Cooperation between Automatic and Interactive Software Verifiers

Martin Spiessl
Joint work with Dirk Beyer and Sven Umbricht

LMU Munich, Germany

Terminology: Automatic vs. Interactive Verifiers

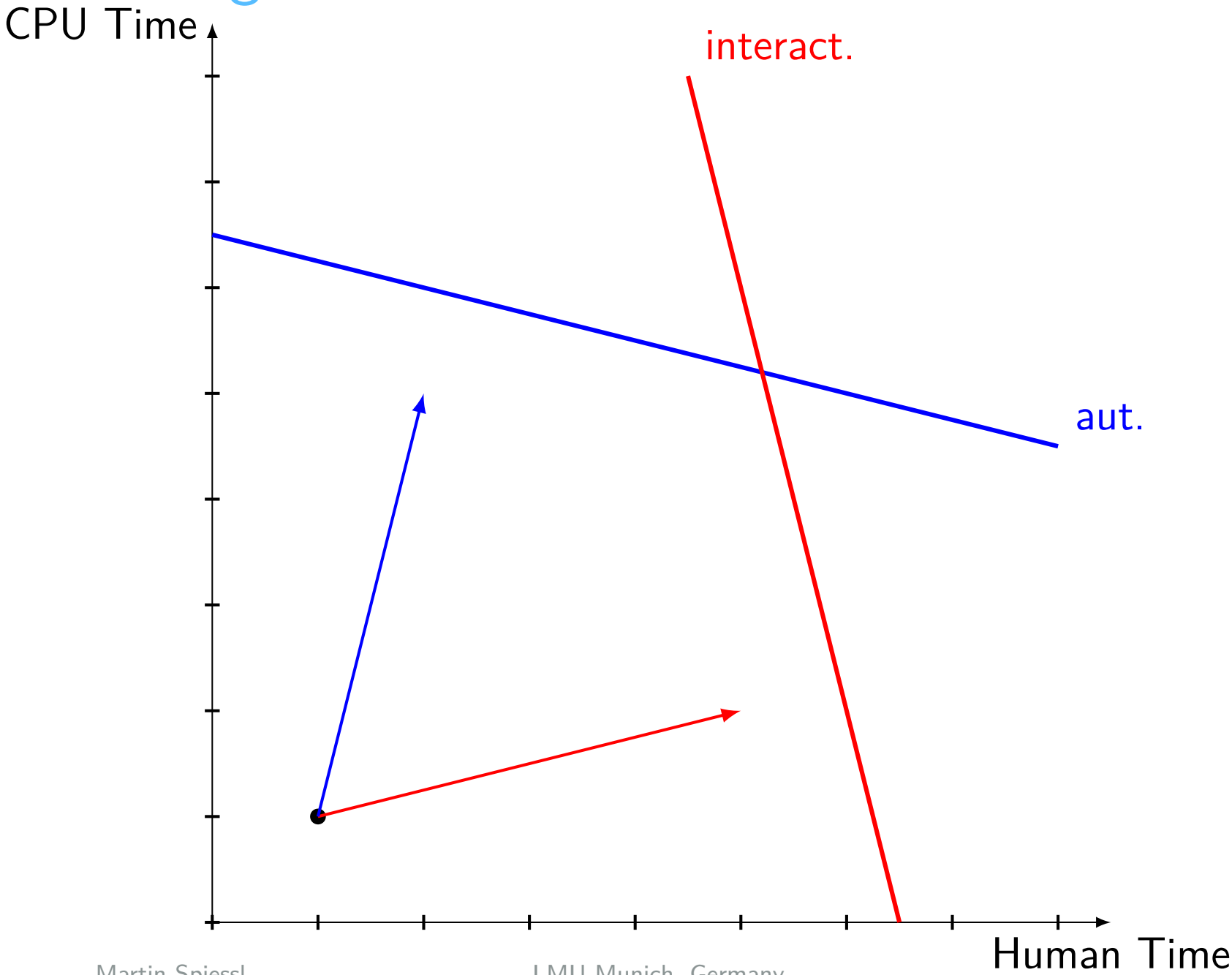
- We can roughly categorize software verifiers into two categories:
  - Automatic Verifiers
    - Try to solve verification task without user interaction
    - Capabilities improved over the last decades
    - But still challenging to scale to real-world examples
    - Examples: CBMC, CPACHECKER, GOBLINT, KORN, PESCO, SYMBIOTIC, UAUTOMIZER, VERIABS
  - Interactive Verifiers
    - User provides contracts, assertions etc. interactively
    - Powerful, but requires expert to provide the right clues
    - Lacking ways to automatically suggest clues
    - Examples: DAFNY, FRAMA-C, KEY, KIV, VERIFAST

