|galois|

Formalizing AADL in the Unifying Theories of Programming

ADEPT 2023 (Lisbon, Portugal)

Joe Kiniry and Frank Zeyda (Galois, U.S.)

16 June 2023

Overview

- Rigorous Digital Engineering
- Some Example RDE Projects
- Challenges for Semantic Integration
- A Brief Introduction to UTP
- Application of UTP to Formalizing AADL
- Current State of Work
- Future Outlook and Conclusion

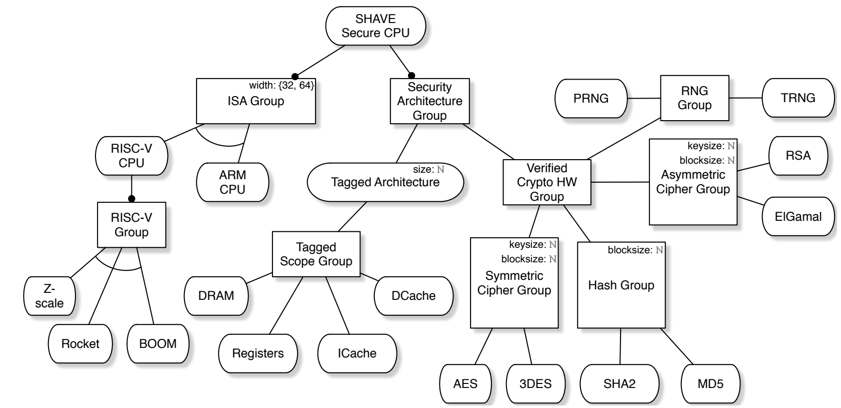
RDE: The Big Picture

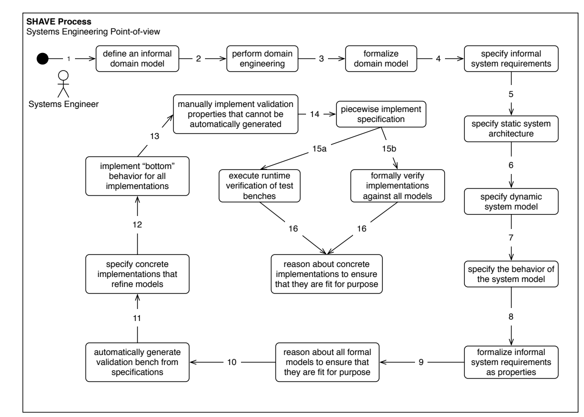
- At Galois we design, build, and assure high-assurance systems using a development process and methodology we call *Rigorous Digital Engineering*, or **RDE** for short.
- **RDE** enables software, hardware, and systems engineers to use formal methods (FM) without really knowing they are doing FM—what we call *Secret Ninja Formal Methods* (**SNFM**).
- Doing **RDE** with **SNFM** means precisely describing what a system is meant to do by stating what properties it must have, and demonstrating that the system conforms to that description—aka writing *specifications* and performing (rigorous) *validation* and (formal) *verification*.
- But any complex system requires writing specifications in several different specification languages—AADL among them—and these specifications all inter-relate to each other, and thus at its core we have a *semantic integration challenge*.

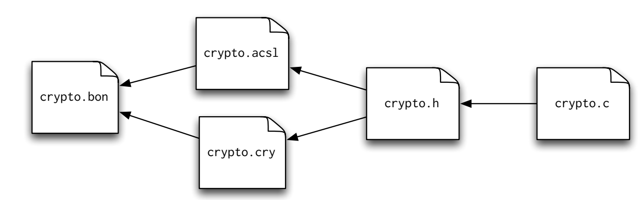
We hypothesize that Unified Theories of Programming (UTP) will help us practically and foundationally to solve this semantic integration challenge.

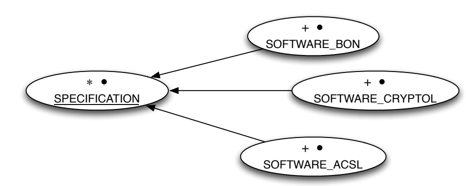
Rigorous Digital Engineering

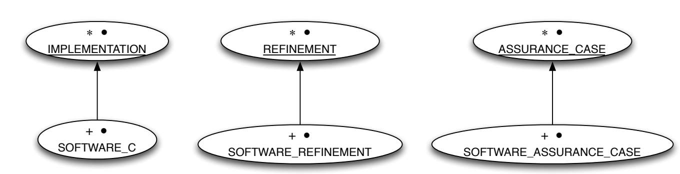
- Rigorous Digital Engineering (**RDE**) is all about…
  - the use of (preferably executable) models (with preferably known fidelity) to
  - rigorously, authentically describe things
  - at various levels of abstraction
  - such that the models relate to each other
  - in well-understood ways
  - and the models refine to bits or atoms
  - and thus all of this connects to software, hardware, and systems engineering
  - and we use the models to provide assurance of various kinds for the product line / product /platform / system











...with Applied Formal Methods

- applied formal methods is about the practical application of formal methods to **all stages** of a system's life cycle:
  - ➡ process, methodology, design, development, assurance, maintenance, and evolution.
- hold no bias in choice of formal method, tool, or technology—just choose the right tool for the job
- often focuses on finding key places where small changes to the lifecycle have large impact
- and nearly always hides formalism from the typical user a la *Secret Ninja Formal Methods*

The Technologies of RDE

The technology stacks supported thus far by the RDE methodology include:

- many different kinds of programming languages (procedural, object-oriented, functional, hardware, logic, and mixed-model, such as C, C++, C#, Rust, Haskell, Java, Scala, Kotlin, Eiffel, Chisel, Bluespec SystemVerilog (BSV), System Verilog, VHDL)
- specification and modeling languages (such as F*, ACSL, JML, CodeContracts, Alloy, Z, VDM, Event-B, RAISE)
- architecture specification tools and languages (such as Cameo, Rhapsody, MagicDraw, OSATE, Visual Paradigm and UML, AADL, and SysML, resp.)
- integrated development environments (such as Eclipse, Visual Studio, Visual Studio Code, and IntelliJ IDEA)
- formal modeling and reasoning tools (such as Alloy, PVS, Coq, Isabelle, UPPAAL, CZT, Overture, Rodin, Frama-C, SAW, Ivy, TLA Toolbox, FDR4, NuSMV, BLAST, and SPIN)
- operating systems (RTOSs, UNIX variants, seL4, etc.)
- spans systems, hardware (ASIC and FPGA-based), firmware, and software

Some Example RDE Projects

- For example, a couple of medium-sized systems created at Galois with RDE over the past decade are the **SHAVE** and **HARDENS** systems.
    - SHAVE is a bump-in-wire encryption device that includes a soft core CPU, measured boot, and cryptography in hardware and firmware.
    - HARDENS is an Instrumentation and Control (I&C) system for a Reactor Trip System, providing a fault-tolerant protection system for Nuclear Power Plants.
- SHAVE includes nearly a dozen specification and programming languages (ACSL, Aoraï, ASM, BSV, C, Cryptol, EBON, LLVM, PVS, SAW, and SV).
- HARDENS includes just over a dozen specification and programming languages (AADL, SysML, ACSL, ASM, BSV, C, Cryptol, FRET, Lando, LLVM, Lobot, SAW, and SV).

Challenges to Semantic Integration

- Relating all of these specifications and implementations—semantically and practically—is currently *fully supported by the RDE process and methodology*, but only *partly by automated tools*.
- Our work using **UTP** via **SNFM** is meant to provide a mathematical foundation to these relations.

Our goal is ***full, invisible automated tooling*** *for: model-model/model-code* ***refinement checking****,* ***extraction*** *of model/code refinements from models,* ***lifting*** *of abstractions from models and code.*

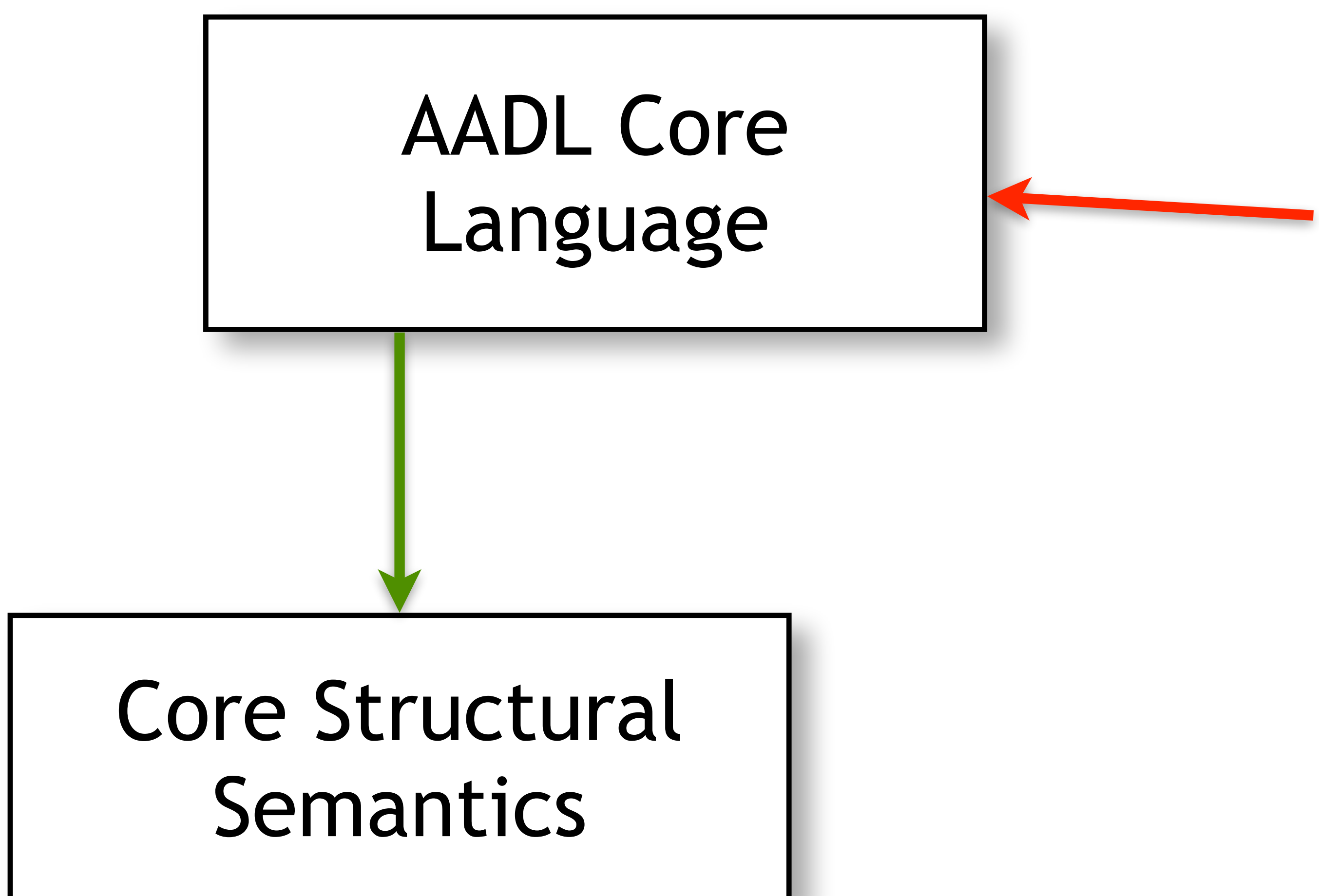
This work is supported by Government grants from many agencies, including AFRL, ARL, DARPA, EC, NRC, NSA, SDA, SFI, and others.

