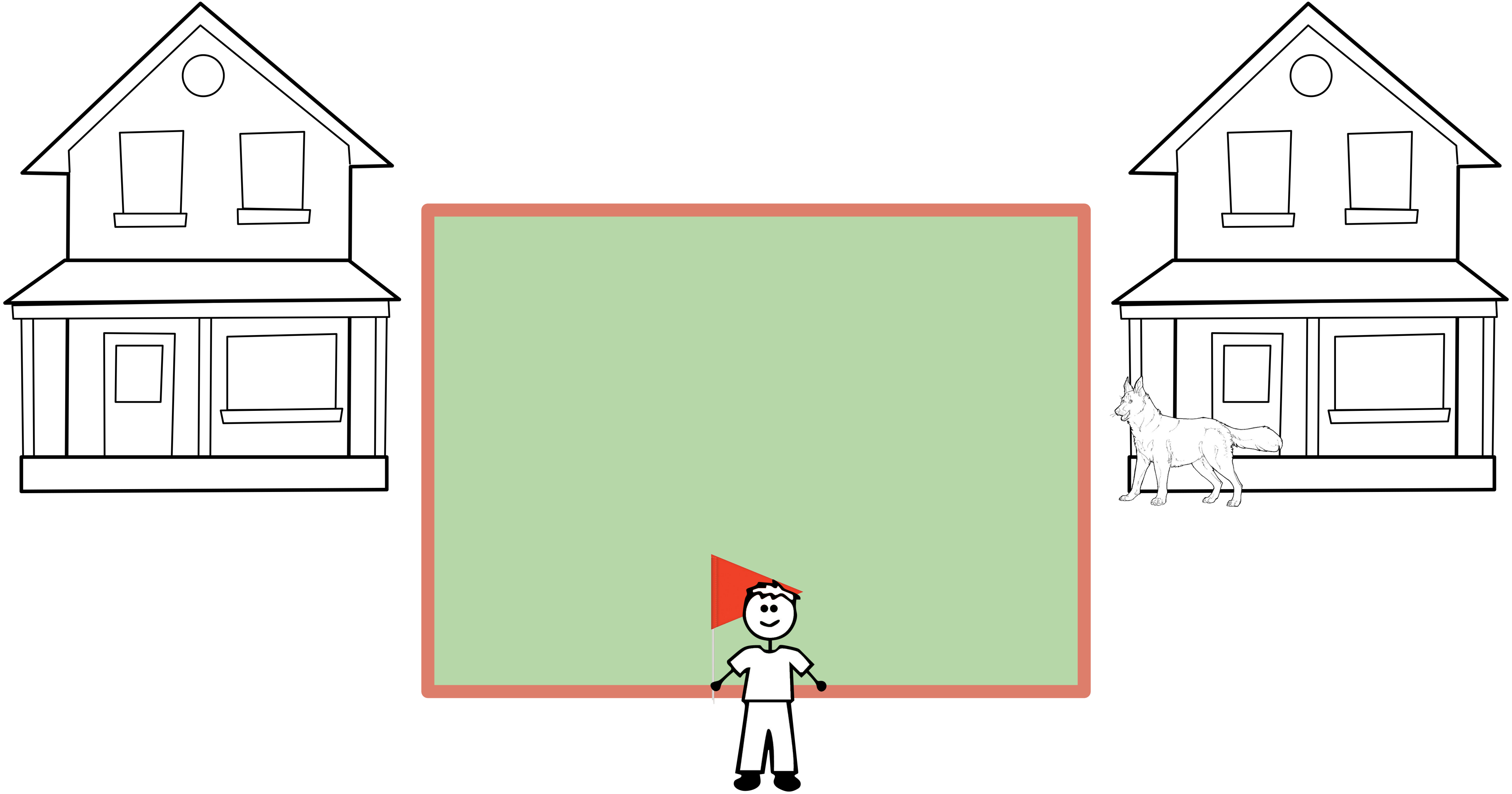


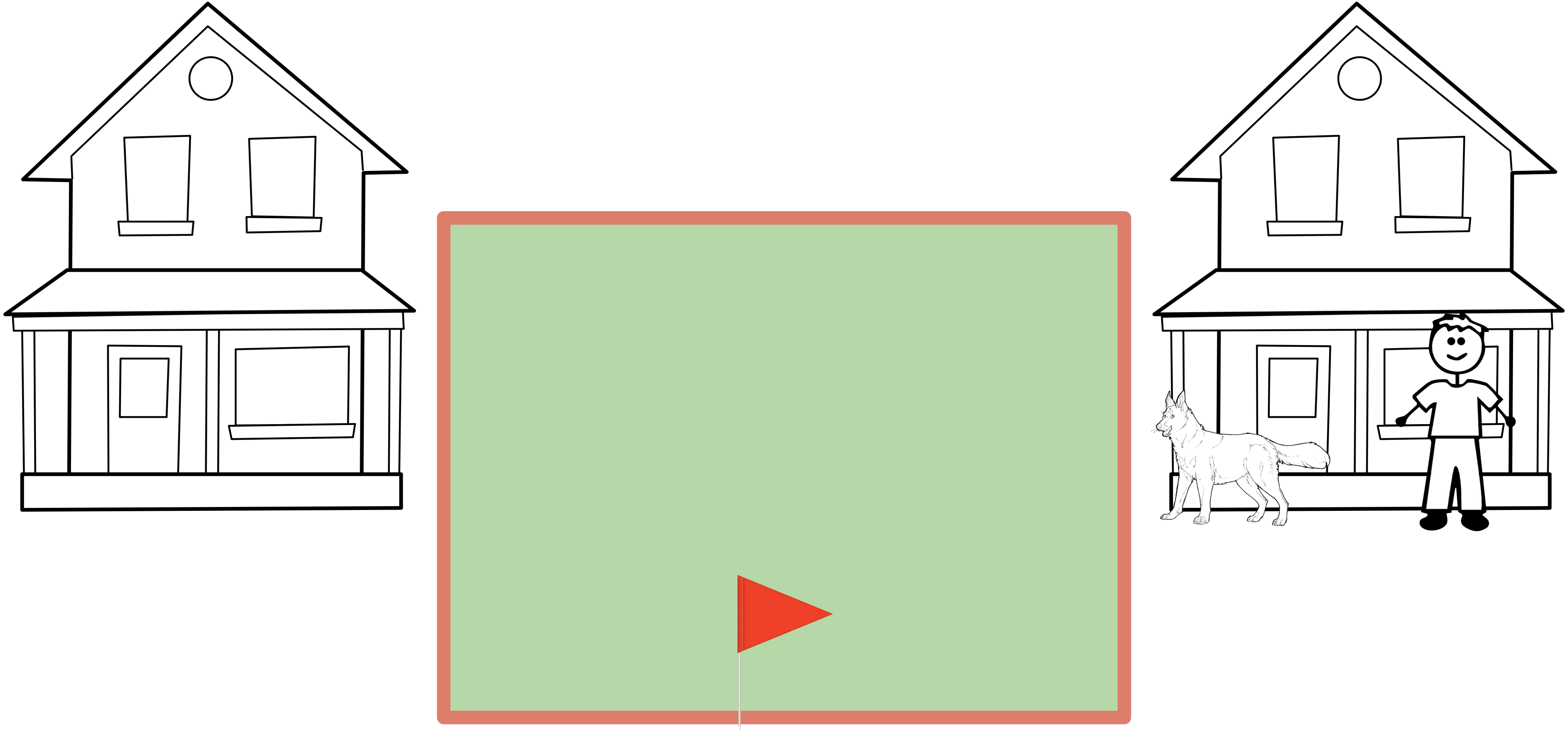


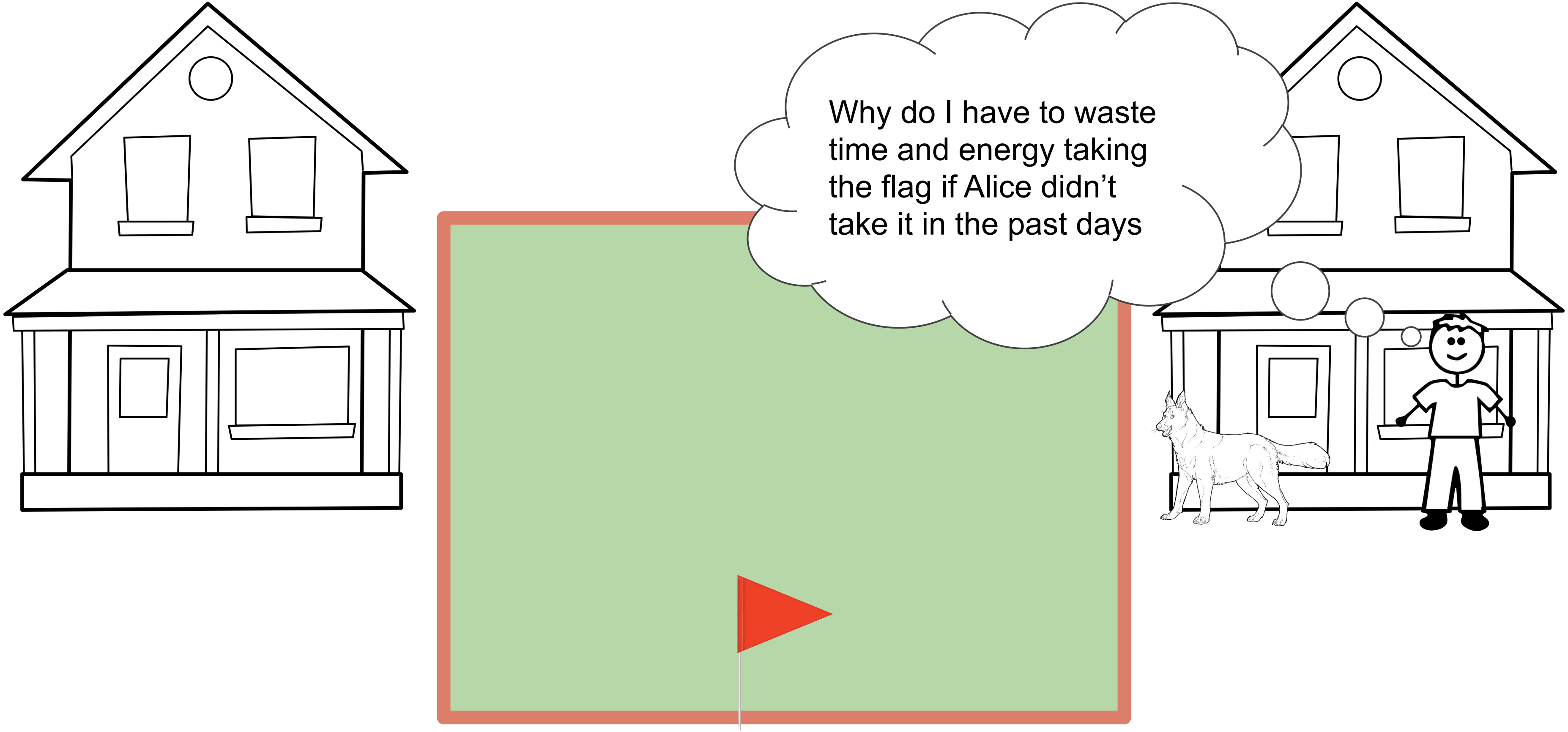











Why do I have to waste time and energy taking the flag if Alice didn't take it in the past days

Problem

- Conservative locking
  + Simple for the developer
  - Performance loss
- Lock granularity
  + Fine grained locking can improve performance
  - Harder for developers to use those techniques

- Thread-unsafe legacy libraries
  - For libraries without multithreading support use a global lock
  - Big performance degradation

Potential parallelism hidden by locks

Thread 1

```
LOCK(hash_tbl.lock)
var = hash_tbl.lookup(X)
if (!var)
    hash_tbl.add(X)
UNLOCK(hash_tbl.lock)
```

Thread 2

```
LOCK(hash_tbl.lock)
var = hash_tbl.lookup(Y)
if (!var)
    hash_tbl.add(Y)
UNLOCK(hash_tbl.lock)
```

Goal of SLE (Speculative Lock Elision)

- Highly concurrent multithreaded execution
    - Multiple threads can access the same critical section
- Simple correct multithreaded code development
    - Programmers can use simpler conservative locking without performance degradation
- Easily implementable
    - Modifications only in microarchitecture

Key Ideas

No read/write conflicts between threads. For other threads the critical section is executed in “an instant”

- Locks often impose **false dependencies**
- **Locks are not the only way** to guarantee atomicity
- **Remove locks** and check for atomicity while executing
- **Buffer result** and commit only if atomicity is preserved

