Formal Verification of a Mixed-Trust Synchronization Protocol

RTNS'2021

Ruben Martins, Michael McCall, Dionisio de Niz,
Amit Vasudevan, Bjorn Andersson, Mark Klein,
John P. Lehoczky, Hyoseung Kim

Copyright 2021 ACM.

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

This material may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other use. Requests for permission should be directed to the Software Engineering Institute at permission@sei.cmu.edu.

Carnegie Mellon® is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

DM21-0245

Motivation

- Cyber-Physical Systems (CPS) are used in many safety-critical applications
- Verifying these complex systems is a challenging problem
- Current verification techniques do not scale to complex components

Problem: How to verify every component and their interaction?

Motivation

- Cyber-Physical Systems (CPS) are used in many safety-critical applications
- Verifying these complex systems is a challenging problem
- Current verification techniques do not scale to complex components

Solution: Do not verify every component but instead use a ***Mixed-Trust Framework:***

- Untrusted components interact with trusted (verified) components
- Trusted components guarantee timing and functional correctness of the entire system

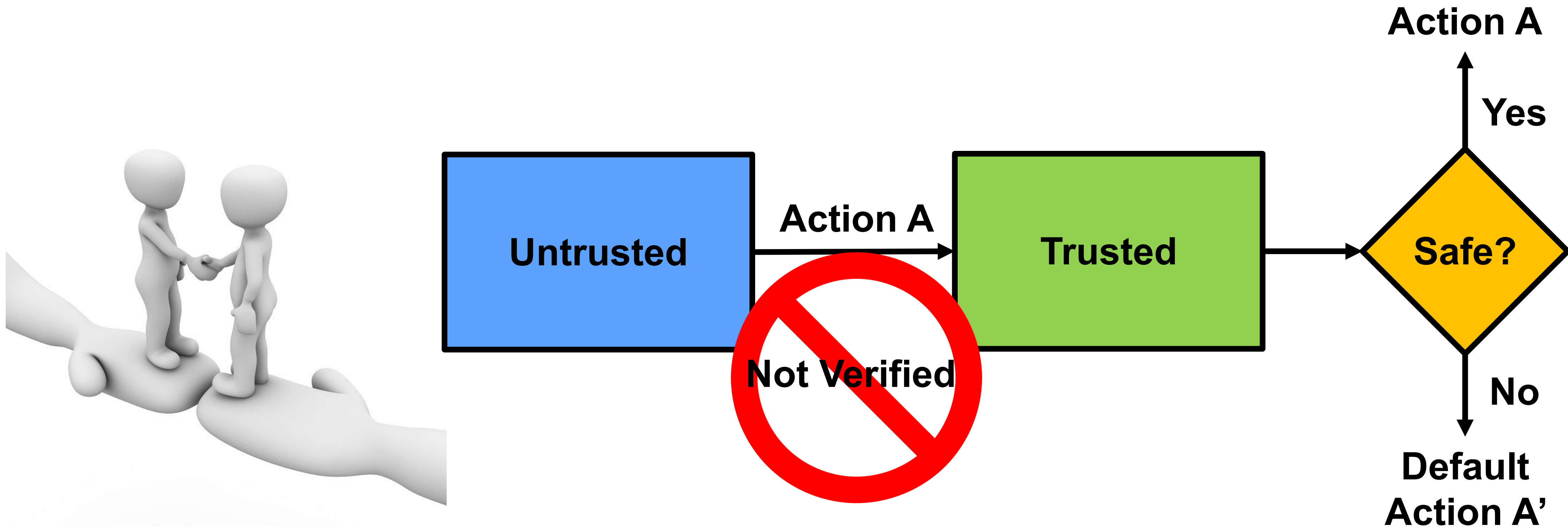
Mixed-Trust Framework

Untrusted
Action A
Trusted
Safe?
Yes
Action A
No
Default Action A'

- Trusted (verified) components monitor untrusted components:
  - Logical guarantees: action is safe
  - Timing guarantees: action is performed on time

- Trusted components are known as **enforcers** and are protected by a verified micro-hypervisor

Mixed-Trust Framework

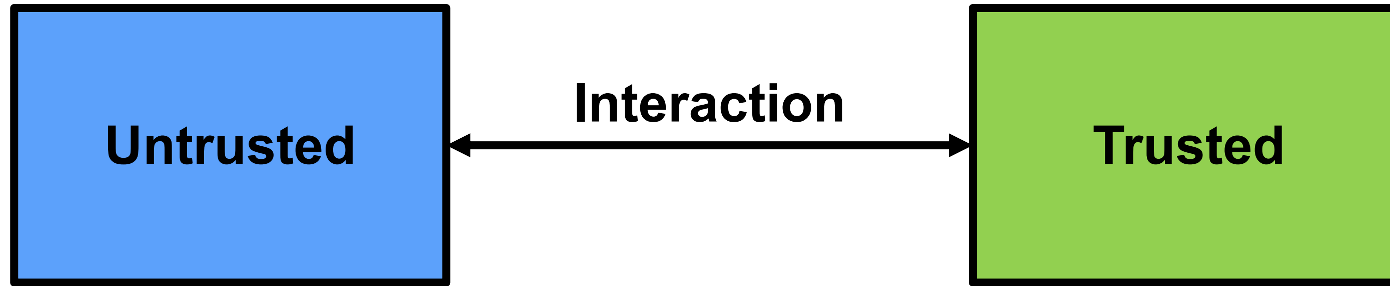


Mixed-trust frameworks are great but:

- **Problem:** Even if trusted components are verified, the system can still be compromised if the *interaction between untrusted and trusted components is not verified*!