Anatomy of AADL Semantics

AADL Core Language

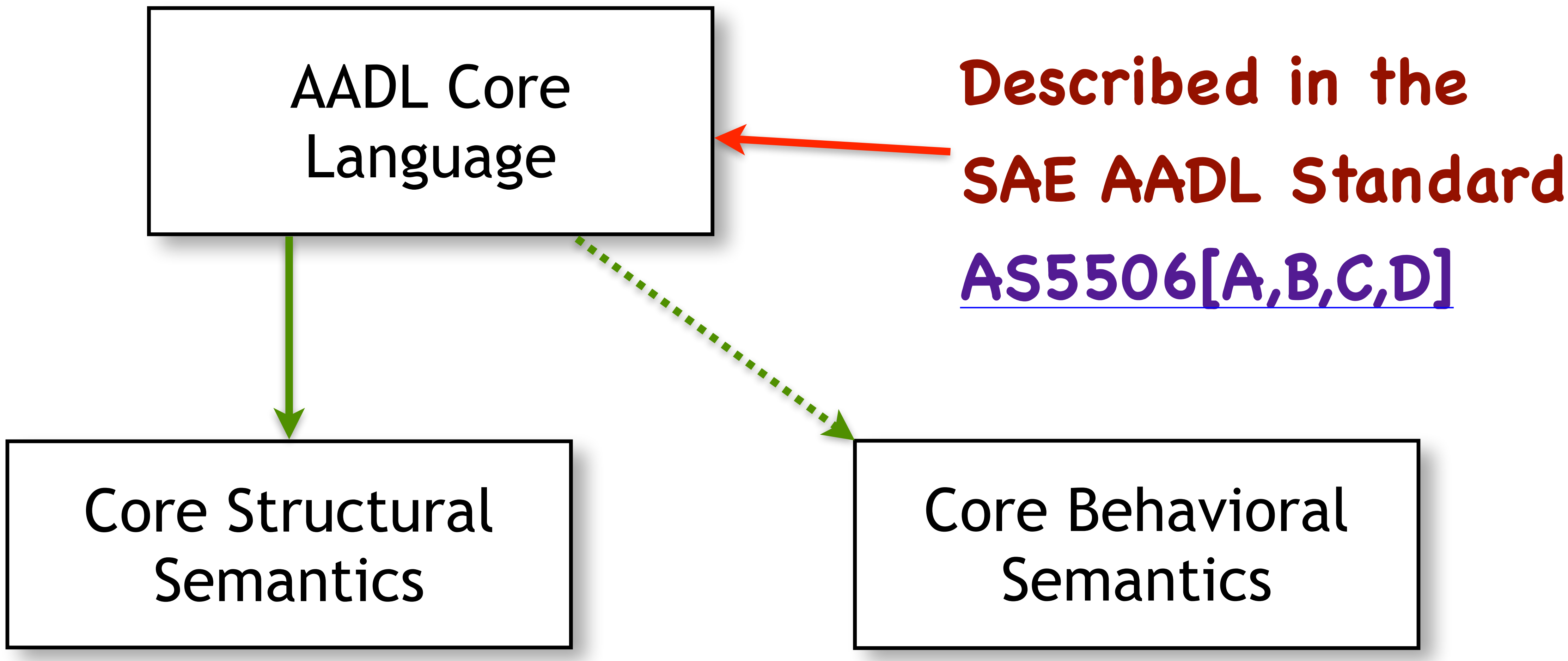
Described in the SAE AADL Standard [AS5506[A,B,C,D]]