Mechanism

SLE Algorithm

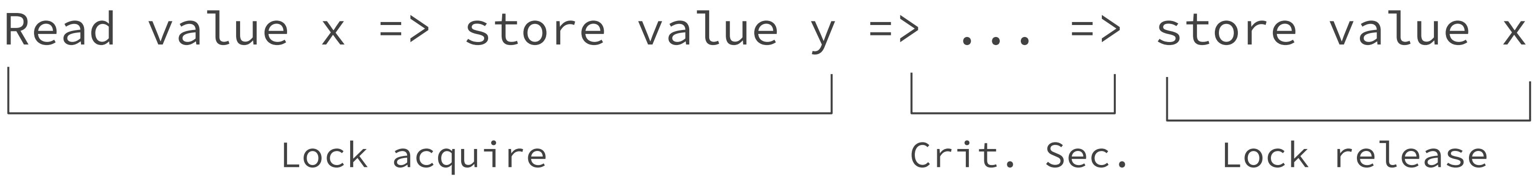
1. **Predict** that memory operation will occur **atomically** in the critical section
2. **Execute** critical section **speculatively** and **buffer results**
3. If **atomicity** cannot be guaranteed **trigger misprediction**
4. **Commit state** and exit speculative critical section

Implementation

Implementation

1. How do we identify where locks are?
2. How can we buffer the state of the execution?
3. What we have to do on mispredictions?
4. How do we commit the state?

1. How do we identify where locks are?



- Read value x => store value y => ... => store value x

Lock acquire

Crit. Sec.

Lock release

- If the critical section occur atomically the second store overwrites the first one

We predict that after the read and store (lock acquire) another store (lock release) will follow

2. How can we buffer the state of the execution?

- Register state (two approaches)

  1. Reorder buffer (ROB)
     - Already implemented for branch misprediction
     - Size of critical section limited by ROB size

  2. Register checkpoint
     - Save architectural state before lock elision in the cache
     - Restore checkpoint on misprediction
     - Limited by cache size

- Memory state
  - Leverage already existing write buffer
  - Add support for speculative stores
  - Speculative data is not committed until lock elision is validated

3. What do we have to do on mispredictions?

1. **Atomicity violations**
   - Can be detected with cache coherence mechanisms
   - The cache blocks must be in the exclusive state

2. **Violations due to limited resources**
   - 2.1. Cache size (Reg checkpoint)
   - 2.2. Write buffer size
   - 2.3. ROB size
   - 2.4. Uncached accesses or events (e.g. some sys calls)

4. How do we commit the state?

- Speculative data is in the write buffer
- The cache blocks are in exclusive state if we read/wrote to that block
- Set the state of the write buffer to the latest
  - This happens in an instant because is only one bit to be set => Atomicity is preserved

Why can we do lock elision?

- No partial updates are made visible to other threads
- The architectural state is the same after the lock release with or without SLE
- If another thread acquires the lock
    - The lock acquire is a write
    - Other speculating threads will observe the write
    - Trigger mispeculation

Methodology

Methodology

- Simulation with SimpleMP derived from Simplescalar
- 3 different multiprocessor (MP) systems:

1. Chip MP (CMP)

   All processors on the same die, fast cache access and inter-processor communication

2. Bus MP (BMP)

   Processors connected by a bus, easily scalable

3. Distributed Shared Memory (DSM)

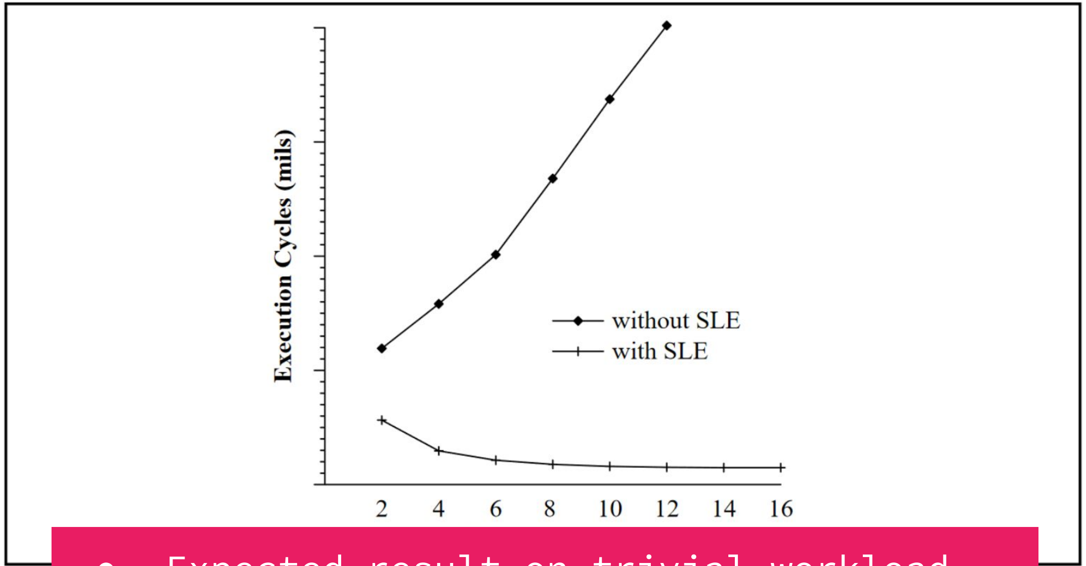
   Every processor have their memory, to share data they have to send it to each other. No race conditions memory is private

