Provably Deductive Assurance Cases

Leveraging Lean for Safety Modelling

Logan Murphy, Marsha Chechik, Torin Viger, Alessio Di Sandro, Ramy Shahin

University of Toronto

January 2021

Contents

Introduction to Assurance Cases

Problem of evaluating software safety:

- How can we know that the software we deploy is safe and won't exhibit undesired behaviour?
- Different approaches: testing, code review, benchmarks, model checking, automated theorem provers...
- Particularly important for safety-critical domains.

What sort of evidence should a safety engineer or regulator look for?

How does one manage that evidence?

Introduction to Assurance Cases

Some industries have embraced the use of goal-based *safety cases*:

- A clear, comprehensive and defensible argument that a system is acceptably safe to operate in a particular context[1].
- Author determines what evidence is appropriate

We use the term *assurance case* (AC) to generalize beyond safety requirements

- Security
- Privacy

[1] Tim Kelly and Rob Weaver (2004). *The Goal Structuring Notation–A Safety Argument Notation*.

Introduction to Assurance Cases

An AC consists of *claims* about the system. A claim can be supported by documenting *context*, *evidence* and *assumptions*.

Claim 1: Control system is acceptably safe.
Context 1: Definition of acceptably safe.

Claim 1.1: All identified hazards have been eliminated or sufficiently mitigated.
Context 1.1-a: Tolerability targets for hazards (reference Z).
Context 1.1-b: Hazards identified from functional hazard analysis (reference Y).

Strategy 1.1: Argument over all identified hazards (H1, H2, H3)

Claim 1.1.1: H1 has been eliminated.
Evidence 1.1.1: Formal verification

Claim 1.1.2: Probability of H2 occurring < $1x10^{-6}$ per annum.
Justification 1.1.2: $1x10^{-6}$ per annum limit for catastrophic hazards.
Evidence 1.1.2.: Fault Tree analysis.

Claim 1.1.3: Probability of H3 occurring < $1x10^{-3}$ per annum.
Justification 1.1.3: $1x10^{-3}$ per annum limit for major hazards.
Evidence 1.1.3: Fault tree analysis.

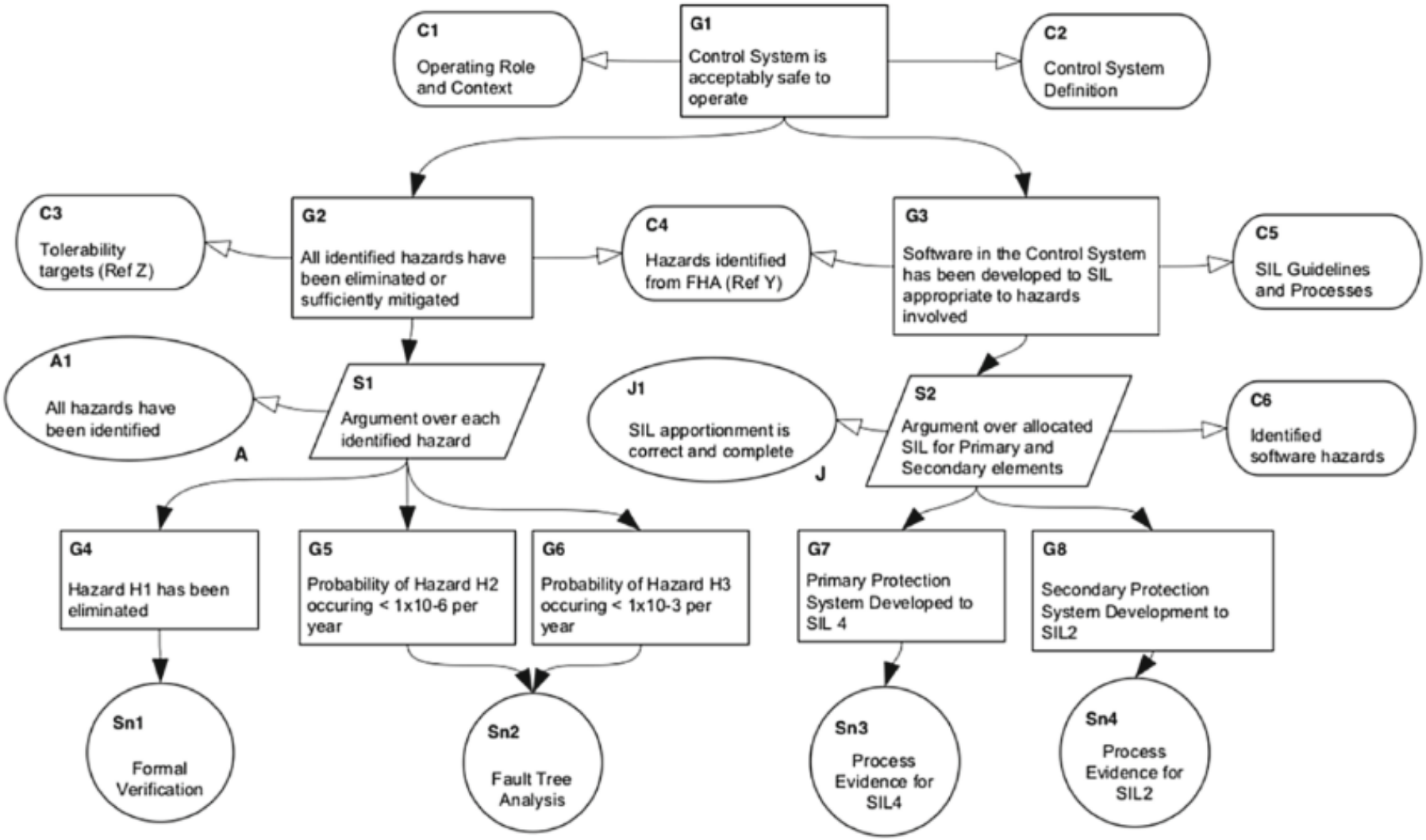
Claim 1.2: The software has been developed to the integrity level appropriate to the hazards involved.
Context 1.2-a: (same as Context 1.1-b)
Context 1.2-b: Integrity level (IL) process guidelines defined by reference X.