Anatomy of AADL Semantics

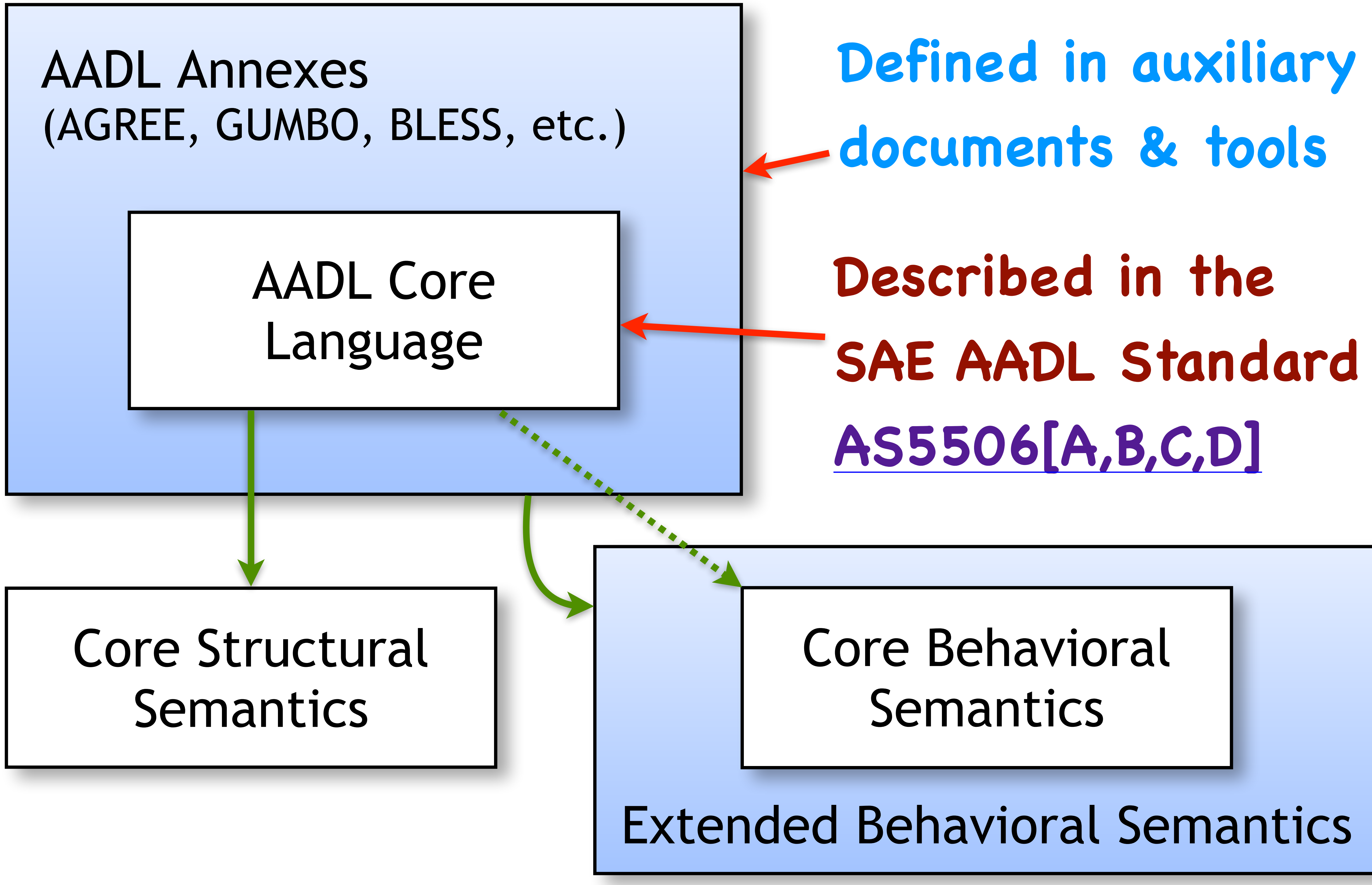


Described in the SAE AADL Standard AS5506[A,B,C,D]

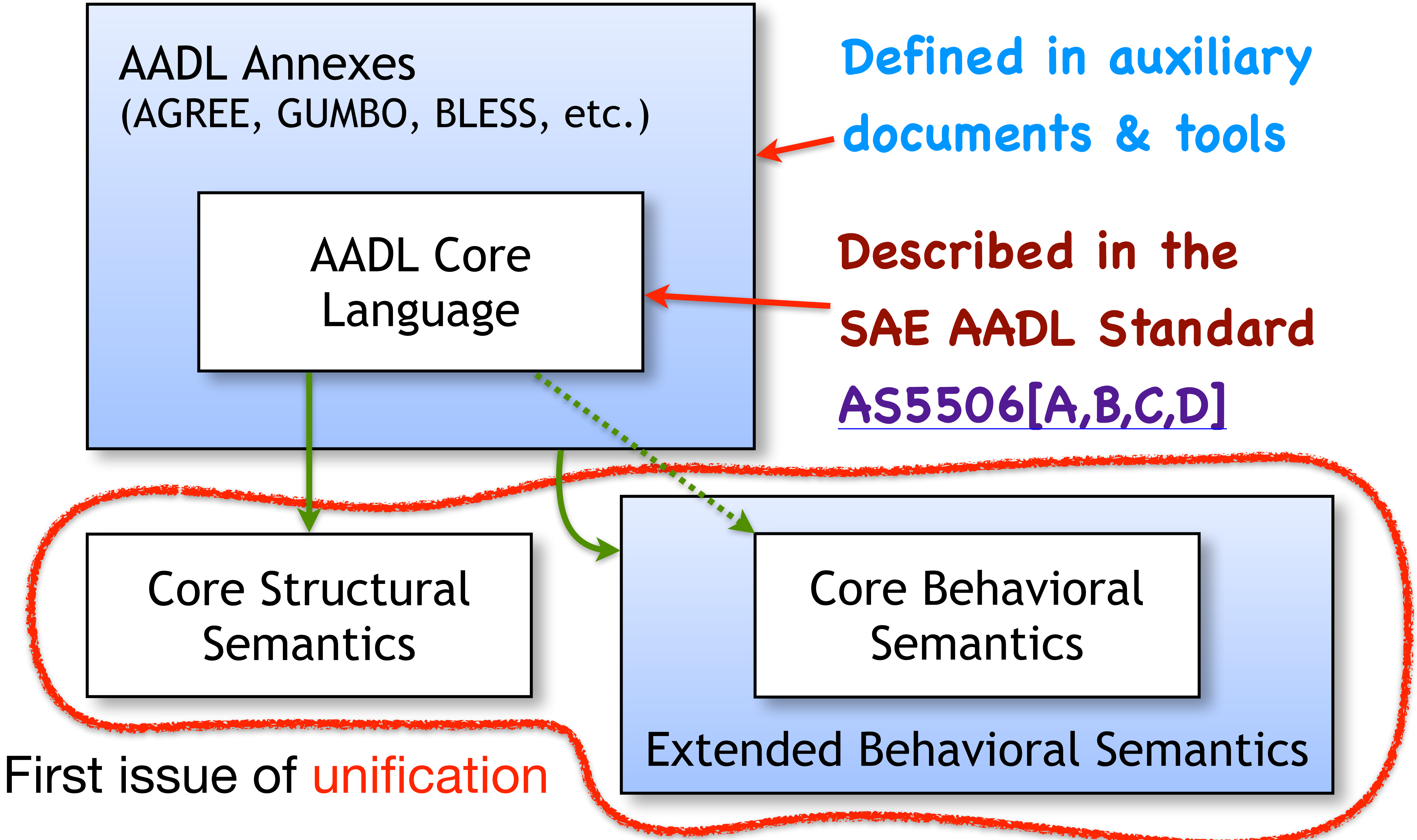
Anatomy of AADL Semantics



Anatomy of AADL Semantics



Anatomy of AADL Semantics



Concerns for Subtheories

- **Core** Structural Semantics
  - well-formedness of AADL models; i.e.,
  - naming, legality and consistency rules
- **Core** Behavioral Semantics
  - reactivity and communication
  - timing and scheduling behavior
  - guarantees made by a run-time framework

?