- **This talk:** We focus on *timing guarantees* enforced by the trusted components

Mixed-Trust Synchronization

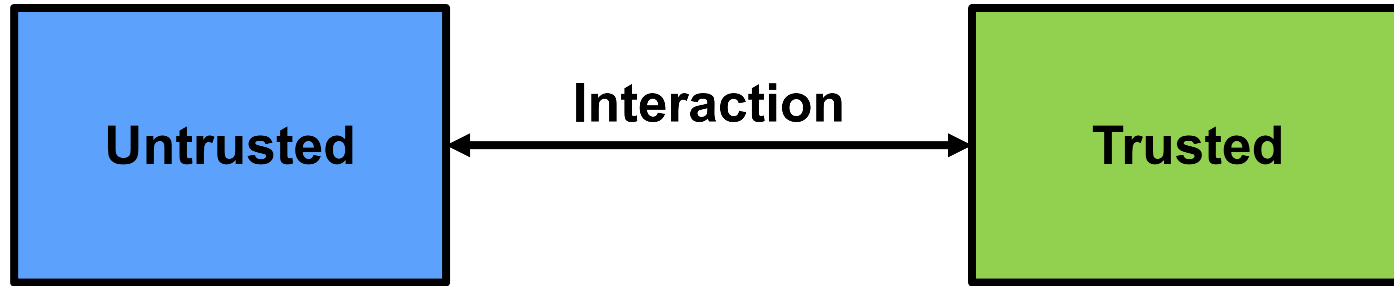


jobComputation()

waitForNextPeriod()

- Periodic tasks:
  - Each task has an untrusted component guarded by an enforcer
- Scheduling scheme:
  - Guest Task (GT) – untrusted
  - Hyper Task (HT) - trusted

Mixed-Trust Synchronization



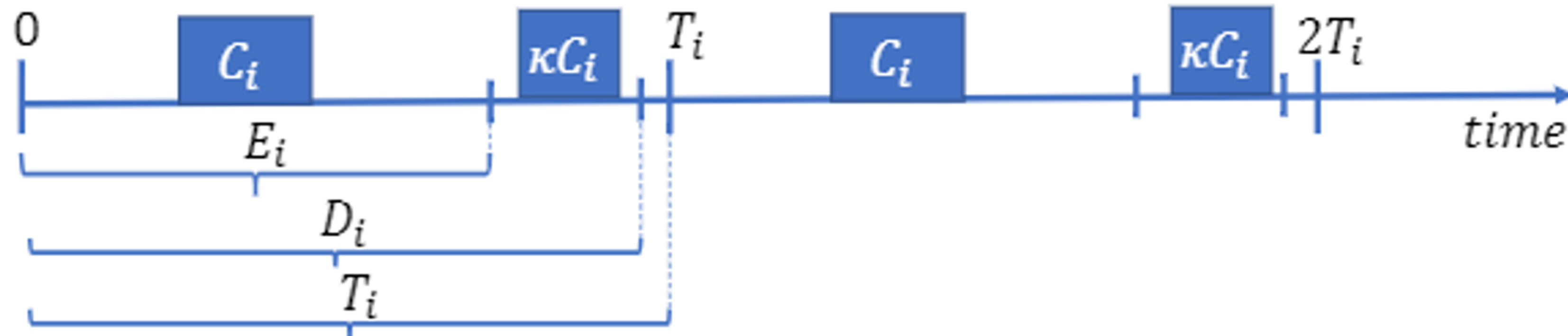
jobComputation()

waitForNextPeriod()

Hyper Task (trusted) ensures that:

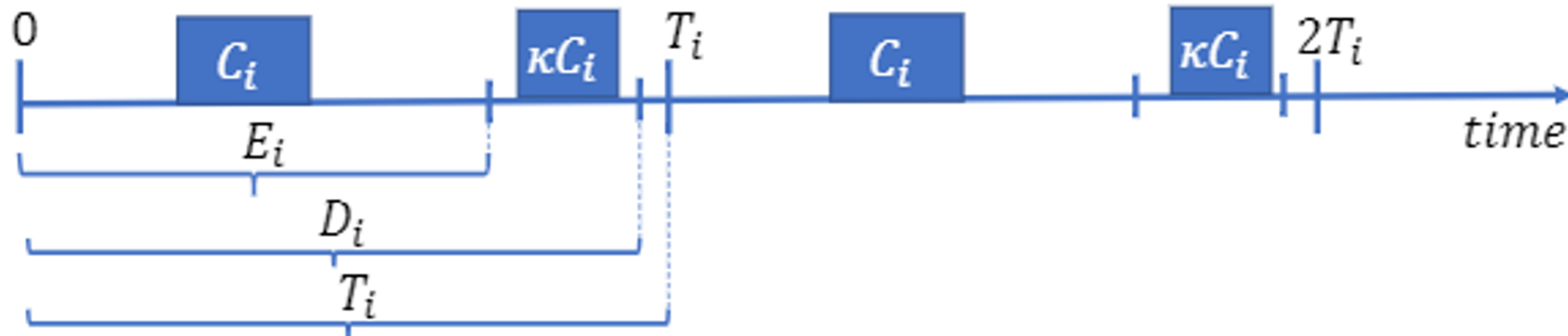
- There is an output at the end of each period:
  1. If untrusted task **finishes on time**:
     - Use output of untrusted task
  2. If untrusted tasks **does not finish on time**:
     - Use default output of trusted task

Mixed-Trust Synchronization



Acronym	Definition
C_i	Worst-case execution time (WCET) of Guest task (GT)
E_i	Deadline for the GT
kC_i	WCET of Hyper task (HT)
D_i	Deadline for the HT
T_i	Time of the period

Mixed-Trust Synchronization



E_i = parameter that can be computed by schedulability testing such that:

- Guest task has enough time to finish (finishes by E_i)
- Hyper task has enough time to finish (finishes by D_i)

More details on schedulability can be found in *de Niz et al. (RTCSA 2019)*

Mixed-Trust Synchronization

To ensure the correct timing behavior the following conditions must be enforced:

Conditions
C1: Each mixed-trust task must produce an output every period
C2: There is only one output per period
C3: The output produced by a task in a period is either from the guest task or the hyper task
C4: A guest output must be the product of a computation that executes within a single period
C5: The hyper task must execute E times units after the arrival of the job it guards and finishes before the end of the period

Note: Only **C5** (schedulability) was formally proven before!