Claim 1.2.1: Primary protection system developed to IL 4.
Evidence 1.2.1: Process evidence of IL 4

Claim 1.2.2: Secondary protection system developed to IL 2.
Evidence 1.2.2: Process evidence of IL 2.

Assurance Case Notation

E.g. Goal Structuring Notation (GSN)



Assurance Case vs. Proof

An AC is **not** a formal proof.

- Needs to be reviewed and understood by stakeholders with different backgrounds

An AC is not even an informal proof.

- Primarily uses *inductive reasoning*, e.g. "Evidence suggests that errors are sufficiently unlikely".

Full formalization and automation of AC creation and assessment is not realistic or even desireable.

Deductive Reasoning in ACs

- However, ACs still rely on deductive reasoning (usually implicitly).
- Fallacious reasoning exists in real-world ACs[2].
- Manual review is an unreliable mitigation strategy[3].

We would like to

- Identify fragments of ACs where deductive reasoning can be made explicit
- Use a formal (i.e. deductive) framework to validate those fragments.

[2] William S. Greenwell et al. *A taxonomy of fallacies in system safety arguments*. 2006.
[3] John Rushby. *The interpretation and evaluation of assurance cases*. 2015.

Lean for Assurance Cases

We hope to use Lean as:

- A framework to formally express and prove properties of AC claims
- A tool to assist in generating formal evidence used in ACs

Current work with Lean builds on two threads of previous work.

Previous Work: Model Management for Assurance Cases

1. *MMINT* : Eclipse-based model management framework[4].
2. *MMINT-A* : Extended for assurance case management in the automotive domain[5].
3. *MMINT-A 2.0* : Extended to support lifecycle maintenance of system and safety models[6]

More information: `https://github.com/adisandro/MMINT`

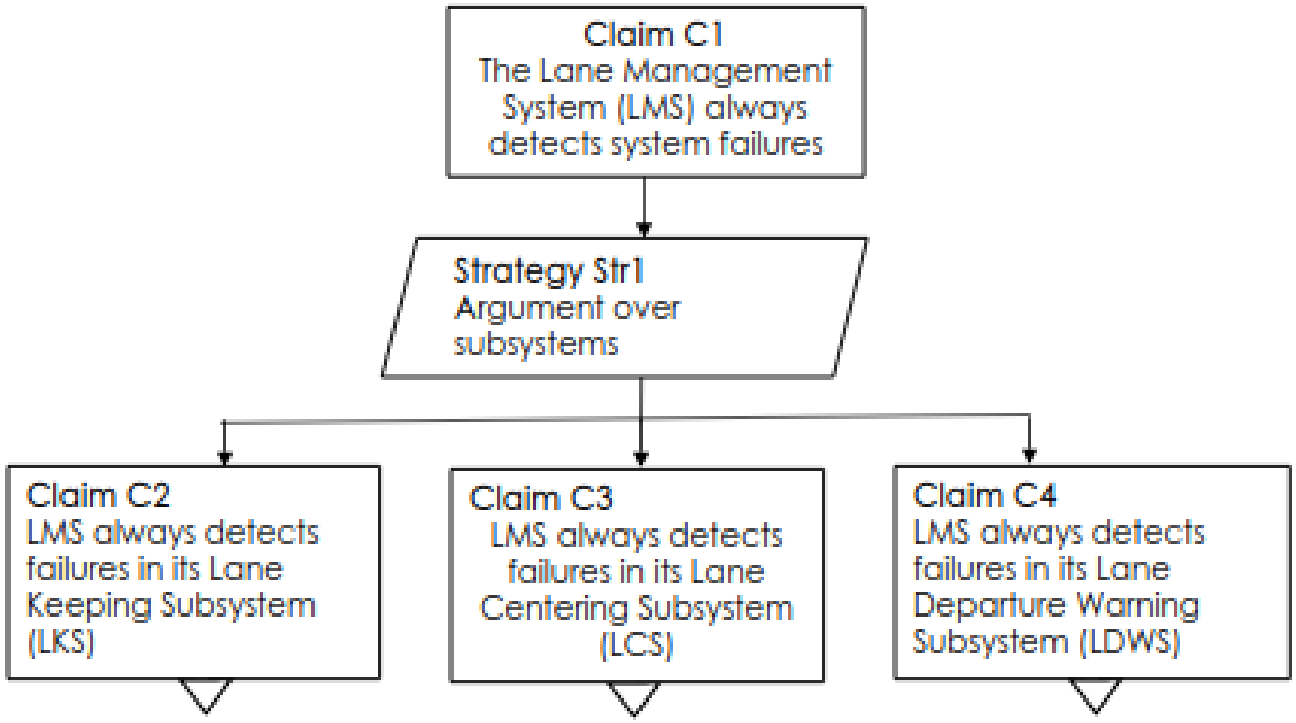
[4] A. Di Sandro et al. "MMINT: A Graphical Tool for Interactive Model Management". In: *P&D @ MoDELS*. 2015.