- **Extended** Behavioral Semantics
  - inclusion of BISL and contract frameworks
  - embedding of a refinement calculus with a guarded command language for expressing implementations
  - whatever formal model a particular **annex** requires …

A Vision for Semantic Integration

- Extensions of the AADL language (via annexes, custom properties, and so on …) are mirrored by an extensions to the (core) semantics.
- As syntactic entities and concepts are referenced and reused, so are formalized semantic ones.
- Requires a certain degree of **modularity** and **compositionality** of the semantic framework.
- Verification notions, such as refinement **change** (become stronger) as we specialize the language.
- **Question**: How to mechanize all this in a theorem prover in a **plug-and-play** fashion?

A Word on Refinement

- Refinement is a formal (mathematical) **relationship** between specification and their implementations.
  - E.g., S ⊑ T *logically* means that T is a valid implementation of specification S.
  - This ought be a provable/falsifiable statement.
- The distinction between **specifications** and their **implementations** is already present in AADL.

- Hence, AADL ought to lend itself well for integration into refinement-centric reasoning techniques.
- Hoare's Unifying Theories of Programming (**UTP**) is one such a technique (and more) …

UTP in a Nutshell

- Proposed in Tony Hoare and He Jifeng's seminal book "Unifying Theories of Programming" (1998).
- Presents a **unified framework** in which the semantics of specification, design, modeling, and programming languages of *any* kind and flavor can be uniformly described.
- Inspired by scientific / engineering theories:
  - theories describe "observable behaviors"
  - consider Boyle's law: $PV = k$ (pressure multiplied by volume equals to some constant k)
  - UTP computations are *in essence* **predicates**

UTP in a Nutshell

- Proposed in Tony Hoare and He Jifeng's seminal book "Uni… …g" (1998).
- Presents a… …the semantics… …deling, and programm… …d flavor can be uniform…

Think of observing the interactions of an AADL component through its ports with an environment.

- Inspired by scientific / engineering theories:
  - theories describe "observable behaviors"
  - consider Boyle's law: $PV = k$ (pressure multiplied by volume equals to some constant k)
  - UTP computations are *in essence* **predicates**

Observable Qualities

- Observable qualities are defined by the **alphabet** of a UTP predicate (αP):
  - they can be program variables …

  - … or auxiliary variables of a **computational paradigm** such as:
  - **ok** : $\mathbb{B}$, **tr** : seq(Event), **ref** : $\mathcal{P}$(Event), and so on.
- In AADL, we, e.g., have a variable that records the topological structure of a model.
  - leaving suitable “gaps” for additional semantic information in subtheories (extend alphabet) …

Observable Qualities

- Observable qualities are defined by the **alphabet** of a UTP predicate (αP):
  - they car
  - … or au … **ational paradig**
  - **ok** : $\mathbb{B}$, … and so on.

A wealth of UTP theories already exists for common sequential, reactive and hybrid prog. notations.

- In AADL, we, e.g., have a variable that records the topological structure of a model.
  - leaving suitable “gaps” for additional semantic information in subtheories (extend alphabet) …