Mixed-Trust Synchronization

In this talk:

- We address the gap in prior work where conditions C1-C4 were not formally proven
- Additionally, we also prove a new safety condition C6 that ensures progress:

C6: If the guest task finishes before the deadline, then the output of this task is used as the output of the mixed-trust task

- C6 with C4 (guest output is the product of a computation in a single period) guarantees that, if there is a **valid run of a guest task**, then that **output is used** instead of the hyper task.

Timing Semantics

- **Guest job (GJ):** a job run by the guest task
- **Hyper job (HJ):** a job run by the hyper task

- **Valid guest job:**
  - The job started after the beginning of the current period and finished before the deadline for the guest task
  - If more than one guest job was run in the same period, we consider as valid the first job

Period (1,start)			Deadline (GT)		Period (1, end)	Period (2, start)			Deadline (GT)		Period (2, end)
	Valid GJ										

Timing Semantics

- **Guest job (GJ):** a job run by the guest task
- **Hyper job (HJ):** a job run by the hyper task

- **Valid guest job:**
  - The job started after the beginning of the current period and finished before the deadline for the guest task
  - If more than one guest job was run in the same period, we consider as valid the first job

Period (1,start)			Deadline (GT)		Period (1, end)	Period (2, start)			Deadline (GT)		Period (2, end)
	Invalid GJ										

Timing Semantics

- **Guest job (GJ):** a job run by the guest task
- **Hyper job (HJ):** a job run by the hyper task

- **Valid guest job:**
  - The job started after the beginning of the current period and finished before the deadline for the guest task
  - If more than one guest job was run in the same period, we consider as valid the first job

Timing Semantics

- **Guest job (GJ):** a job run by the guest task
- **Hyper job (HJ):** a job run by the hyper task

- **Valid execution:**
  - The hyper task:
    - only runs one hyper job in the same period
    - always terminates before the end of the period
  - If there are **no valid** guest jobs:
    - Output of the period is the **output** of the **hyper task**
  - If there is a **valid** guest job:
    - Output of the period is the **output** of the **guest task**

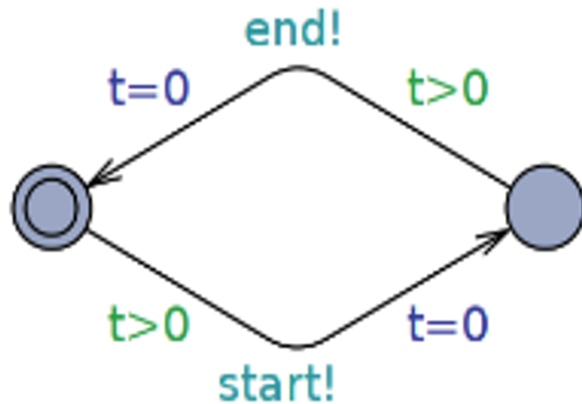
Modeling the Synchronization Protocol

- We will use UPPAAL to model this synchronization protocol and prove the correctness properties

- UPPAAL:
    - Modeling language is an extension of timed automata
    - Bounded discrete variables for additional state information
    - Broadcast synchronization channels

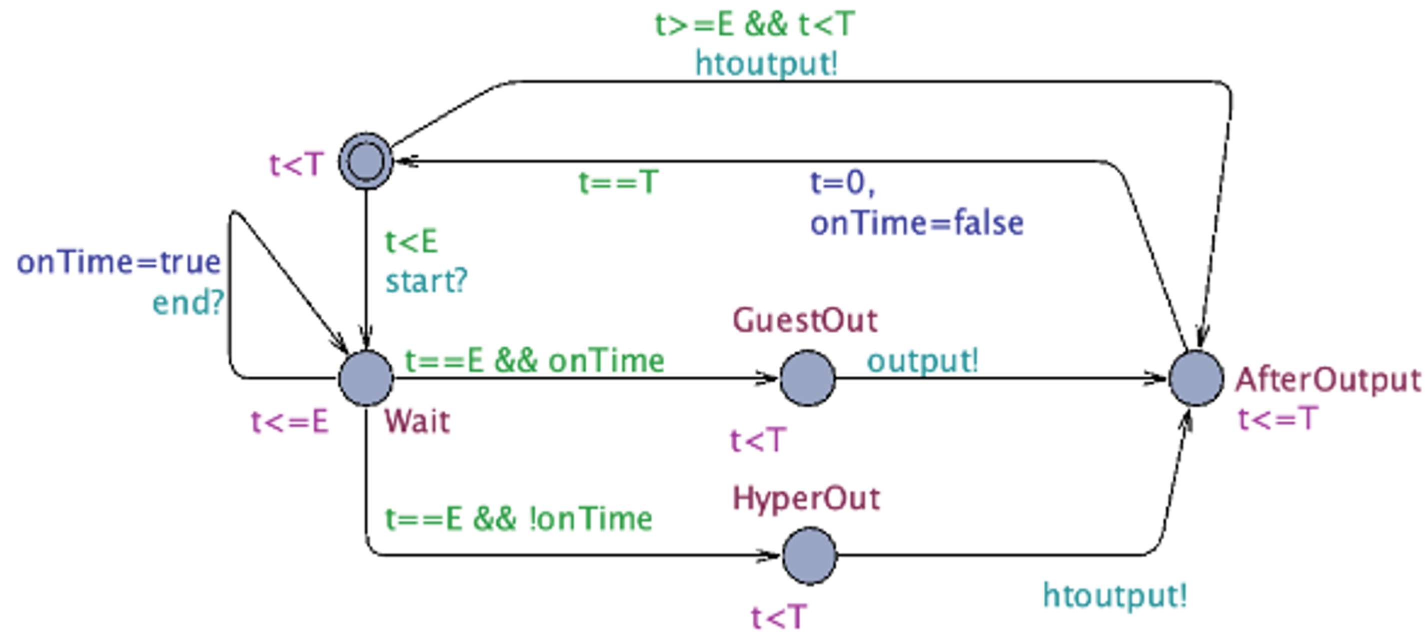
- Timed automaton:
    - Finite-state machine extended with clock variables where all clock variables progress synchronously