[5] Nick L. S. Fung et al. "MMINT-A: A Tool for Automated Change Impact Assessment on Assurance Cases". In: *Computer Safety, Reliability, and Security*. 2018.

[6] A. Di Sandro et al. *MMINT-A 2.0: Tool Support for the Lifecycle of Model-Based Safety Artifacts*. MODELS '20. 2020.

Previous Work: Formality in Assurance Cases

One possible framework for validating AC fragments focuses on *claim decomposition strategies*[7].



[7] Torin Viger et al. *Just Enough Formality in Assurance Argument Structures*. Computer Safety, Reliability, and Security (SAFECOMP), 2020.

Previous Work: Formality in Assurance Cases

Restrict ourselves to claims of a particular form:

Fix a type α. A *claim* consists of a `set` X of elements of type α, and a property (predicate) P on α.

To a claim $C(X, P)$ we assign the meaning

$$\forall x \in X, P(x)$$

Previous Work: Formality in Assurance Cases

Given an claim C, a *strategy* is a map from C to a list of claims $C_1, C_2, ...C_n$.

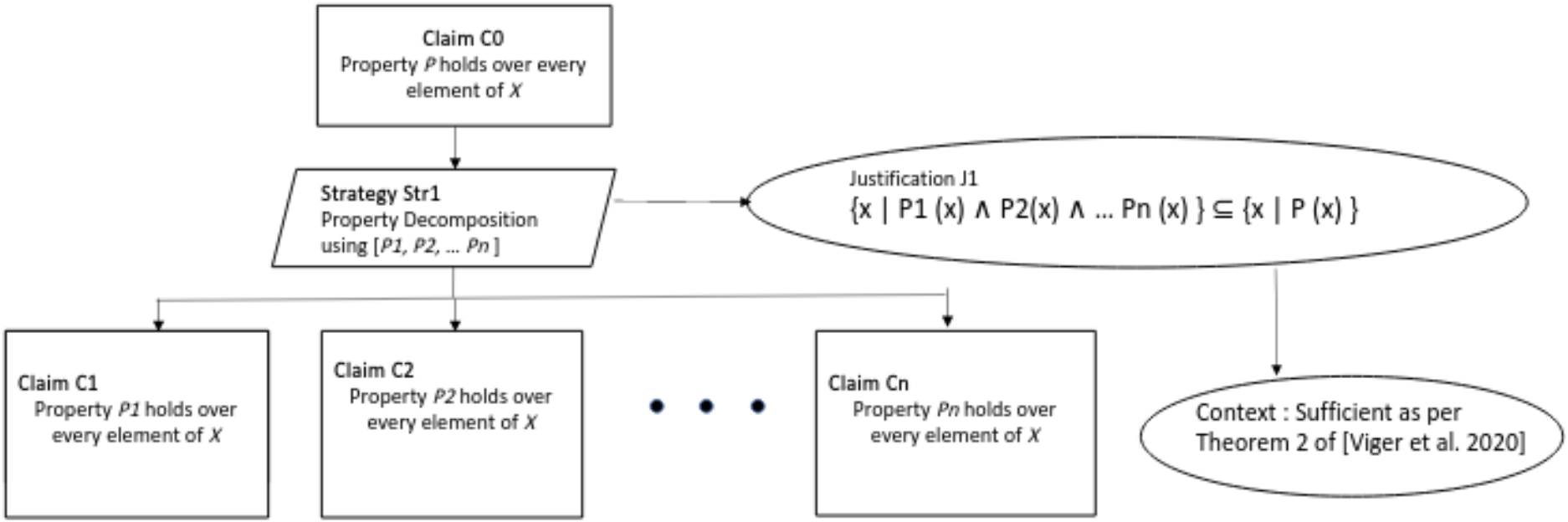
A strategy S is called *deductive* iff

$$(C_1 \wedge C_2 \wedge ... \wedge C_n) \implies C$$

We can identify two useful classes of decomposition strategies: *Domain Decomposition* and *Property Decomposition*.

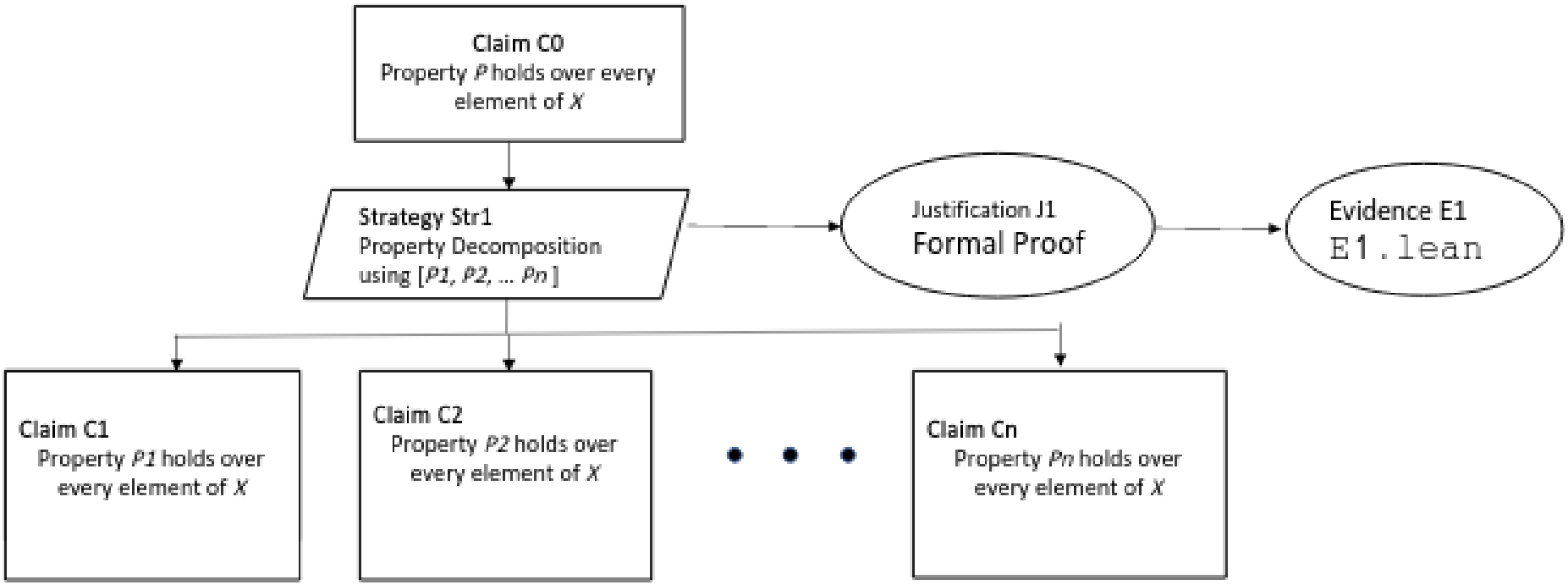
Previous Work: Formality in Assurance Cases