- Single register checkpoint
- Restart threshold of 1 (number of retries)

Benchmarks

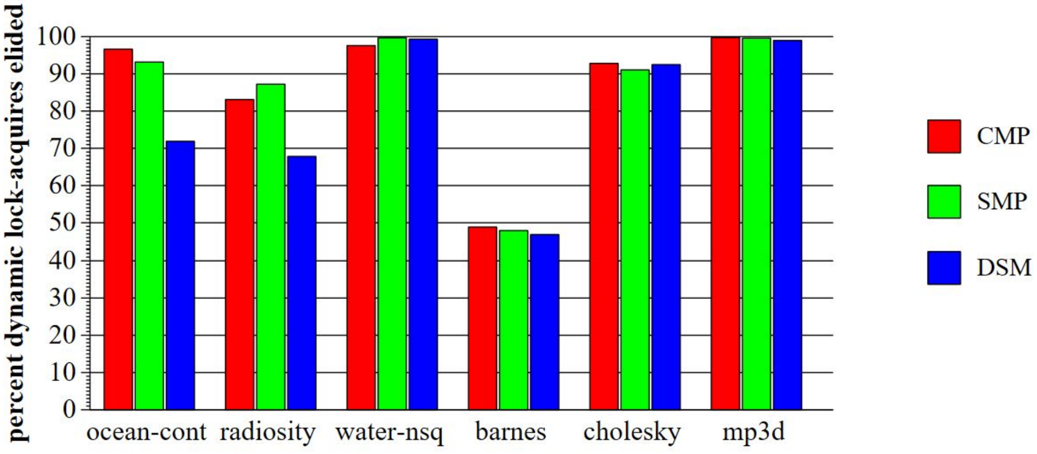
- N threads each incrementing a unique counter all protected by the same lock
- **Barnes:** N-Body, nested cell locks
- **Cholesky:** Matrix factoring, task queues and col. locks
- **Mp3D:** Rarefied field flow, cell locks
- **Radiosity:** 3D rendering, nested task queues
- **Water-nsq:** Water molecules, global structure
- **Ocean-cont:** Hydrodynamics, conditional updates

Simple N threads counting



- Expected result on trivial workload
- Without SLE: more threads worst performance
- With SLE: more threads better performance

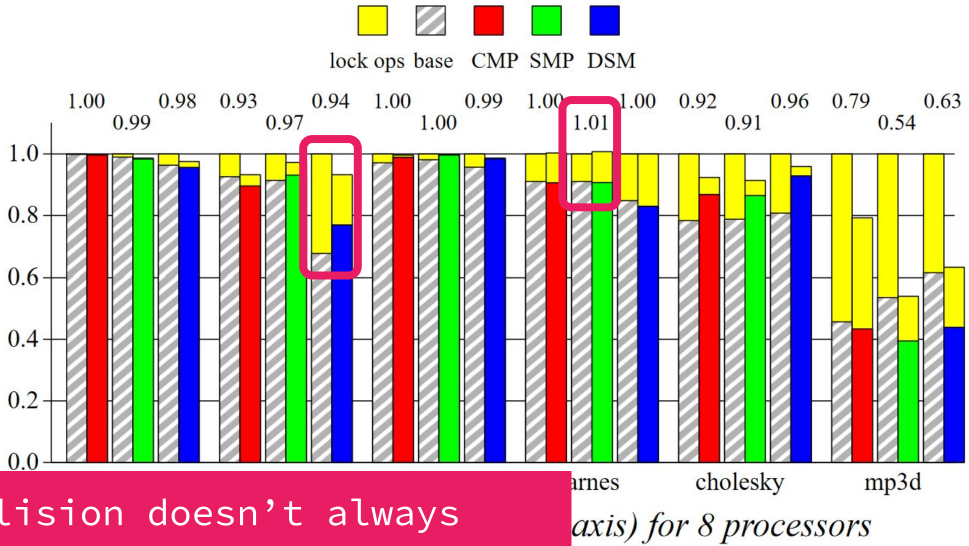
Real world benchmark



a) Percentage for 8 processors

- More than 50% locks were removed
- On average more that 80%

Real world benchmark



- Higher lock elision doesn’t always translate to better speedup
- Sometimes removing locks worsen the performance showing other bottlenecks
- On most workloads we still obtain some speedup