Guest Task Automaton



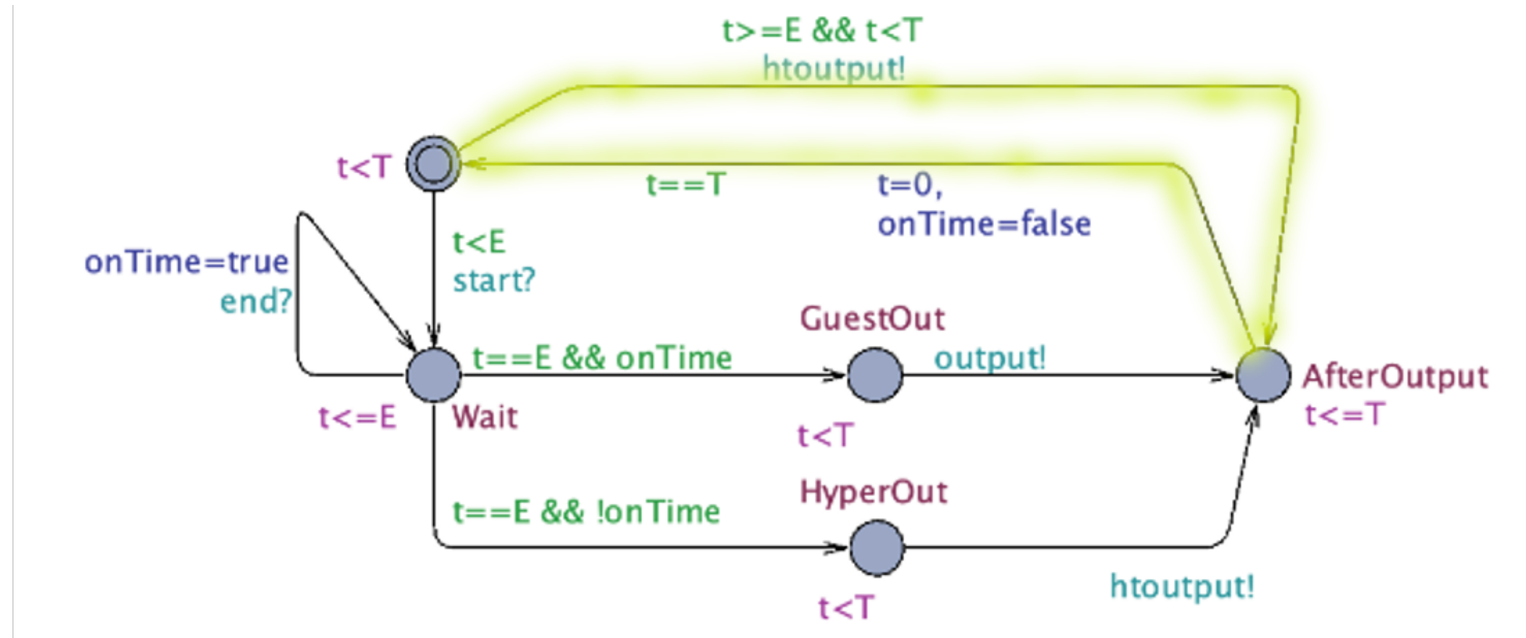
- We do not trust the guest task
- We model its behavior with an automaton that represents all possible behaviors
- Edges:
  - Pre-condition (green): e.g., t>0
  - Post-condition (blue): e.g., t=0
- Broadcast channels:
  - start!
  - end!