The purpose of this framework is to provide "templates" for deductive decomposition strategies.



Previous Work: Formality in Assurance Cases

Preferable: use a concrete proof!



Formalization

As expected, it was easy enough to formalize the [Viger et al. 2020] framework in Lean.

```
theorem deductive_of_justfd (Γ : property.auxiliary α) :
justified Γ → deductive α (property.to_strategy Γ)
```

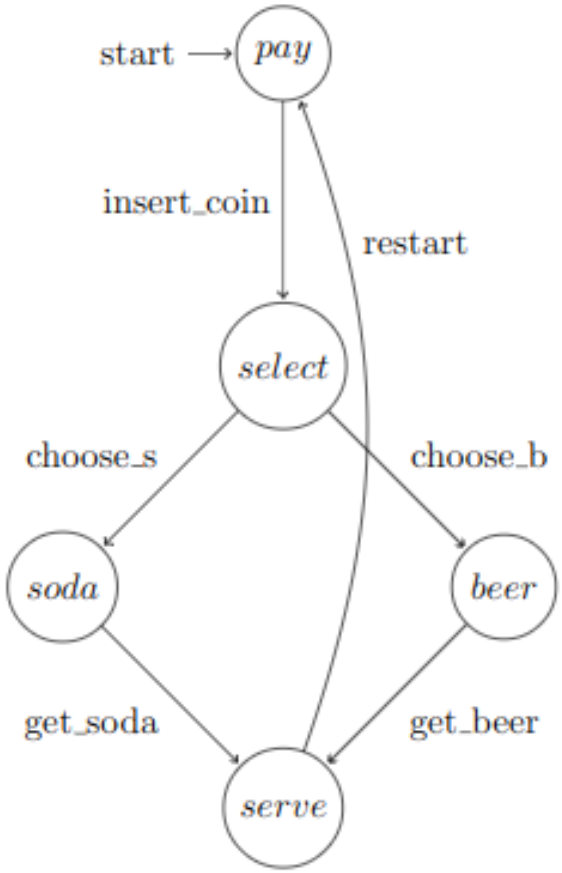
```
theorem deductive_of_justfd_comp  (Γ : domain.auxiliary α β) :
justified Γ → complete Γ →  deductive α (domain.to_strategy Γ)
```

Now, we can try to use this formalization in MMINT-A to facilitate AC verification.

Lean/MMINT-A Integration : What models to use?

Needed to experiment with a class of system models to be the subject of AC claims.

We chose to start with Labelled Transition Systems (LTS's), since they're simple, useful and ubiquitous.

