# Mechanical Verification of the Correctness of an Embedded Verifier

Piergiorgio Bertoli (IRST)

bertoli@irst.itc.it

Joint work with

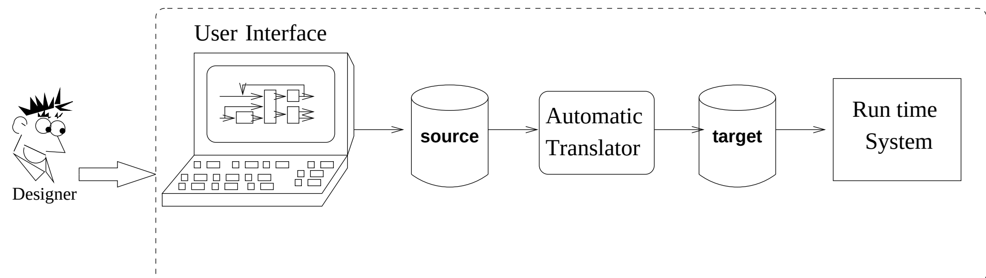
Paolo Traverso (IRST) and Alessandro Cimatti (IRST)

## Structure of the talk:

- The problem
- The on-line/off-line logging and checking approach
- A non-automated verification
- Automatizing the verification: why and how
- Partial results in automatization
- Workplan

# The VFRAME industrial problem

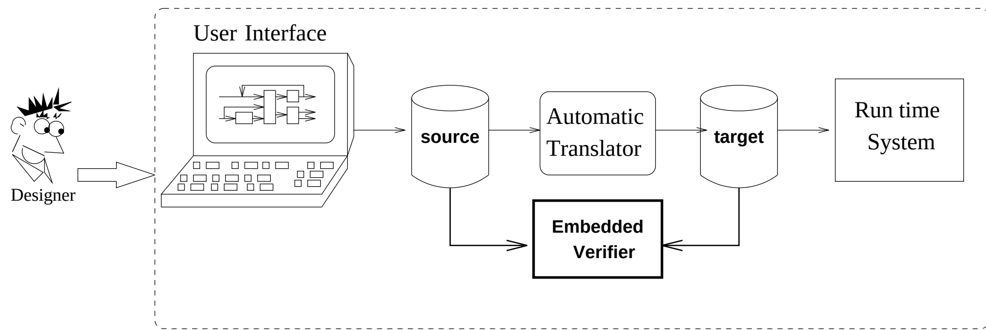
- Task: verify the correct behaviour of a COTS-based translator used to design safety-critical applications




- The runtime system is designed to guarantee correct behaviour w.r.t. transient errors
- The translation is safety-critical
- COTS may introduce runtime errors in the translation