Hyper Task Automaton



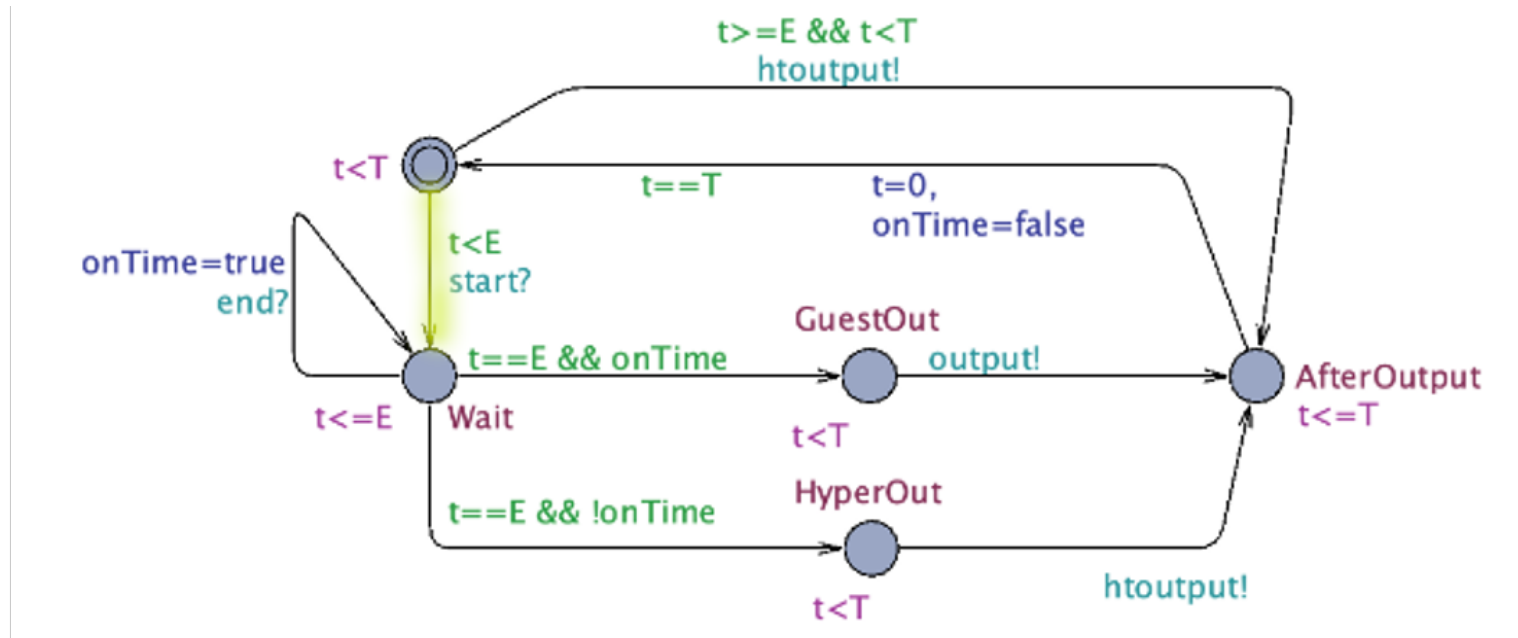
- ***Output* signal** is broadcast if the guest's job execution is **valid**
- ***Htoutput* signal** is broadcast if the guest's job execution is **invalid**

Hyper Task Automaton



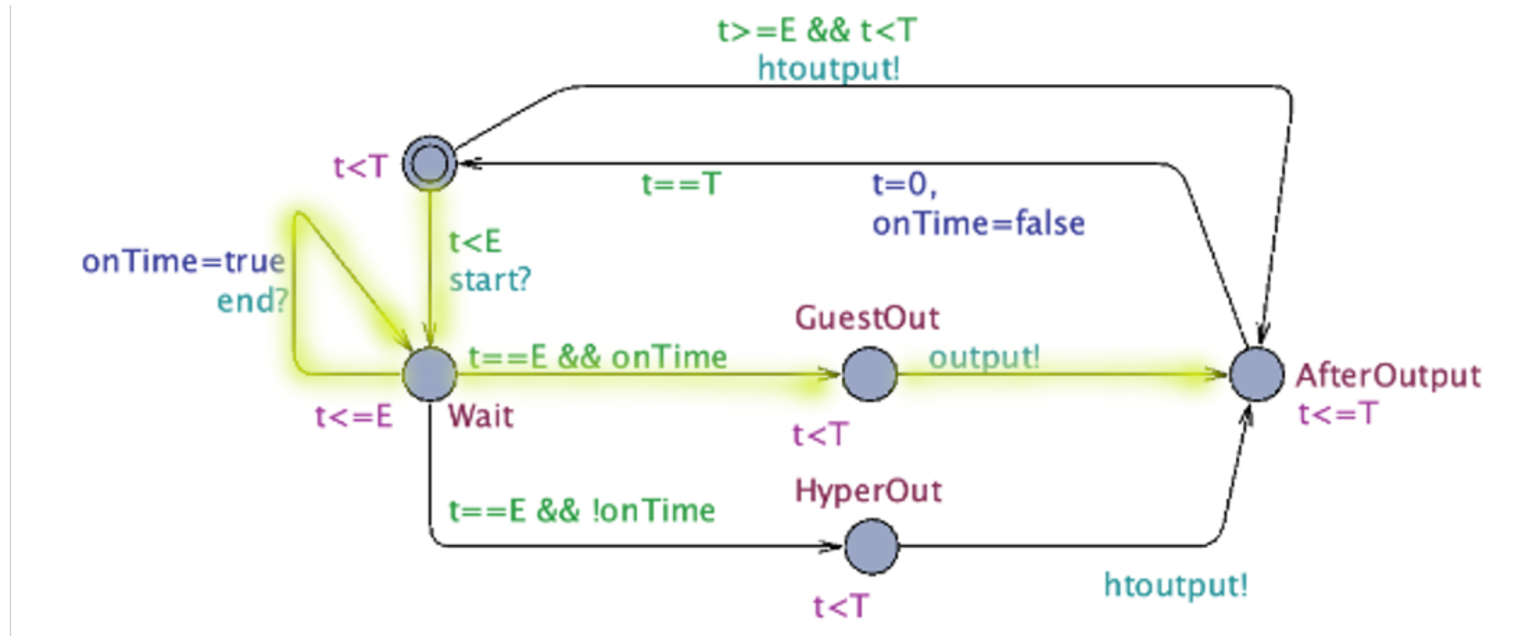
- ***Output* signal** is broadcast if the guest’s job execution is **valid**
- ***Htoutput* signal** is broadcast if the guest’s job execution is **invalid**
- Case 1:
  - The start signal is never received