Outline

In this talk we will investigate cooperation in the following three directions:

1. Interactive Verifier $\rightarrow$ Automatic Verifier
2. Automatic Verifier $\rightarrow$ Interactive Verifier
3. Automatic Verifier $\rightarrow$ Automatic Verifier

0. The Big Picture: Automatic vs. Interactive



0. Idea for Cooperation

- How to achieve cooperation between automatic and interactive verifiers?
- Idea: Try to use existing interfaces for information exchange

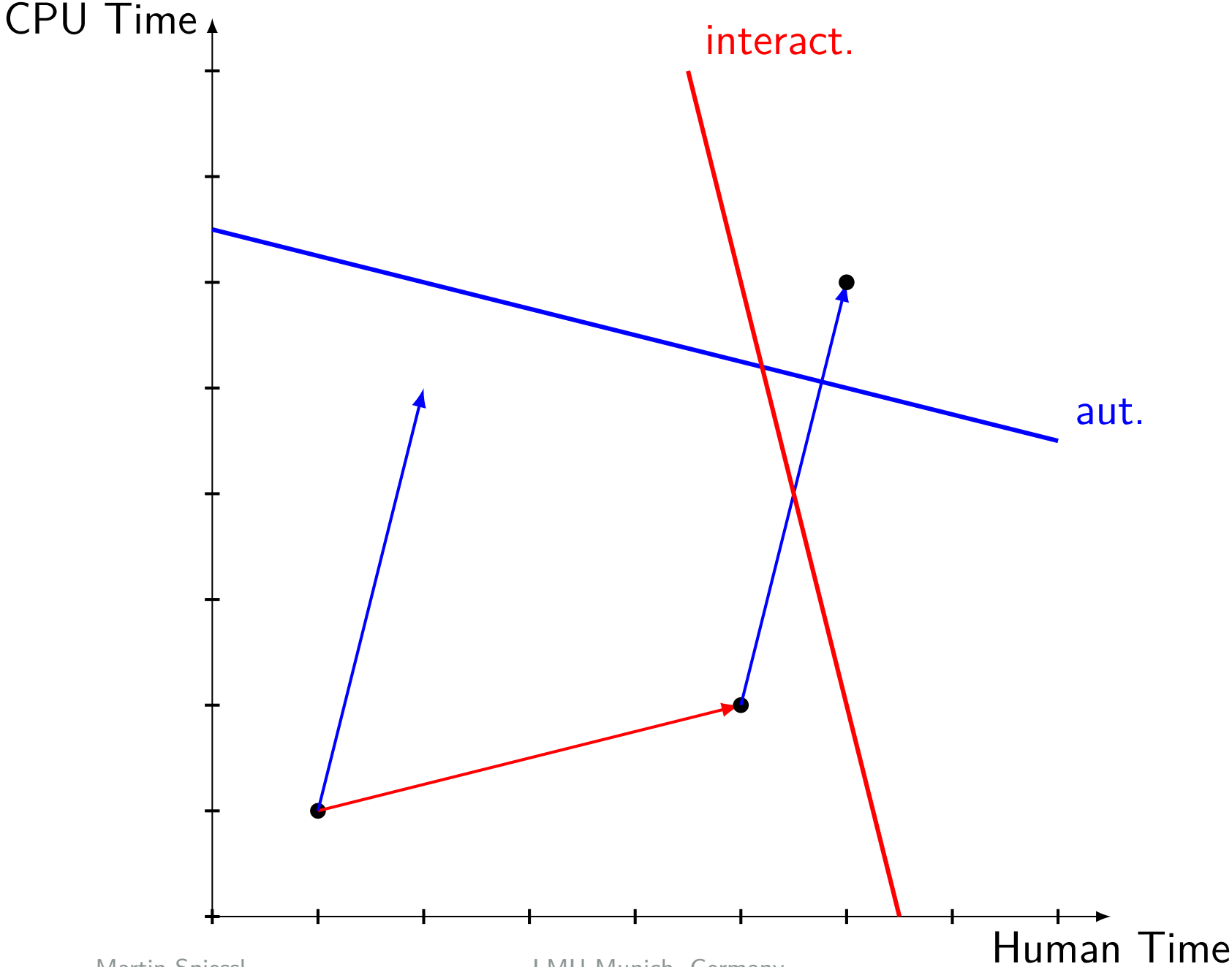
```
//@ensures \return==0;
int main() {
  unsigned int x = 0;
  unsigned int y = 0;
  //@loop invariant x==y;
  while (nondet_int()) {
    x++;
    //@assert x==y+1;
    y++;
  }
  assert(x==y);
  return 0;
}
```