# The approach

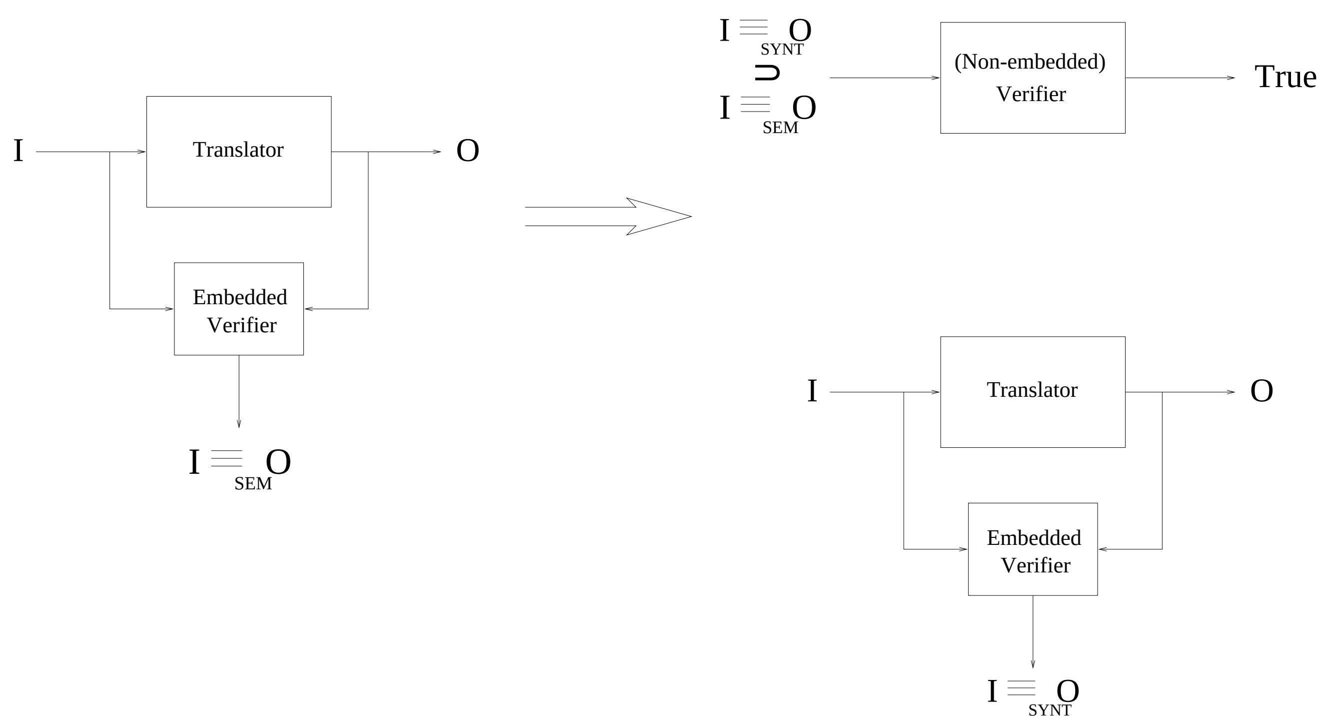
- Introduce an **Embedded Verifier:**



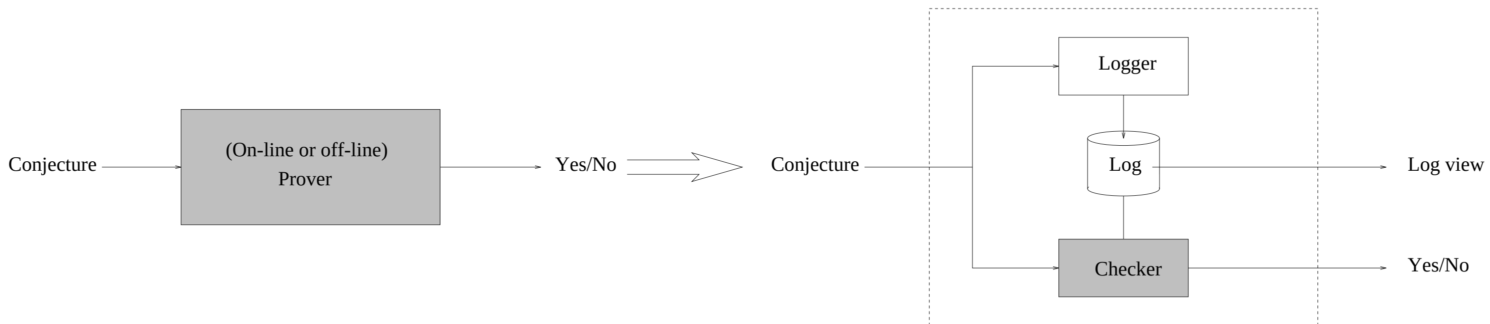

- **Each** execution is verified; but:
  - **Req. 1:** Verification must be fast and automatic
  - **Req. 2:** The embedded verifier must be certified