Hyper Task Automaton



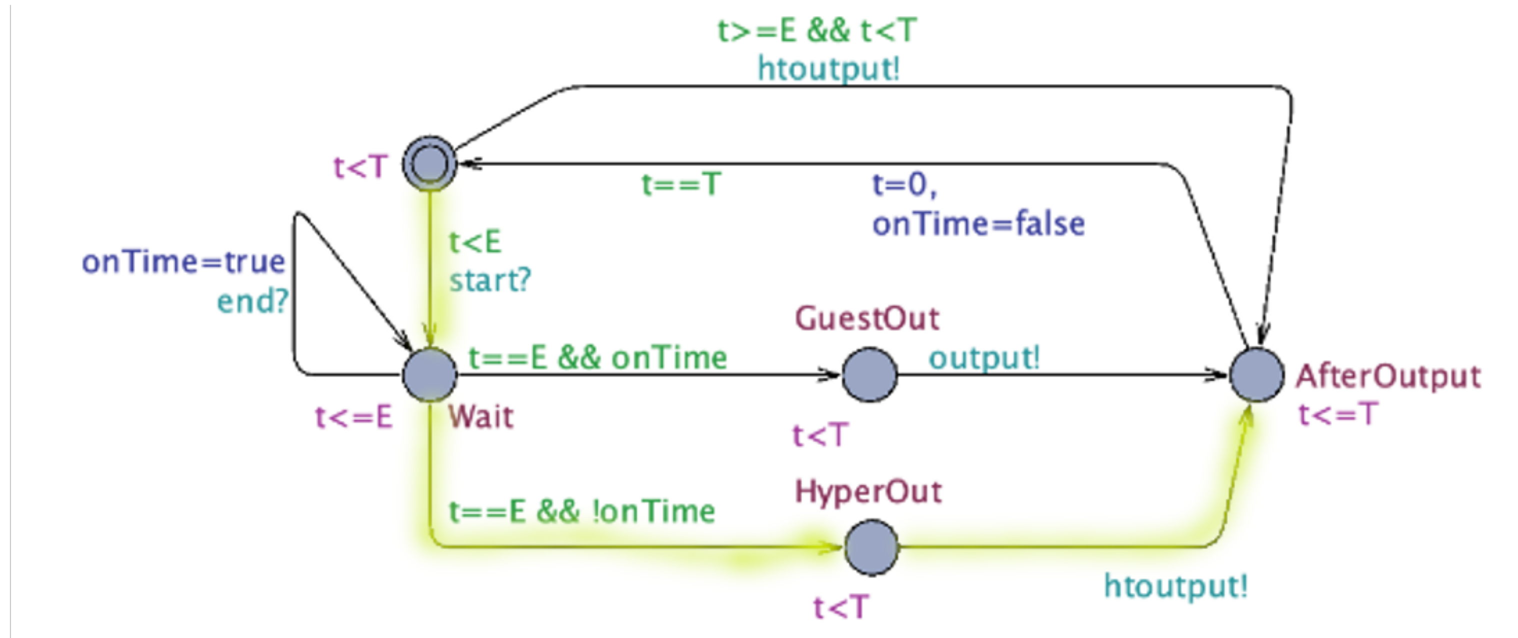
- ***Output* signal** is broadcast if the guest's job execution is **valid**
- ***Htoutput* signal** is broadcast if the guest's job execution is **invalid**
- Case 2:
  - The start signal is received before the clock reaches E
  - Hyper task transition to the wait location where it waits for an end signal

Hyper Task Automaton



- ***Output* signal** is broadcast if the guest's job execution is **valid**
- ***Htoutput* signal** is broadcast if the guest's job execution is **invalid**
- Case 2a:
  - End signal is received before the clock reaches E
  - **valid guest task execution**

Hyper Task Automaton



- ***Output* signal** is broadcast if the guest's job execution is **valid**
- ***Htoutput* signal** is broadcast if the guest's job execution is **invalid**
- Case 2b:
  - End signal is not received before the clock reaches E
  - **invalid guest task execution**

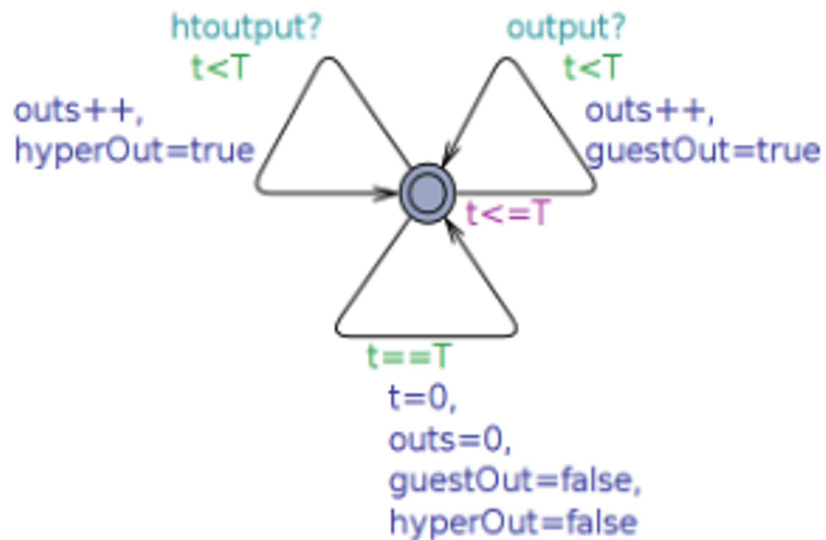
Verification of Timing Properties

Properties
P1: Each mixed-trust task must produce an output every period
P2: There is only one output per period
P3: If the guest job execution is valid, the output is from the GT
P4: If the guest job execution is invalid, the output is from the HT