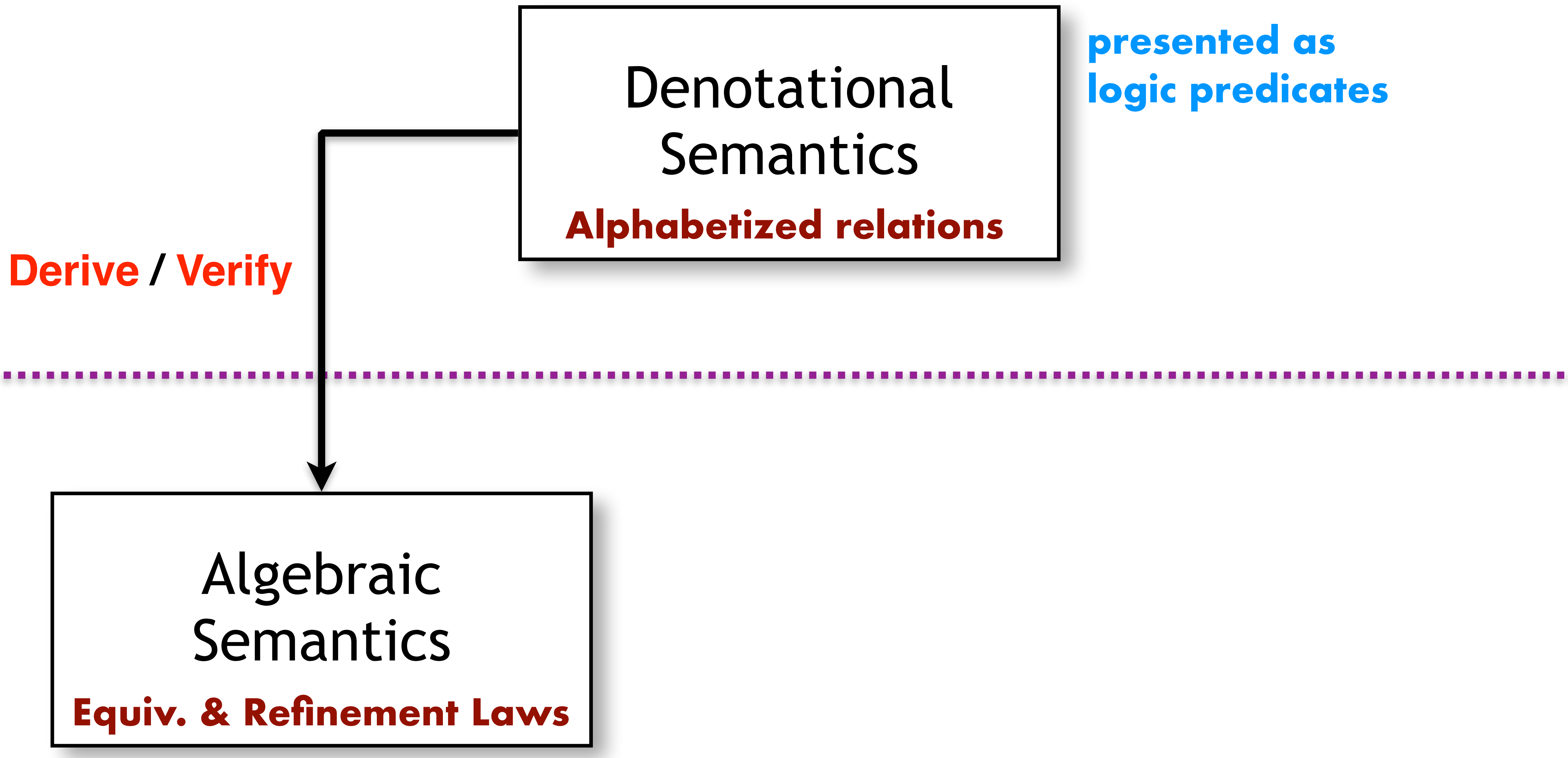
UTP Semantic Triangle

Denotational Semantics

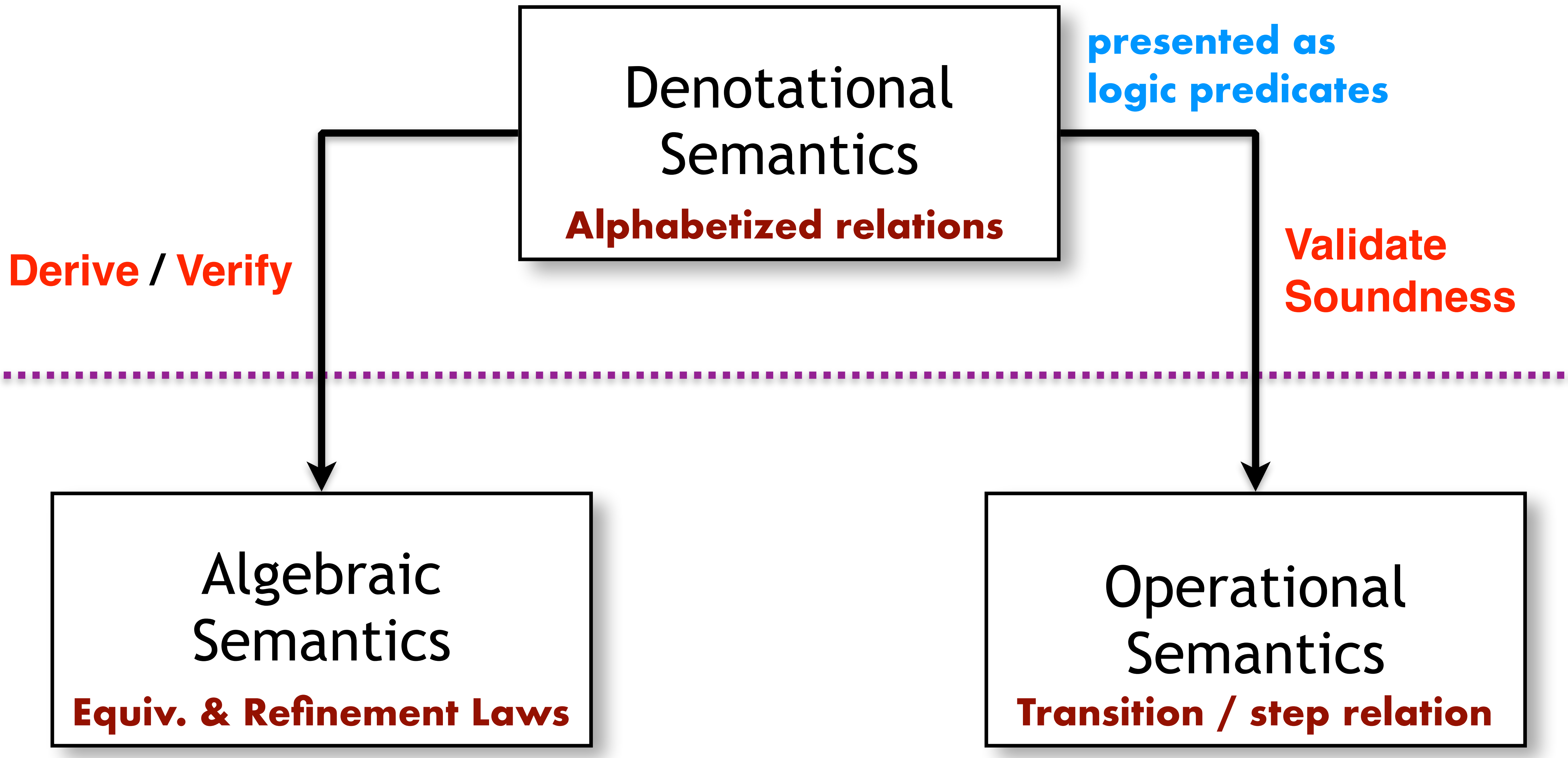