# Satisfying requirement 1: off/on-line decomposition

- Idea: simplify verification conditions **once for all**



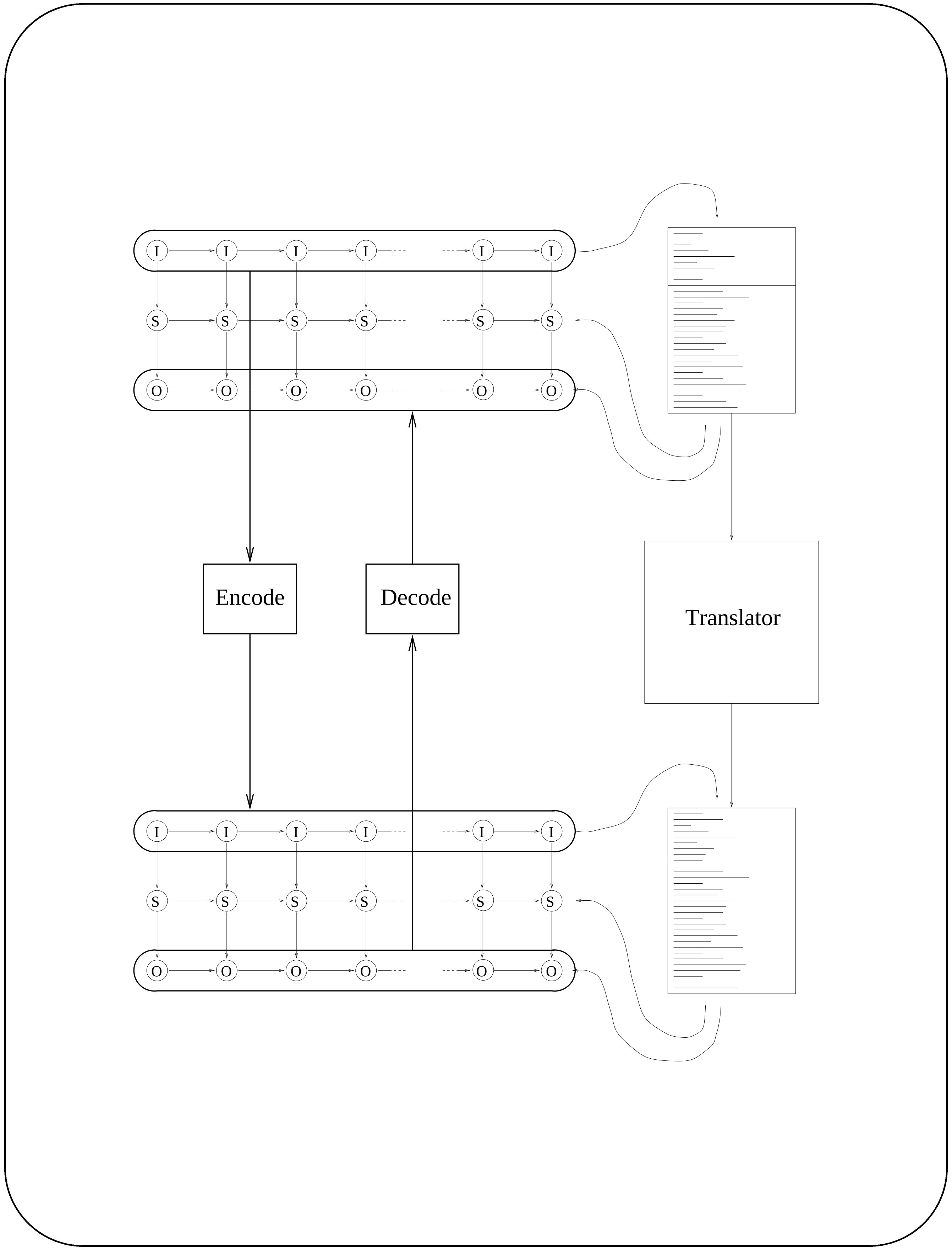
# Satisfying requirement 2: logging-and-checking




- Logger: complex but not critical
- Checker: simple and critical

## The notion of correct translation

- Correct translation amounts to equivalent observable (I/O) behaviour
- I/O behaviour described by the I/O semantics ($[P]_{I/O}$), i.e. by maps between sequences of inputs and sequences of outputs
- Programs may represent data using different format. We represent this by an encoding $E$ (and its inverse decoding $D$)
- Equivalent I/O behaviour of $P_1$, $P_2$:
  $[P_1]_{I/O}(I) = D([P_2)_{I/O}(E(I)))$


I
S
O
Encode
Decode
Translator

## Correct translation for GEM and RTM

- GEM and RTM are reactive languages: they execute an endless input-compute-output loop
- input means updating the values of tagged *input* variables
- output means extracting the values of tagged *output* variables
- GEM and RTM are stack-free, jump-free **really???** languages
- GEM supports long integers and booleans
- RTM supports short integers only (in order to implement table-based mathematics)

Data conversion:

- GEM integers are translated into tuples of RTM integers via the Chinese Remainder Theorem
- GEM booleans are converted into single RTM integers