Executive summary

- **Problem:** Easier for the developers to do conservative locking but it comes with performance tradeoffs
- **Goal:** Enable highly concurrent multithreaded execution in a simple way
- **Key Approach**
  - Try to execute critical section without locks
  - If atomicity cannot be guaranteed trigger mispeculation
- **Evaluation:** Test SLE on different kind of multiprocessors
  - More than 50% of the locks were removed
  - Speedup on most workloads on most multiprocessors

Strengths

- Does not require software modifications and developer effort
- Only small changes in microarchitecture required
- The SLE algorithm has negligible performance overhead
- Ahead of his time
  - MP not widely spread
- The idea can be applied to various type of MP systems

Weaknesses

- Naive lock identification
- Optimistic prediction algorithm
- Performance overhead where no locks present not evaluated
- It is unclear what happens when we mispredict that a store is a lock
- Possible security vulnerability
  - Read buffer of speculative execution
- Additional hardware cost not explained

A bit of history

- 2001 (When this paper was presented)
    - Intel Pentium 4 (1 core, 2.8 GHz, 512 KB L2 cache)
    - IBM Power 4 (2 core, 1.9 GHz, 1.4 MB L2 cache)
        - First commercially available multiprocessor chip
- June 2013: Intel processors introduced TSX with HLE (Hardware Lock Elision)
    - New instruction to support transactional memory
- October 2014: AMD proposed ASF (Advanced Synchronization Facility)
- 2019: Intel removed HLE (CVE-2019-11135)
- 2020: Intel removed TSX on 10th gen. CPUs
- AMD never implemented ASF on any production CPUs

Thoughts and ideas

- Paper ahead of his time
- Developers were not used to doing parallel optimized programming
- Probably not very useful when using a framework that is already optimized for parallel execution
- The security risk was not negligible
  - The actual performance improvement was not great

Questions?

Do you agree with the choice of Intel in removing TSX because of a security risk?

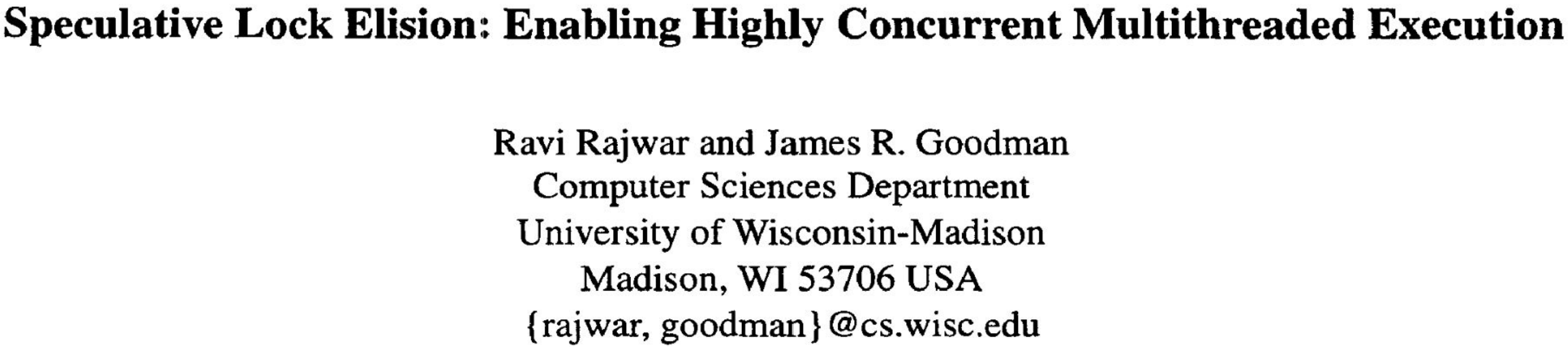
Instead of an only hardware approach where are other approaches better/worse?

Hardware only vs. software only vs. compiler assisted

Can we apply the same idea to different lock types? (e.g. barriers, semaphore)

Can we elide different instruction(s) using the same mechanism?

For example silent pairs that are not lock acquire/release



Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution

Ravi Rajwar and James R. Goodman
Computer Sciences Department
University of Wisconsin-Madison
Madison, WI 53706 USA
{rajwar, goodman}@cs.wisc.edu