Alphabetized relations

presented as logic predicates

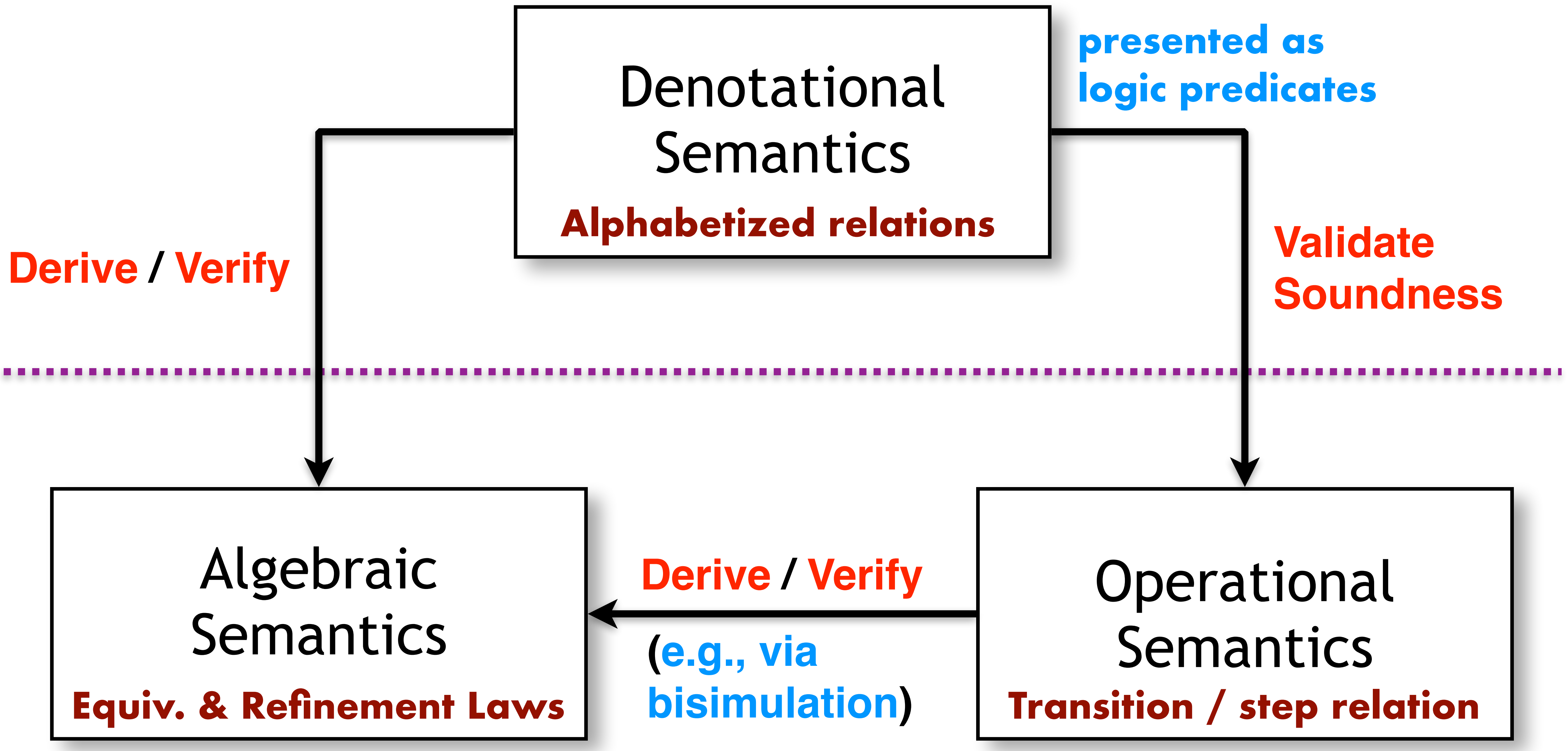
UTP Semantic Triangle