Operation conversion

- GEM operations on integers translate to sequences of RTM instructions
- GEM operations on booleans translate to single RTM instructions

Resuming:

- GEM and RTM states consist of the values of their variables
- Decoding and encoding can be represented as static correspondence matrices
- GEM and RTM inputs are obtained by projecting the "input" tagged cells of their memories at the start of the endless loop
- GEM and RTM inputs are obtained by projecting the "input" tagged cells of their memories at the start of the endless loop

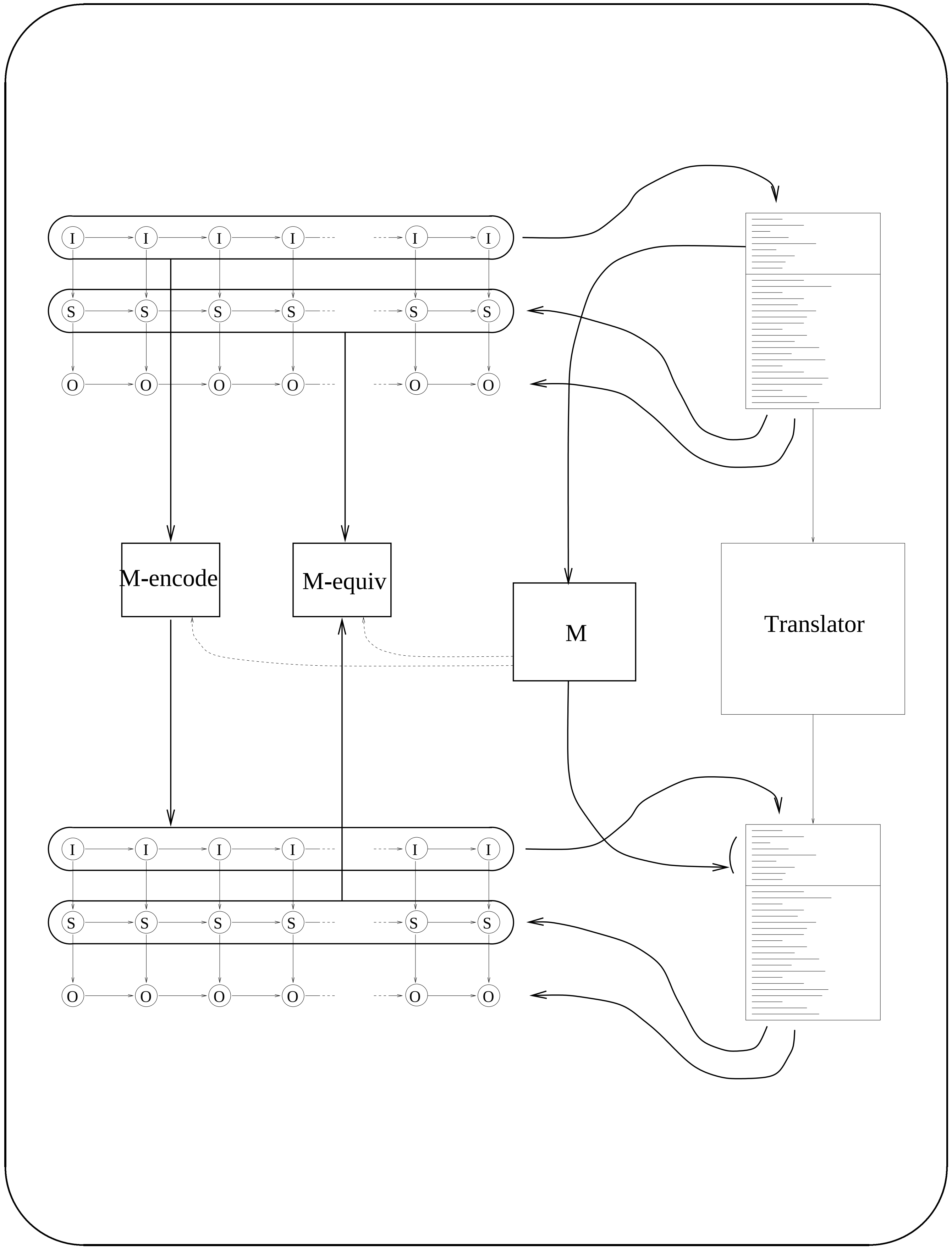
## The off-line reduction: simplified conditions