ACSL-annotated program, as used by FRAMA-C

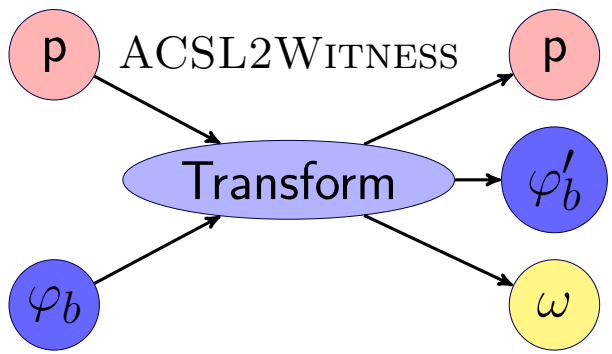
```
...
<node id="q1">
<data key="invariant">( y == x )</data>
<data key="invariant.scope">main</data>
</node>
<edge source="q0" target="q1">
<data key="enterLoopHead">true</data>
<data key="startline">6</data>
<data key="endline">6</data>
<data key="startoffset">157</data>
<data key="endoffset">165</data>
</edge>
...
```

GraphML-based witness automaton generated by automatic verifiers

1. Cooperation: Interactive $\rightarrow$ Automatic



1. Cooperation: Interactive → Automatic



1. Small Case Study on Annotated Inductive Invariants

We perform the following steps:

- Select a small set of tasks from SV-Benchmarks[1]
- Annotate sufficiently strong[2], inductive invariants via ACSL
- Convert annotations into witnesses using ACSL2Witness
- Run Frama-C in the annotated programs
- Run CPAchecker's k-induction validation on the witnesses

[1] 11 tasks, mostly from the loop-invariant folder

[2] strong enough that the tasks should be provable with this information

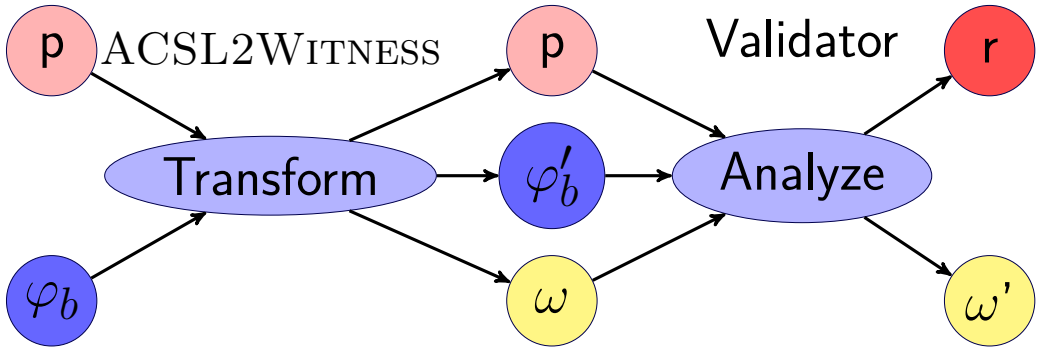
1. Small Case Study on Annotated Inductive Invariants

Verification Task	Frama-C	CPAchecker's *k*-induction
afnp2014.yml	✗	✓
bin-suffix-5.yml	✗	✓
const.yml	✓	✓
eq1.yml	✓	✓
eq2.yml	✓	✓
even.yml	✗	✓
linear-inequality-inv-a.yml	✗	✓
mod4.yml	✗	✓
nested_1.yml	✓	✓
nested_2.yml	✓	✗
odd.yml	✗	✓