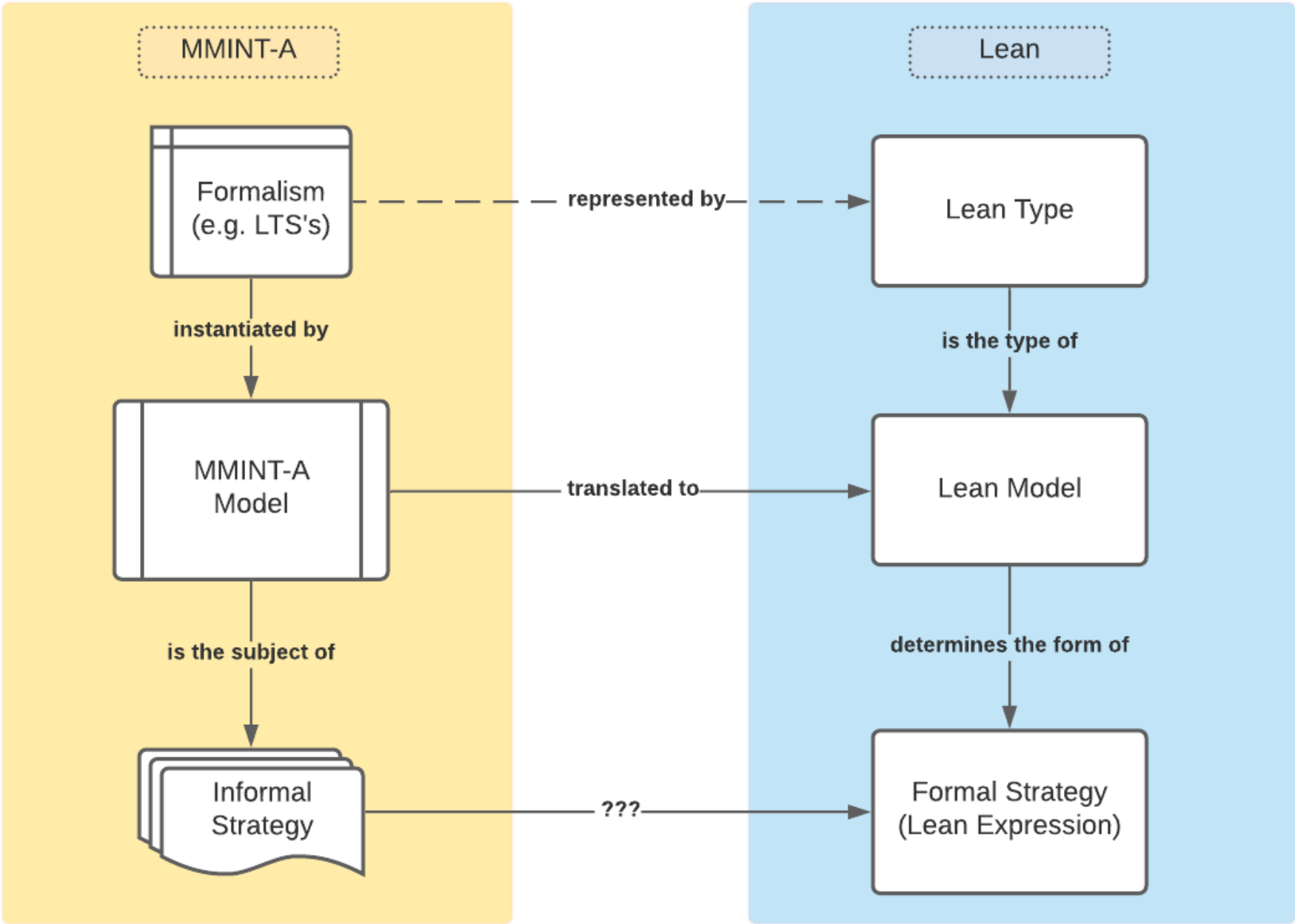


Lean/MMINT-A Integration : Strategy Specification

We can define LTS's in Lean and provide a framework to make claims about their paths using temporal logic properties.

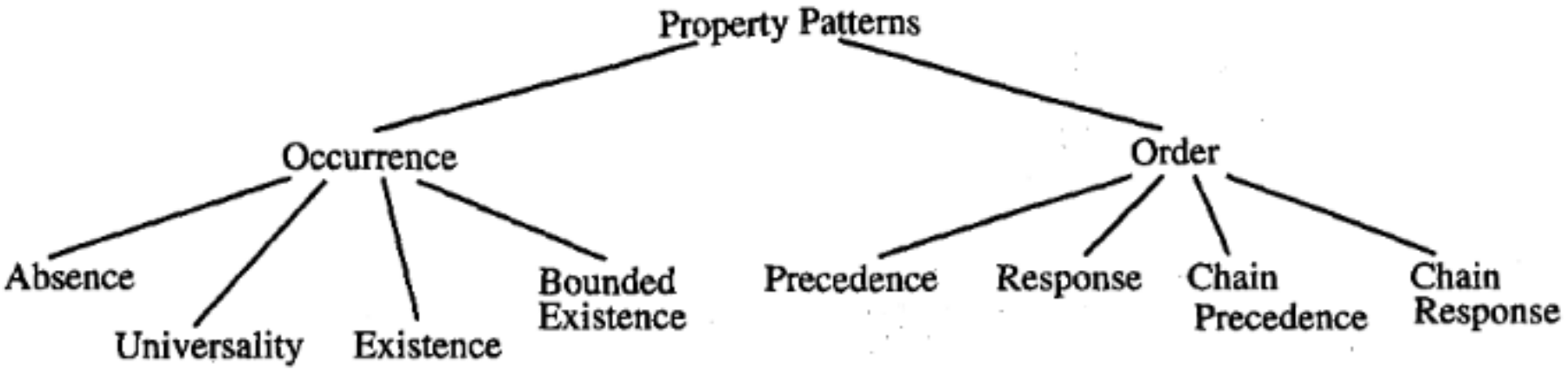
However, we also don't want to force our users to learn to write Lean themselves in order to use MMINT-A.

How can users express a decomposition strategy about their LTS in a way Lean can understand?


MMINT-A
Lean
Formalism
(e.g. LTS's)
represented by
Lean Type
instantiated by
is the type of
MMINT-A
Model
translated to
Lean Model
is the subject of
determines the form of
Informal
Strategy
???
Formal Strategy
(Lean Expression)

Lean/MMINT-A Integration : Strategy Specification

We provide a Lean-encoded "catalogue" of ~ 30 temporal logic formula patterns commonly used in verifying concurrent/reactive systems[8] .



[8] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. *Patterns in property specifications for finite-state verification*. Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002), 1999.