Definition of simplified conditions bases on:

- concrete mapping $M$ between GEM and (ev. tuples of) RTM variables
- translation scheme $\tau$ between GEM instructions and (ev. tuples of) RTM instructions

Derived definitions:

- $M$-induced encoding $E_M$ and decoding $D_M$
- $M$-induced state correspondence between
- $M$-induced inputs correspondence between
- $M$-induced outputs correspondence
- $M$-induced programs correspondence



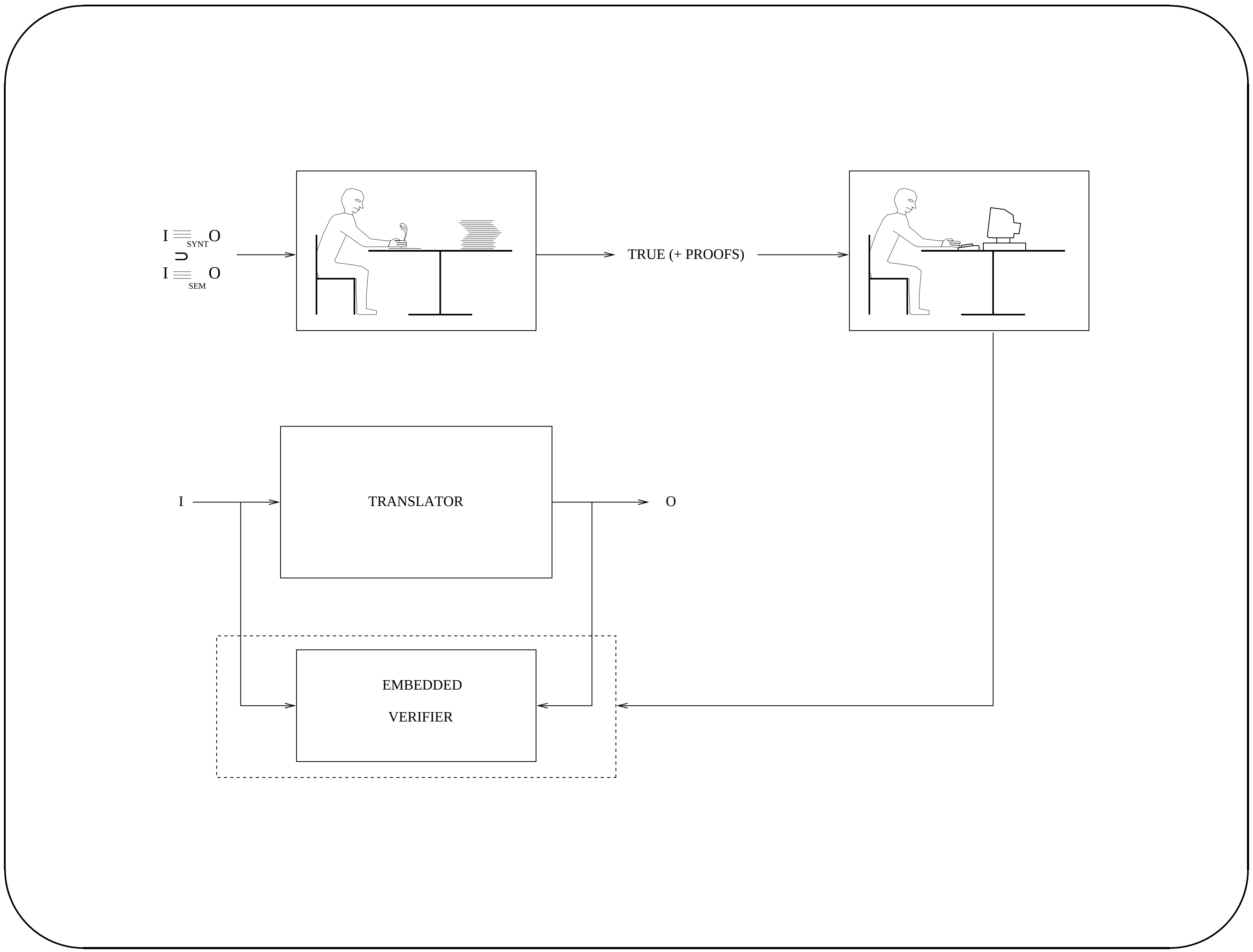

Mechanical verification of the Correctness of an Embedded Verifier

## The specific on/off-line decomposition

- Tasks of the offline proof phase:
  - Prove that, given any variable mapping $M$ and instruction translation scheme $\tau$, under given hypothesis $Hyps_M(M)$ and $Hyps_\tau(\tau)$, M-correspondent programs are I/O equivalent
- Tasks of the Embedded verifier (online proofs):
  - Logger: find mapping $M$ and instruction fibering (instance of $\tau$)
  - Checker: check that $M$ and $\tau$ obey $hyps(M)$, $hyps(\tau)$

## The non-automated solution to the problem

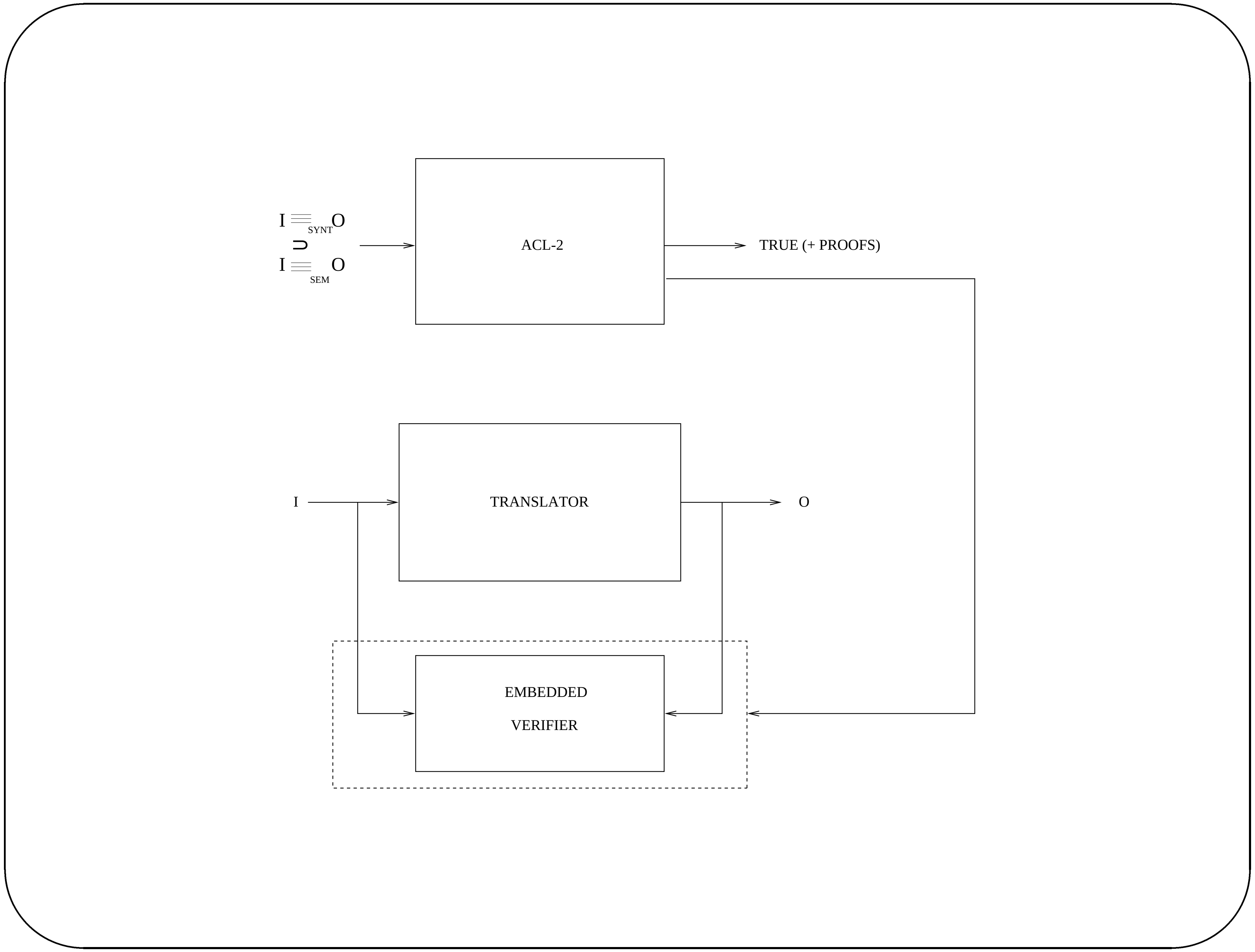
- Off-line simplification of verification conditions carried out by hand (proofs provided as a deliverable)
- Embedded verifier designed as a custom C program (claimed to be trustable due to "easy" mapping between checker and $\equiv_{SYNT}$)