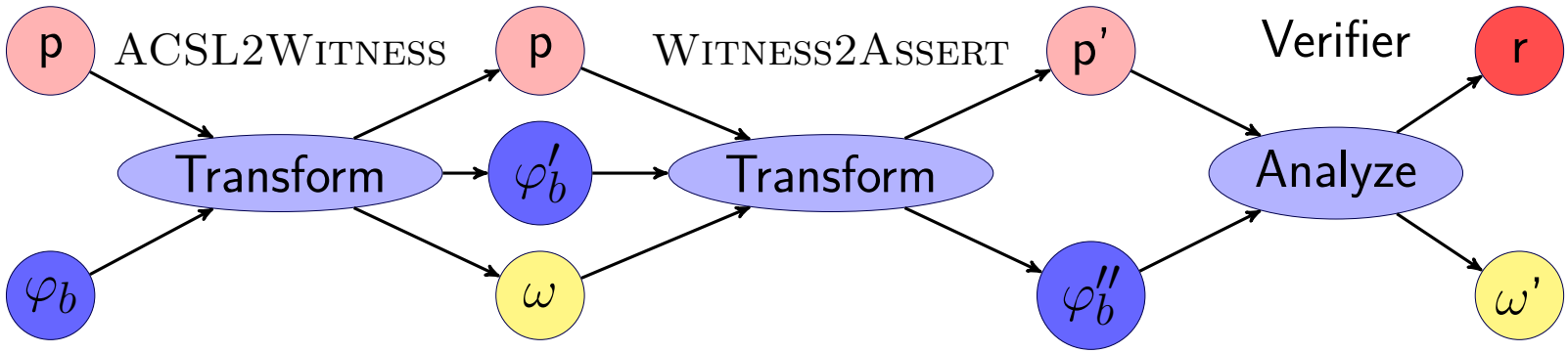
- appears to work well in practice, CPAchecker even solves sometasks that Frama-C cannot

1. Component Framework: Constructing Interactive Verifiers

- Turn a witness validator into an interactive verifier:

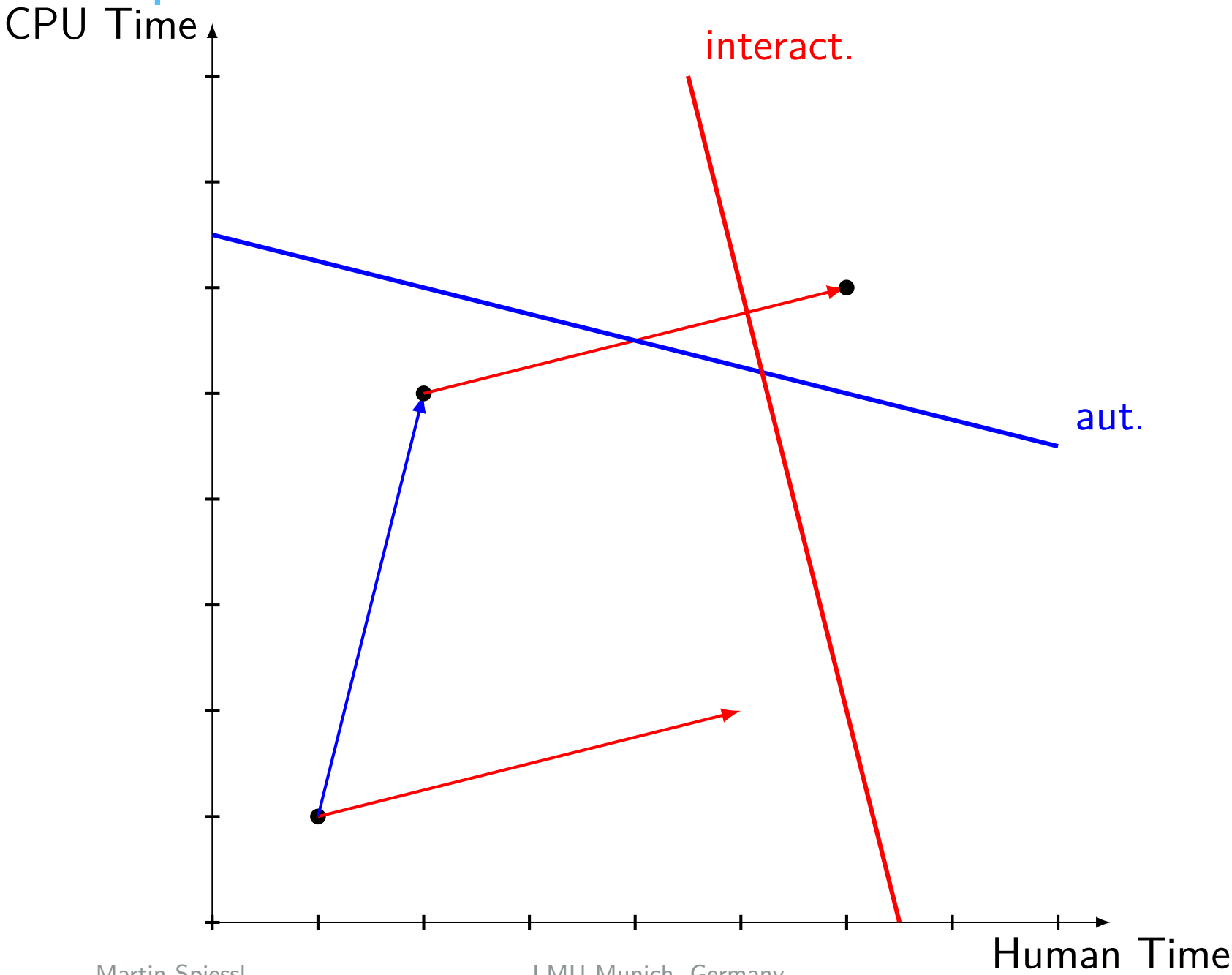


- Turn an automatic verifier into an interactive verifier::

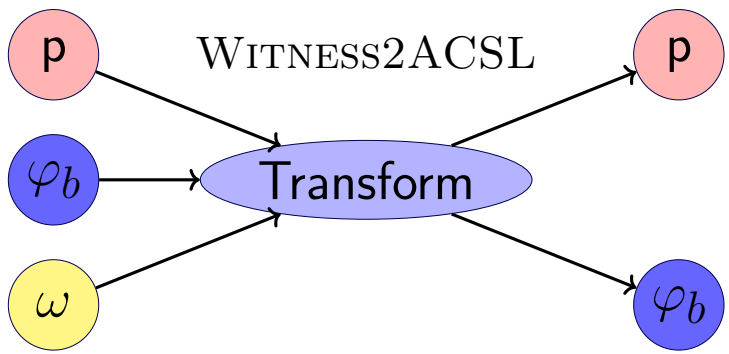


- Annotating in ACSL is more human-readable than witness automata
- Works for a wide range of automatic verifiers/validators