Lean/MMINT-A Integration : Strategy Specification

MMINT-A UI allows the specification of LTS properties without writing Lean code manually.

```
{ property.auxiliary .
  Clm := {
        Claim .
        X := {x : path VM | true},
        P := λ (π : path VM), sat (precedes.globally pay restart) π
           },
  Props := [ λ (π : path VM), sat (absent.before pay serve) π,
             λ (π : path VM), sat (precedes.globally serve restart) π]
  }
```

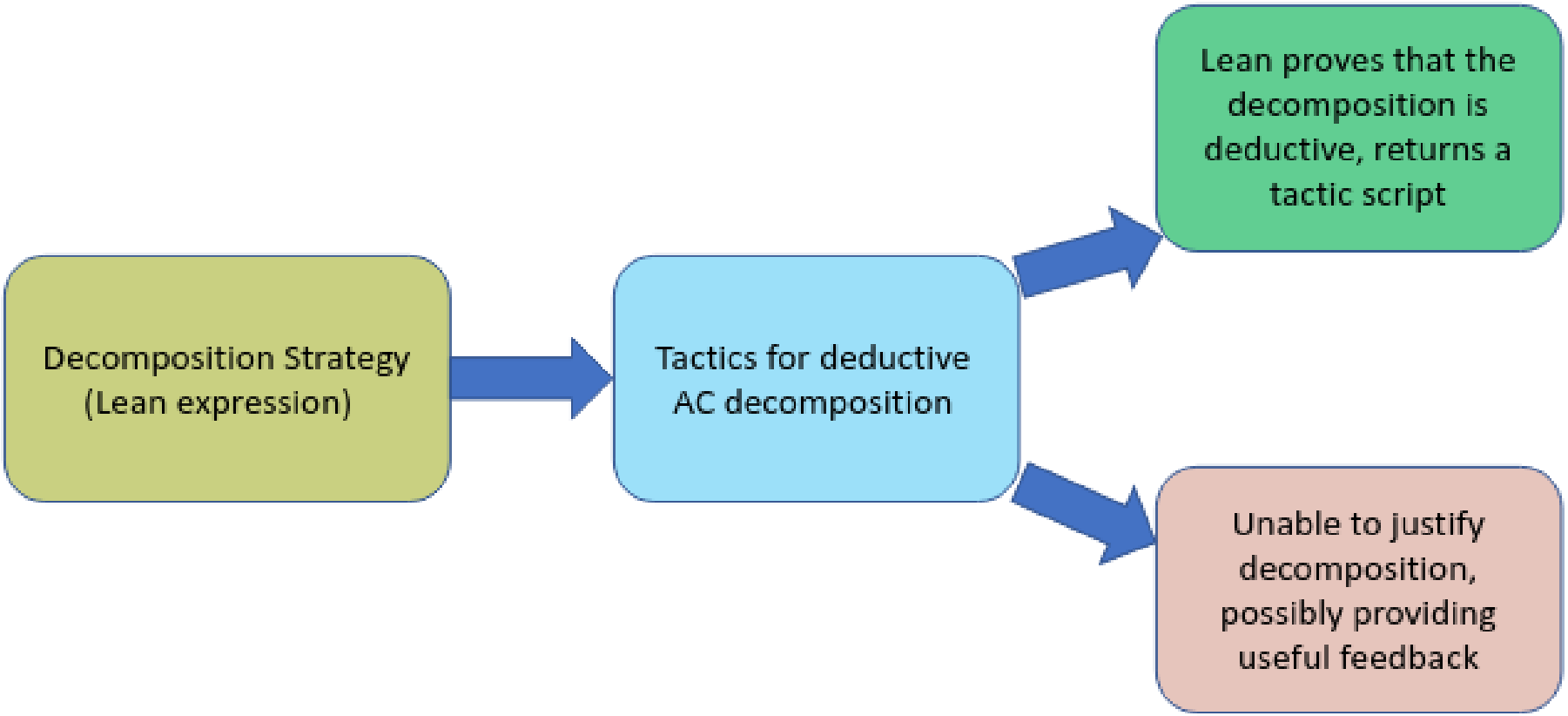
Lean/MMINT-A Integration : Strategy Specification

MMINT-A UI allows the specification of LTS properties without writing Lean manually.

```
{ property.auxiliary .
   Clm := {
         Claim .
         X := {x : path VM | true},
         P := λ (π : path VM), sat ( precedes.globally  pay  restart ) π
            },
   Props := [ λ (π : path VM), sat ( absent.before  pay  serve ) π,
              λ (π : path VM), sat ( precedes.globally  serve  restart ) π]
   }
```

Lean/MMINT-A Integration : Automation

Given a Lean specification of a property decomposition strategy, we want to use Lean's metaprogramming framework to (try to) find a proof that it is deductive.



Lean/MMINT-A Integration : Automation

Since we assume Lean-literate users need the least "help", we've focused on automating strategies using our property catalogue.

Idea : if we assume that all the properties are going to be chosen from our catalogue, we can build a database of proofs using those particular formula patterns.

Lean/MMINT-A Integration : Automation

Example : Suppose "target" property is of the form

```
precedes.globally P Q
-- if Q happens, P must happen first
```

We can prove the following:

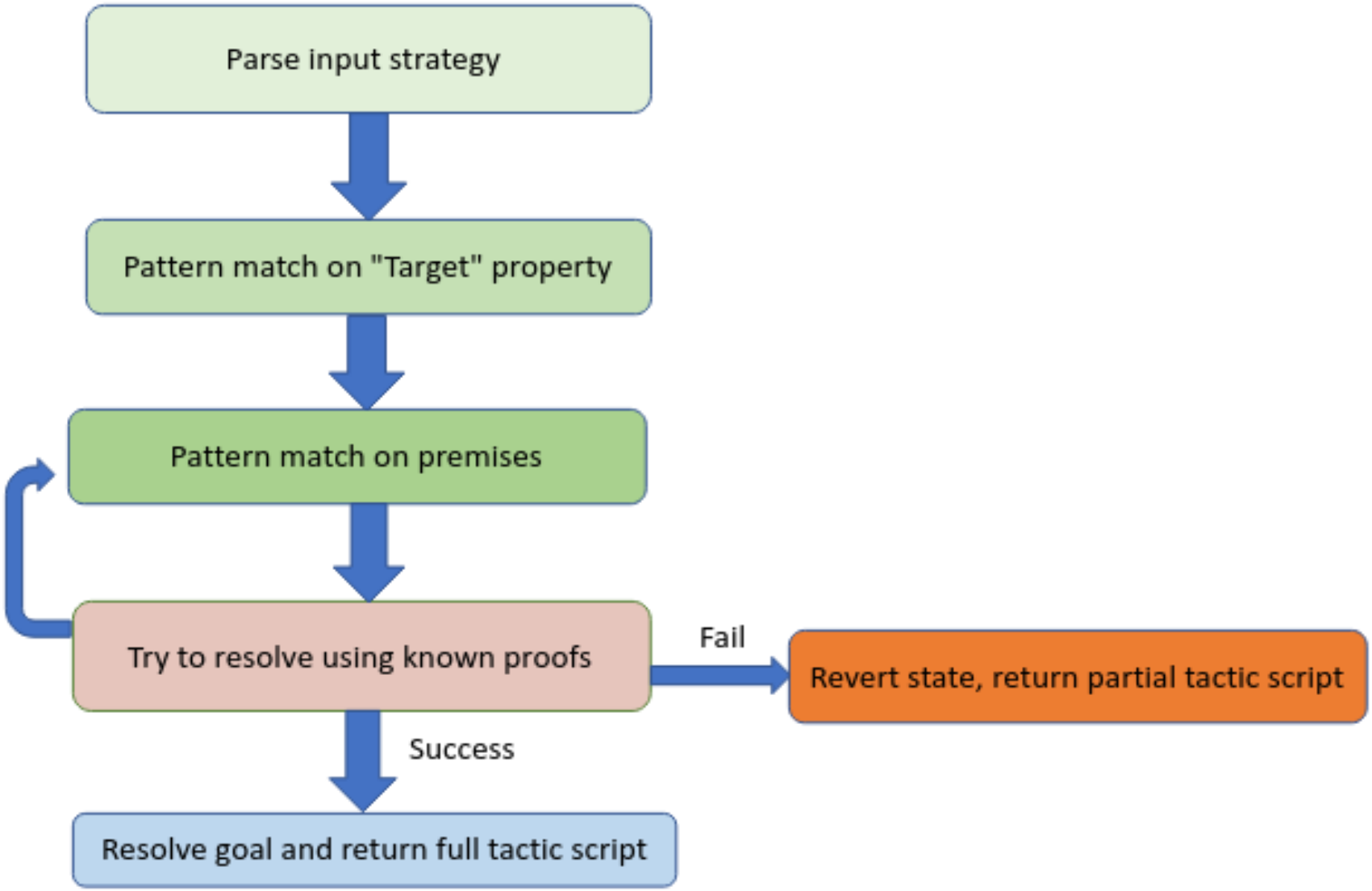
```
variable M : LTS

-- Holds because Q never happens
lemma vacuous (P Q : formula M) (π : path M) :
(sat (absent.globally Q) π) → sat (precedes.globally P Q) π

-- Holds because Q can't happen before P
lemma by_absent_before (P S : formula M) (π : path M) :
(sat (absent.before S P) π) ∧ (sat (exist.globally P) π)
→ sat (precedes.globally P S) π :=

-- Holds because ''precedes'' is transitive
lemma by_transitive (P Q R : formula M) (π : path M) :
(sat (precedes.globally P Q) π ) ∧ (sat (precedes.globally Q R) π)
→ (sat (precedes.globally P R) π) :=
```

Decision Procedure



Decision Procedure

```
meta def switch (str : string) : tactic string :=
 do
 tgt ← tactic.target, ctx ← tactic.local_context,
 match tgt with
 | `(sat (precedes.globally %%tok1 %%tok2) _) :=
    precedes.globally.solve tok1 tok2 str ctx
 | `(sat (absent.globally %%tok1) _) :=
    absent.globally.solve tok1 str ctx
 | ...
 | _ := return string.empty
 end
```

Decision Procedure

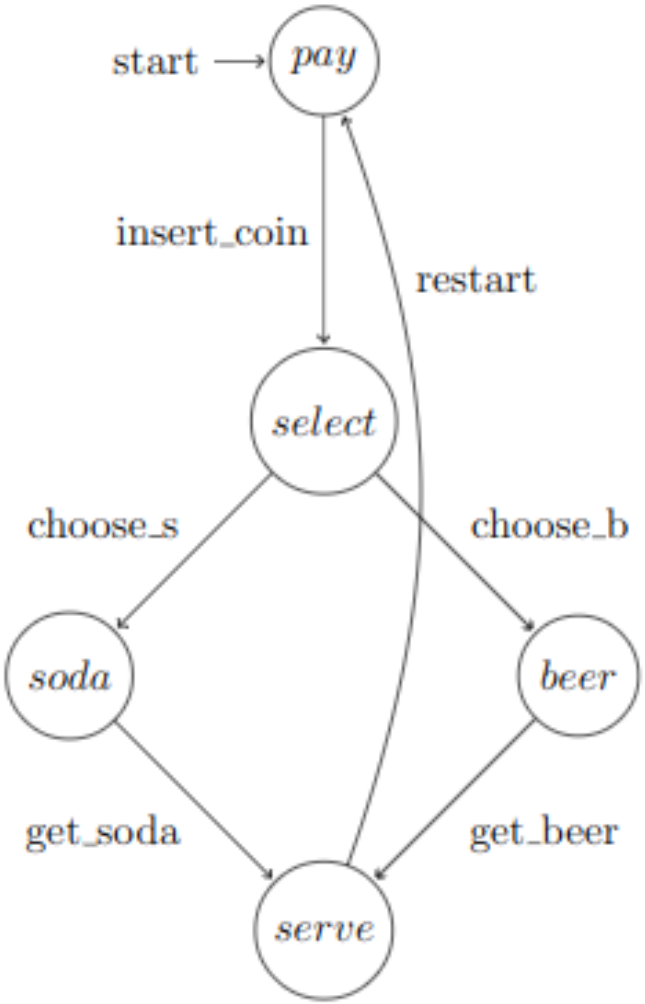
```
meta def solve (tok1 tok2 : expr) (str : string) : list expr → tactic string
| [ ] :=  return string.empty
| (h::t) :=
do typ ← infer_type h,
  match typ with
  | `(sat (precedes.globally _ %%new) _) :=
    solve_by_transitive tok1 tok2 new str < | > solve t
  | `(sat (absent.before _ _) _) :=
    solve_by_absent_before tok1 tok2 str  < | > solve t
  | ...
  | _ := solve t
     end
```