`Mechanical verification of the Correctness of an Embedded Verifier`

## Our task: Mechanizing off-line proofs via ACL2

- **Concrete objectives:**
  - Increase trust in proofs
  - Automatically synthesize embedded verifier ("executing" ACL2's logic)
- **Abstract objective:**
  - Study feasibility of mechanizing off/on-line approach




Mechanical verification of the Correctness of an Embedded Verifier

## Off-line reduction structure

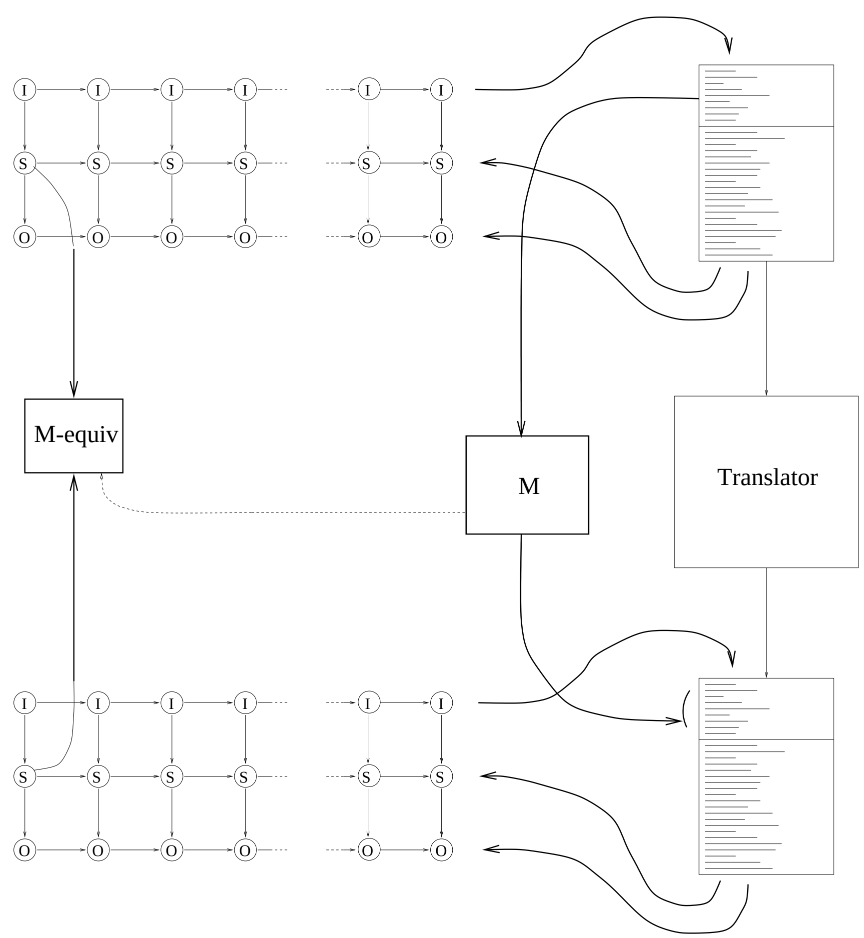
1. Prove that M-correspondence between states implies I/O equivalence:

$$[P^g](I) \equiv_M [P^r](E_M(I)) \Longrightarrow [P^g]_{I/O}(I) = [P^r]_{I/O}(E_M(I))$$