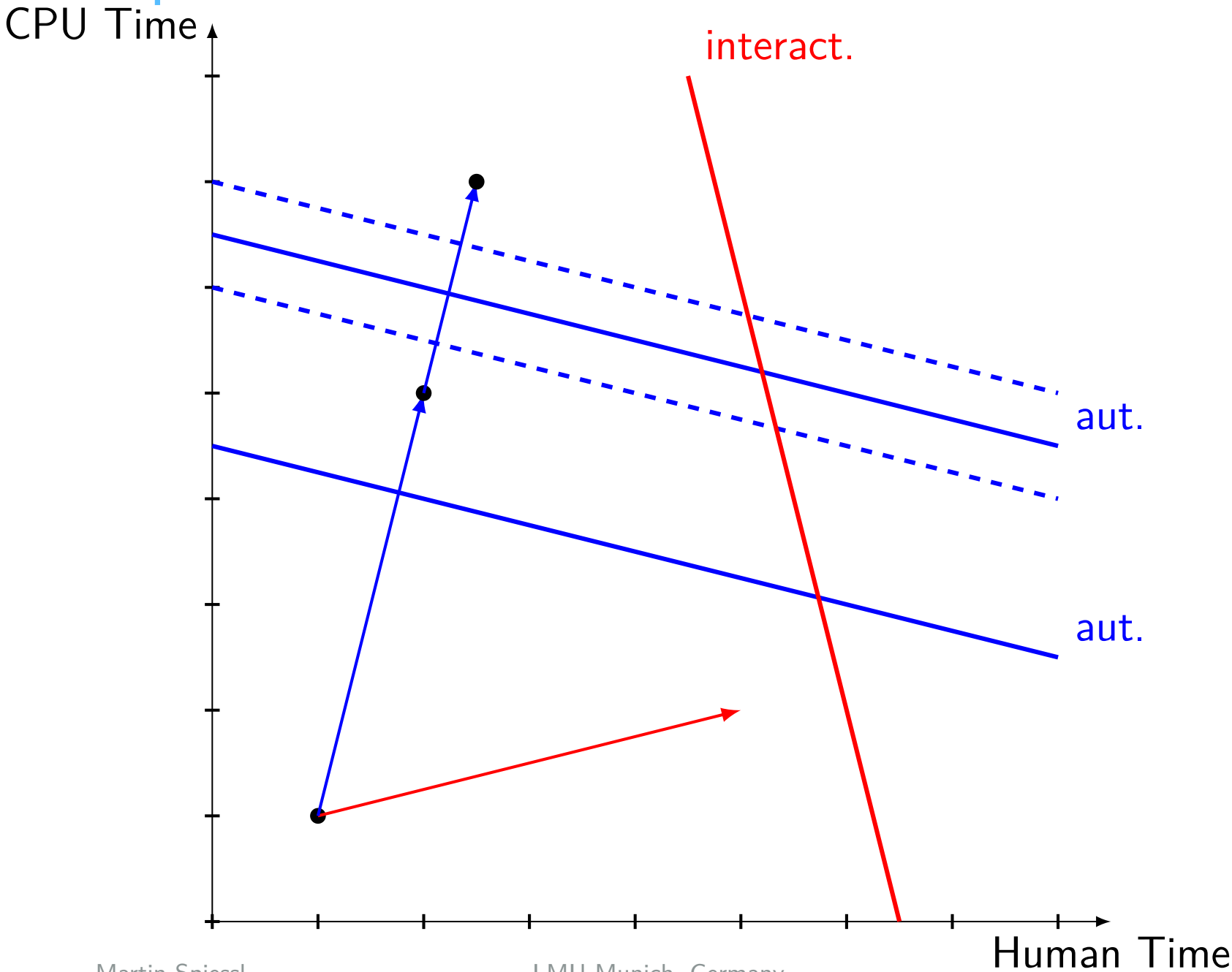
2. Cooperation: Automatic → Interactive



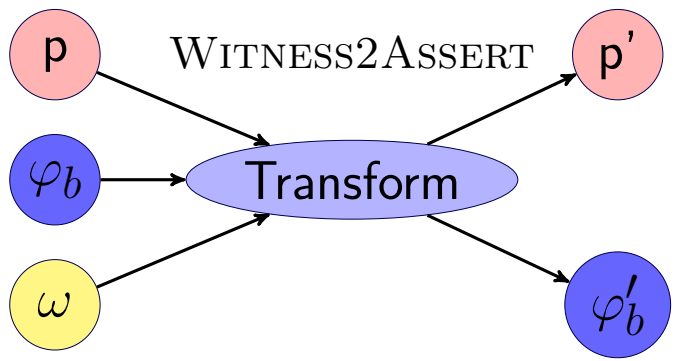
2. Cooperation: Automatic → Interactive



3. Cooperation: Automatic → Automatic

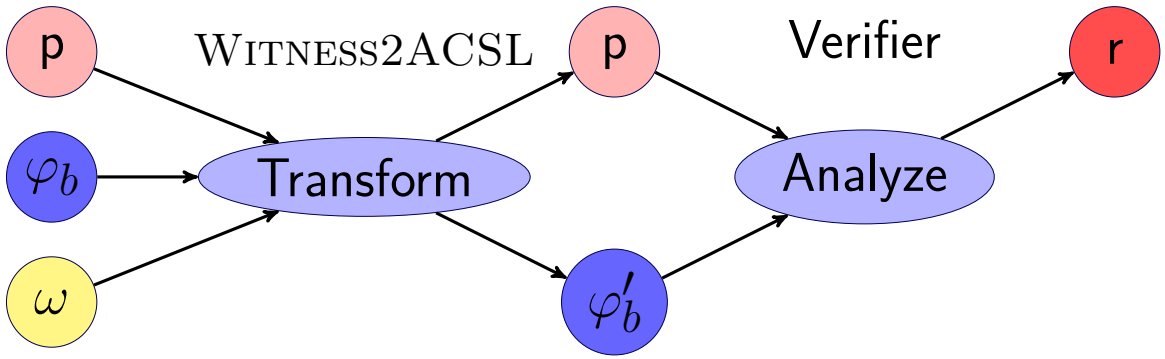