Decision Procedure

```
meta def solve_by_absent_before (tok1 tok2 : expr) (str : string) :
tactic string :=
do
 tactic.interactive.apply ``(by_absent_before %%tok1 %%tok2),
 t1 ← tactic_format_expr tok1,
 t2 ← tactic_format_expr tok2,
 new_str str $
 "apply precedes.globally.by_absent_before " ++
 t1.to_string ++ " " ++
 t2.to_string ++ ",\n"
```

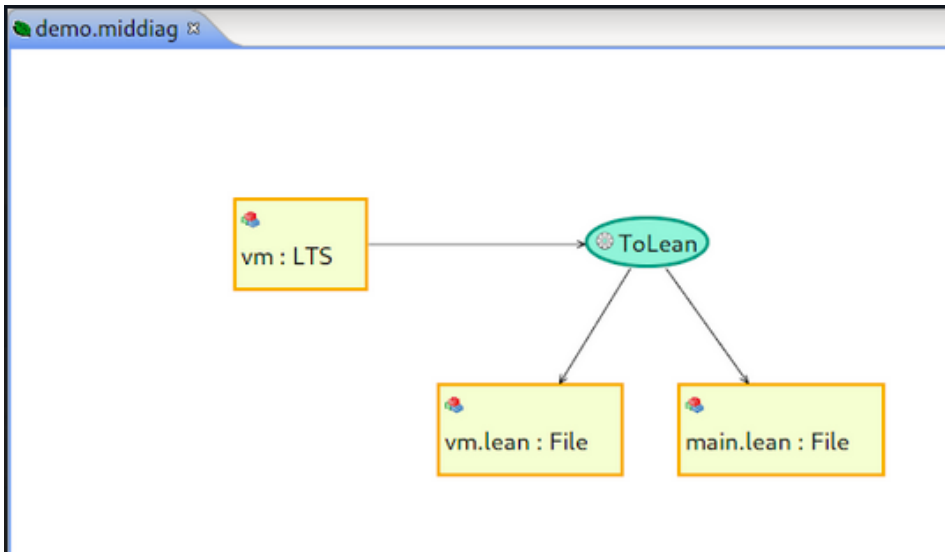
Lean/MMINT-A Workflow Example

Suppose we are a MMINT-A user, and we want to verify a claim decomposition regarding a vending machine.



1. Create a system model in MMINT-A
2. Perform model-to-Lean translation
3. Specify a property decomposition strategy
4. Receive complete (or partial) tactic script

Lean/MMINT-A Workflow Example



1. Create a system model in MMINT-A
2. Perform model-to-Lean translation
3. Specify a property decomposition strategy
4. Receive complete (or partial) tactic script

Lean/MMINT-A Workflow Example

```
{ property.auxiliary .
  Clm := { Claim .
         X := {x : path myLTS | true},
         P := λ (π : path myLTS), sat (precedes.globally pay restart) π
       },
  Props :=
  [λ (π : path myLTS), sat (precedes.globally pay serve) π,
   λ (π : path myLTS), sat (precedes.globally serve restart) π]
}
```

1. Create a system model in MMINT-A
2. Perform model-to-Lean translation
3. Specify a property decomposition strategy
4. Receive complete (or partial) tactic script

Lean/MMINT-A Workflow Example

```
example : deductive (path myLTS)
  (property.to_strategy  {property.auxiliary ...}) :=
begin
apply property.deductive_of_justfd {property.auxiliary ...},
rw property.justified,
simp,
rw set.Inter,
intro x,
simp,
intro H,
deconstruct_hyp 2,
apply precedes.globally.by_transitive pay serve restart,
repeat {split},
repeat {assumption}
end
```

1. Create a system model in MMINT-A
2. Perform model-to-Lean translation
3. Specify a property decomposition strategy
4. Receive complete (or partial) tactic script

Conclusion: Contributions & Lessons Learned

What have we contributed?

- Formalizations (ACs, LTS's, LTL) - although simple and not necessarily novel
- Methodology for integrating Lean/proof assistants in AC management
- Case study in leveraging Lean in a novel domain

What have we learned?

- Lean has potential for real contribution in AC domain
- Tradeoffs: Usability, integration challenges, expressiveness, automation...
- Lean & Meta-Lean

Conclusion: Future Work

- Improve MMINT-A UI, improve metaprogramming, generalize beyond LTS's
- How to use evidence generated by Lean in real-world ACs
- New formalizations of AC fragments
- Different uses for Lean/similar tools in AC managemet
- Lean 4?