- UPPAAL’s specification language is restricted:
  - No alternating temporal logical operators
  - **Solution:**
    - Construct two observer automata
    - Easier to write temporal specifications using observers

Output Observer Automaton

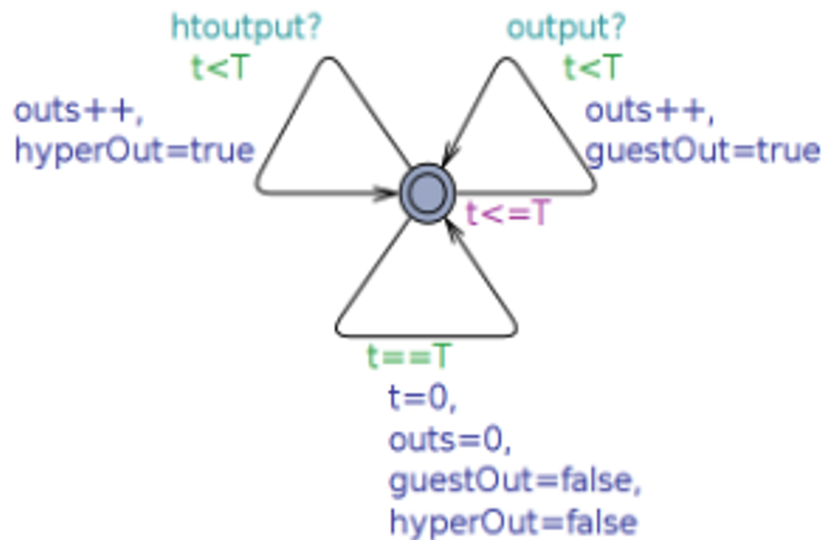
Properties
P1: Each mixed-trust task must produce an output every period
P2: There is only one output per period



- Behaviors related to P1 and P2 (exactly one output per period) are tracked by the output observer automaton
- We want to prove safety properties, i.e. nothing bad will happen
- Temporal formulas will take the form AG ¬F:
  - ¬F holds in the **current state** and in **all paths of execution**

Output Observer Automaton

Properties
P1: Each mixed-trust task must produce an output every period
P2: There is only one output per period



- Temporal formulas:

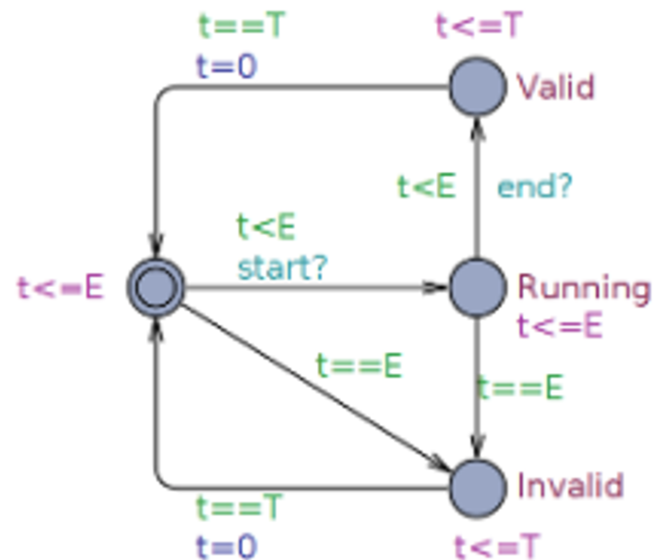
$$AG \neg (outputObserver.t == T \wedge outputObserver.outs == 0)$$

$$AG \neg (outputObserver.outs > 1 \wedge outputObserver.t < T)$$

- If these formulas are valid then it is guaranteed that the protocol never finishes a period with zero outputs or with more than one output

Job Observer Automaton

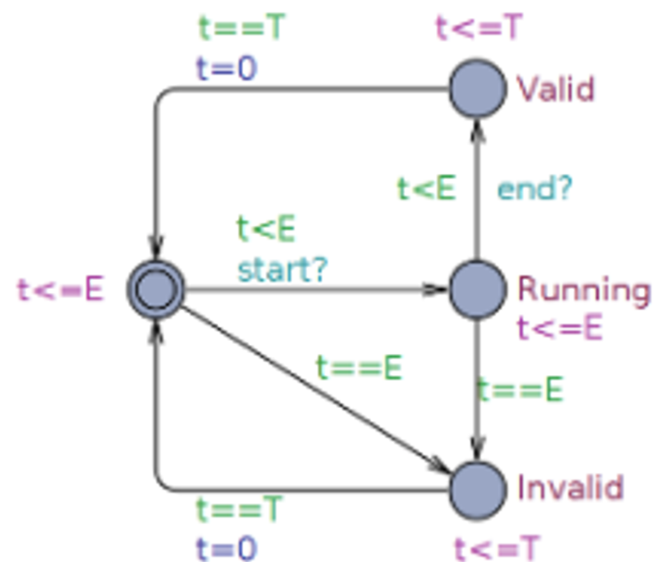
Properties
P3: If the guest job execution is valid, the output is from the GT
P4: If the guest job execution is invalid, the output is from the HT



- Behaviors related to P3 and P4 (output of the mixed-trust task) are tracked by the job observer automaton
- The signals *start* and *end* from the guest task are tracked to determine if the guest task job execution is valid:
  - **Valid**: if the *start* and *end* signals are received (in that order) within one period before the clock reaches *E*
  - **Invalid**: otherwise