3. Cooperation: Automatic → Automatic

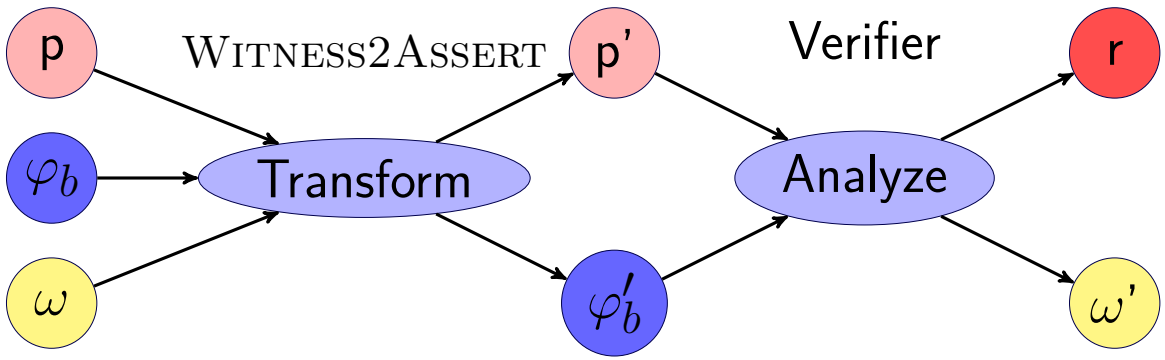


Component Framework: Constructing Validators

- 2. Turn an interactive verifier(FRAMA-C) into a validator:



- 3. Turn an automatic verifier into a validator:



Experiment on Cooperation using Witnesses

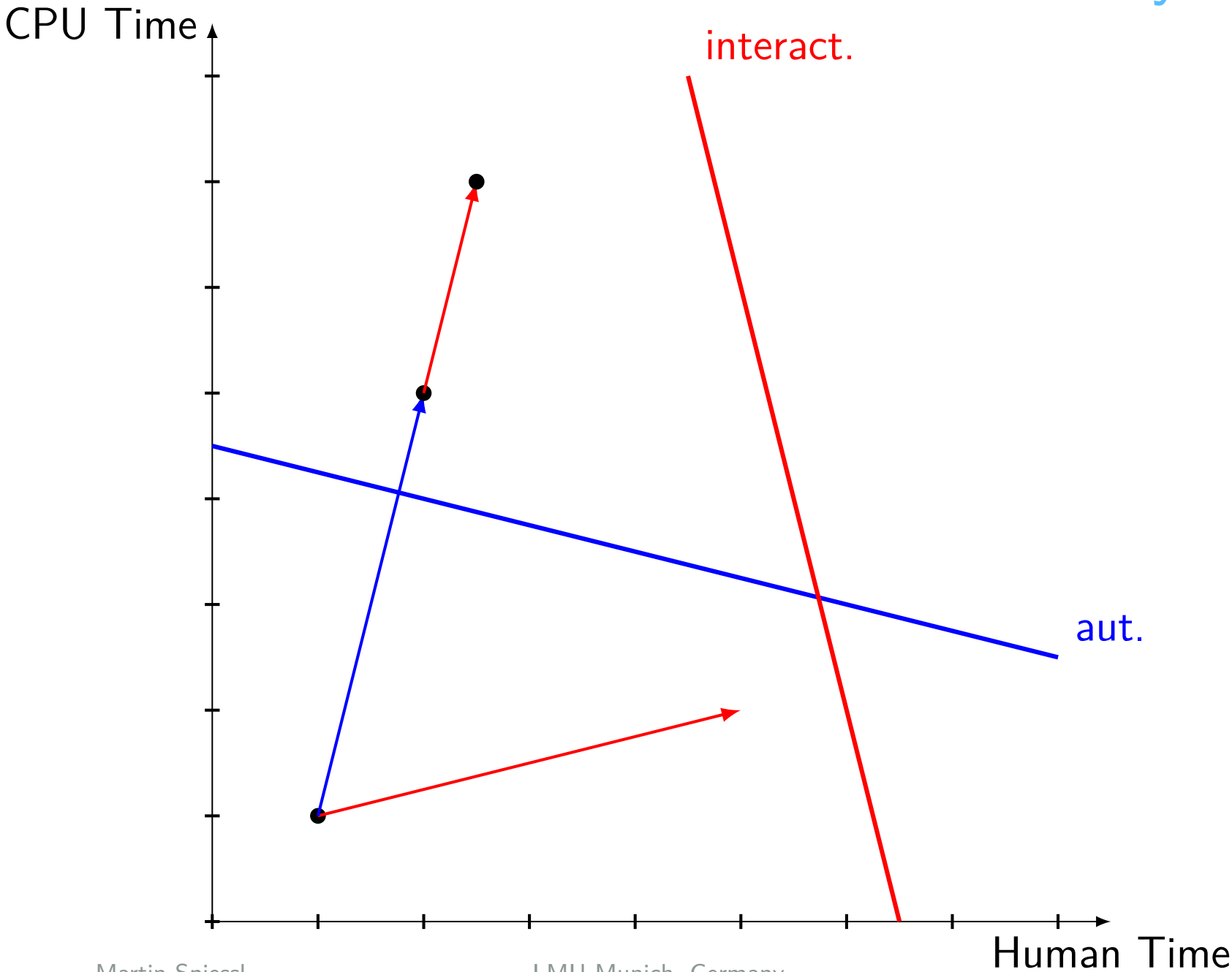
We perform the following steps:

- Take witnesses from the Reachability category of SV-COMP 2021
- Transform their invariants either into assertions or ACSL annotations
- Execute verifiers:
  - SV-COMP participants on the programs with assertions
  - Frama-C on the programs with ACSL annotations
- Analyze which verifiers were able to solve verification tasks only with the additional information, but not on the original task

Experimental Results on Cooperation (Consumers)

Consuming verifier	Benchmark Tasks (330 total)		Projection on Original Programs (197 total)	
	Baseline	Improved via Cooperation	Baseline	Improved via Cooperation
2LS	115	134	70	99
UAUTOMIZER	179	90	98	72
CBMC	240	30	147	16
CPACHECKER	253	40	153	22
DARTAGNAN	170	114	105	73
ESBMC	201	91	124	58
GAZER-THETA	221	72	136	39
GOBLINT	29	90	14	70
UKOJAK	68	72	38	42
KORN	116	160	70	104
PESCO	171	97	93	77
PINAKA	231	52	141	26
SYMBIOTIC	141	124	70	91
UTAIPAN	142	161	87	96
VERIABS	272	55	167	28
FRAMA-C	3	11	3	6

2. We use Frama-C in an "Automatic" Way!



Experimental Results on Cooperation (Producers)

Producing verifier	Useful witnesses	Cases of cooperation
UAutomizer	8	15
Cbmc	76	88
CPAchecker	144	808
Goblint	1	1
Korn	10	30
PeSCo	91	451
Sum	330	1 393

- Our transformations use CPAchecker
  $\Rightarrow$ Technology bias: most coop via CPAchecker
- But also other tools contribute useful invariants!
- Potential basis for a dataset of "objectively useful" invariants

Contributions

1. Building blocks for new compositions and ways of cooperation
   - Witness2ACSL
     - Let interactive verifiers use results from automatic verifiers
     - Use interactive verifiers as validators
     - Make information in witnesses more accessible to the user
   - ACSL2Witness
     - Make automatic verifiers "more interactive"
   - Witness2Assert
     - Enables cooperation between automatic verifiers
2. Experimental results proving practicality of our approach
3. Byproduct: benchmark set on helpful invariants/ annotated programs