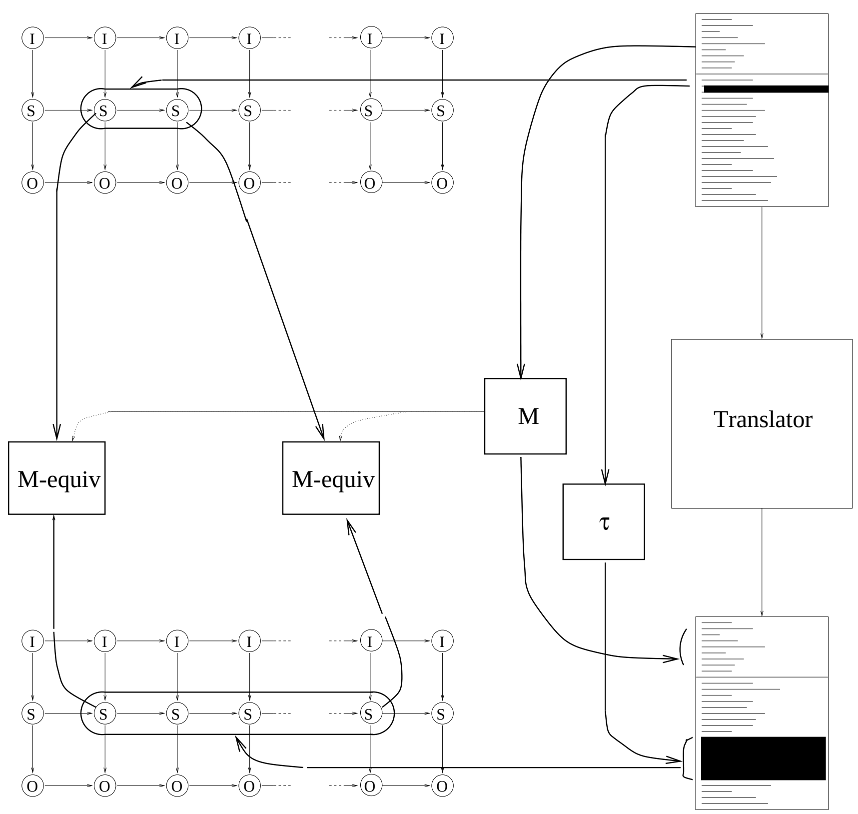
2. Prove that M-correspondent programs produce sequences of M-correspondent states

# Off-line reduction structure

Base step: Prove that M-correspondent programs feature M-correspondent initial states



Induction: prove preservation of states M-correspondence upon execution of $\tau$-corresponding sequences of instructions:



## Project status overview

Phase1: Proved

Phase2: To Do

# Phase 1 detail

- Typed-and-tagged memories formalized as lists of triples (similar to Moore's small machine)

  ```
  code here...
  ```

- Mapping defined by description of properties which identify the class of acceptable conversions.

  ```
  some example code here
  ```

  Note: this is quite different from providing a constructively defined witness for the mapping. It allows to capture all of the possible mappings generated by the black-box compiler.

## Main property proved: initial state correspondence

"If

- $M$ is a well-formed mapping,
- $g$ is a GEM memory,
- $r$ is a RTM memory,
- $g$ is an M-correspondent state w.r.t. $r$,

then - the M-induced decoding of the input (output) of $r$ is equivalent to the input (output) of $g$."

```
(defthm decoded-mem-is-equal-mem-to-gem-mem-proj
 (implies
  (and
   (true-listp m)
   (is-typed-mem-p gem-typed-mem)
   (m-good-wrt-attributes m rtm-typed-mem)
   (m-correspondent-values-p m gem-typed-mem rtm-typed-mem)
   (no-duplicates-gemvars-p m)
   (in-range-m-gemvars-p m gem-typed-mem)
   (correct-arity-wrt-mem-p m gem-typed-mem)
   (m-contains-gemvars m gem-typed-mem))
  (equal-mems
   (decode m (projectio rtm-typed-mem attr))
   (projectio gem-typed-mem attr))))
```

## Structure of phase 2: based on symbolic execution (Piton experience)

What we can take:

- Von Neumann machine ACL2 formalization and basic properties
- Idea: *"break down transformations using annotations"*
- Idea: *"use symbolic stepper routine and clock correspondences"*
- Idea: *"use partial inverting mappings"*

## Current status and workplan

- Current work status:
  - Base case proved lemma proved
- Workplans:
  - Model translation schemes for each GEM instructions and prove them correct
  - Use (adapted) Piton machinery to prove correctness of composition