Job Observer Automaton

Properties
P3: If the guest job execution is valid, the output is from the GT
P4: If the guest job execution is invalid, the output is from the HT



- If the guest job is **valid** then the **output** must be the one from the **guest task**:

$$AG \neg (jobObserver.Valid \wedge hyperOut)$$

- If the guest job is **invalid** then the **output** must be the one from the **hyper task**:

$$AG \neg (jobObserver.Invalid \wedge guestOut)$$

Proving Temporal Properties

Properties	Time (s)
P1: Each mixed-trust task must produce an output every period AG ¬ (outputObserver.t == T /\ outputObserver.outs == 0)	57
P2: There is only one output per period AG ¬ (outputObserver.outs > 1 /\ outputObserver.t < T)	54
P3: If the guest job execution is valid, the output is from the GT AG ¬ (jobObserver.Valid /\ hyperOut)	52
P4: If the guest job execution is invalid, the output is from the HT AG ¬ (jobObserver.Invalid /\ guestOut)	24

Modeling Challenges

- Zeno executions:
  - The conditions and invariants for each edge must be chosen carefully
  - Avoid multiple events from occurring simultaneously or having infinite discrete events occurring in finite time

- Model simplicity
  - Models look simple but they are the result of multiple iterations to improve readability
  - Previous models were either:
    - Too complex
    - Did not satisfy the desired properties

Protocol Implementation Corrections

- When modeling this protocol in UPPAAL we found a critical flaw in a previous implementation
- Our modeling and verification properties are ***tightly connected*** with detecting a ***start*** and ***end*** of a guest job
- Previous implementation:
  - Different methodology to detect the completion of a job
  - It was only tracking the ***end*** of the job
  - **Issue:** when a job ends, we do not know if it started in the current period or in the previous period; it does not satisfy **C4**:

C4: A guest output must be the product of a computation that executes within a single period

Protocol Implementation Corrections

Bug: not able to detect invalid jobs where the guest task would **finish** in a **different period**

- The following invalid guest job would be *incorrectly* classified as *valid*:

Period (1,start)			Deadline (GT)		Period (1, end)	Period (2, start)			Deadline (GT)		Period (2, end)
	Invalid GJ										

C4: A guest output must be the product of a computation that executes within a single period

***Solution*: To satisfy property C4 we now track *start* and *end* of a job**

Conclusions

- Using the **mixed-trust framework** we can have safety guarantees for **complex systems**
- In systems where **not all parts are verified**, it is critical to **verify** the **interaction** between trusted and untrusted components
- **Modeling** of protocols **increases** our **assurance** in the correctness of the properties that we want to enforce

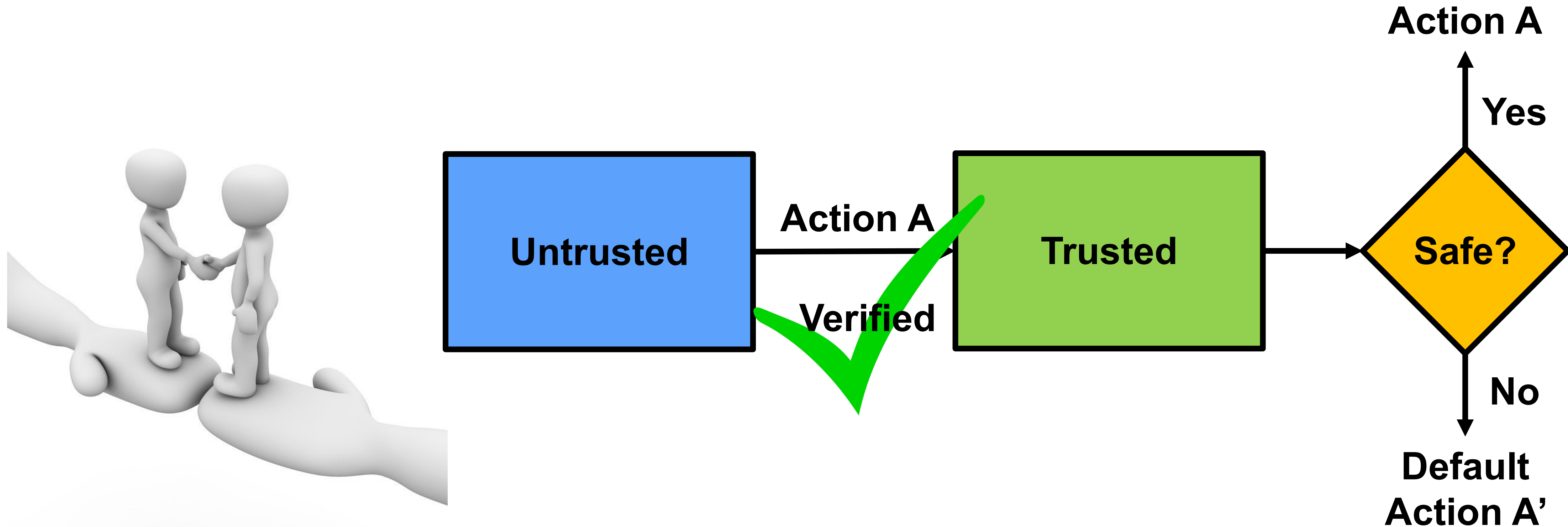
Future Work

- The model can help us improve the implementation

- How can we guarantee that the code follows the model?
    - Automatically generate C code that follows the UPPAAL model
        - Verify the translation

    - Verify that the current implementation follows the model
        - Prove the temporal properties at the code level

Verified Mixed-Trust Synchronization Protocol



Questions?