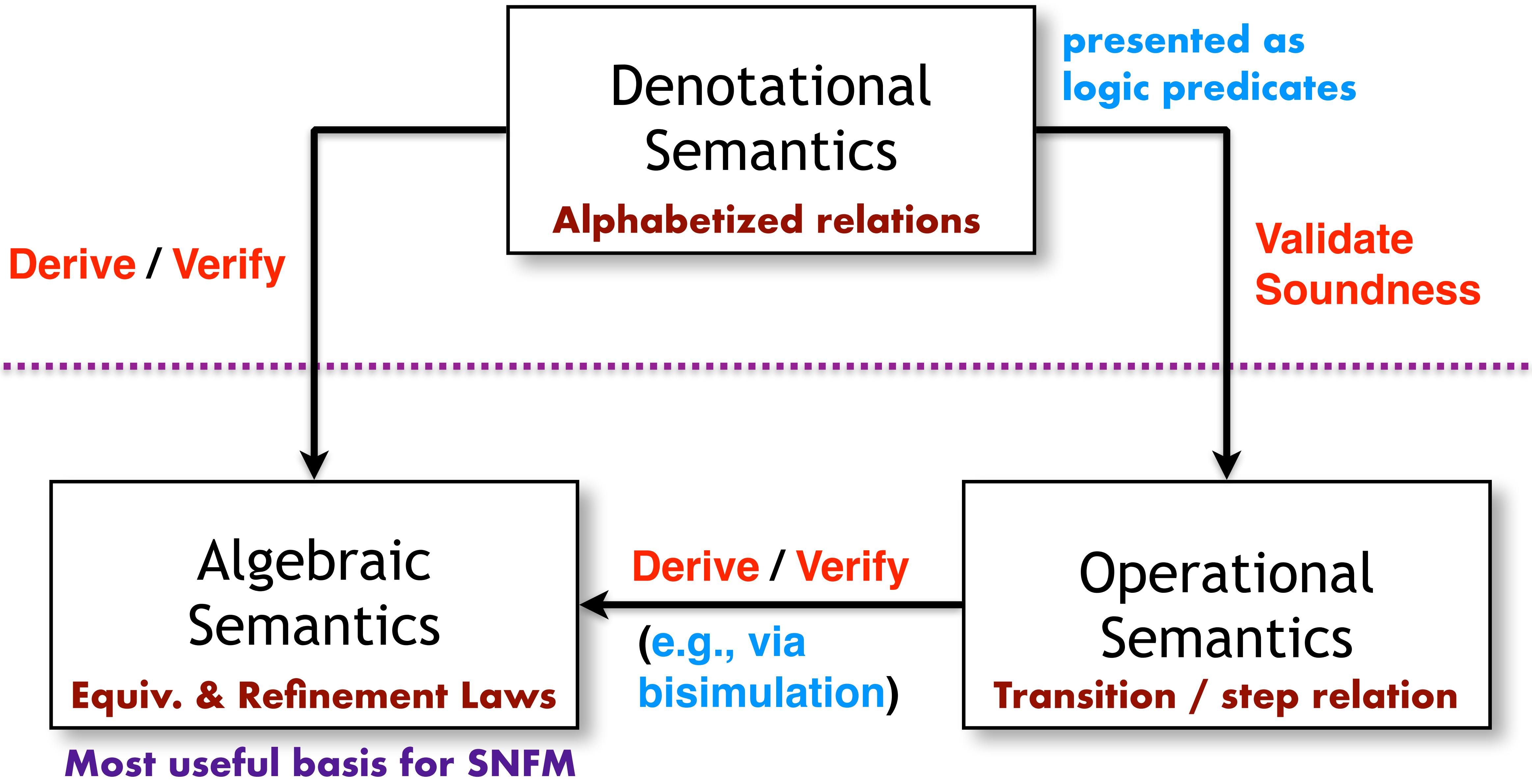
UTP Semantic Triangle



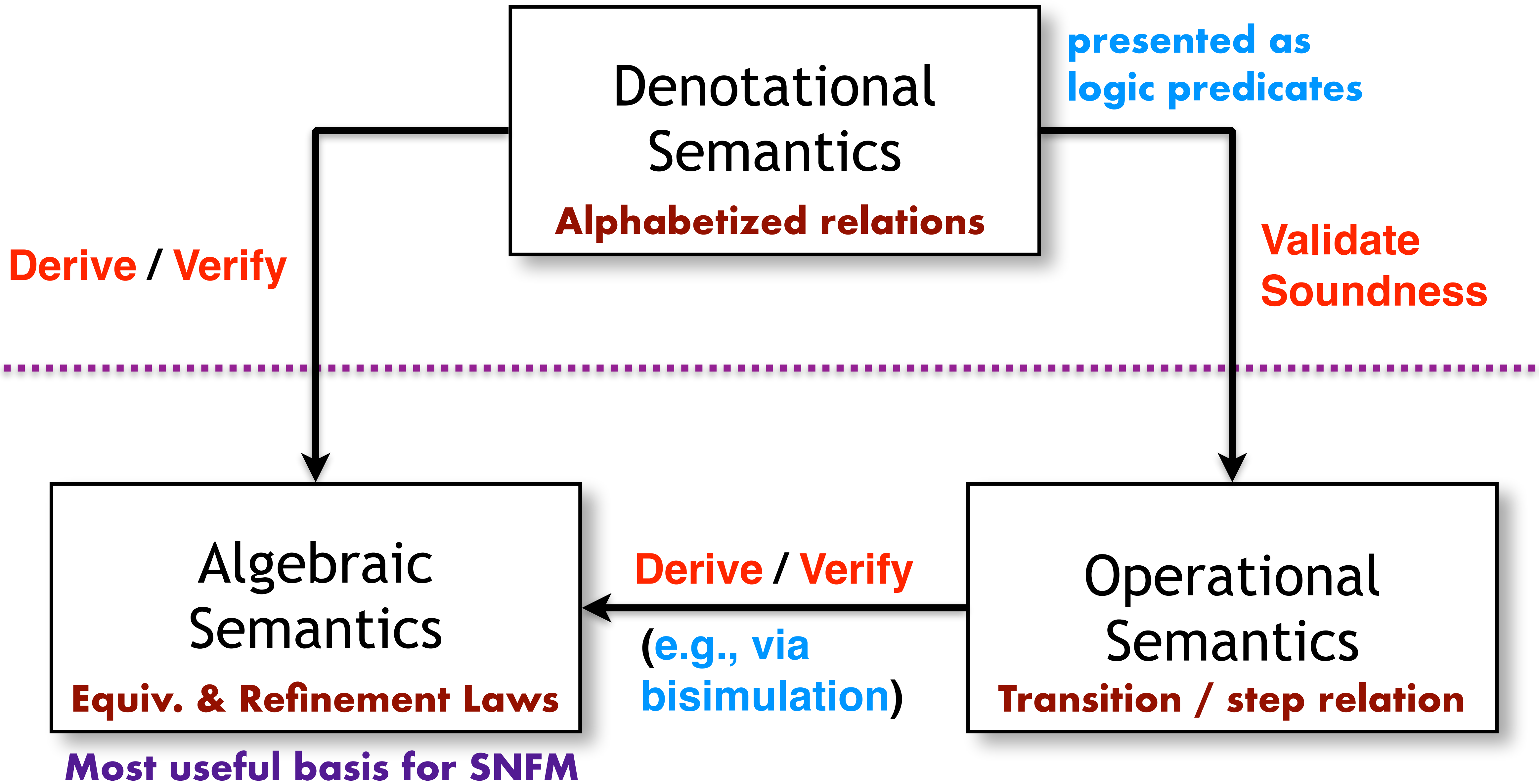
UTP Semantic Triangle



UTP Semantic Triangle

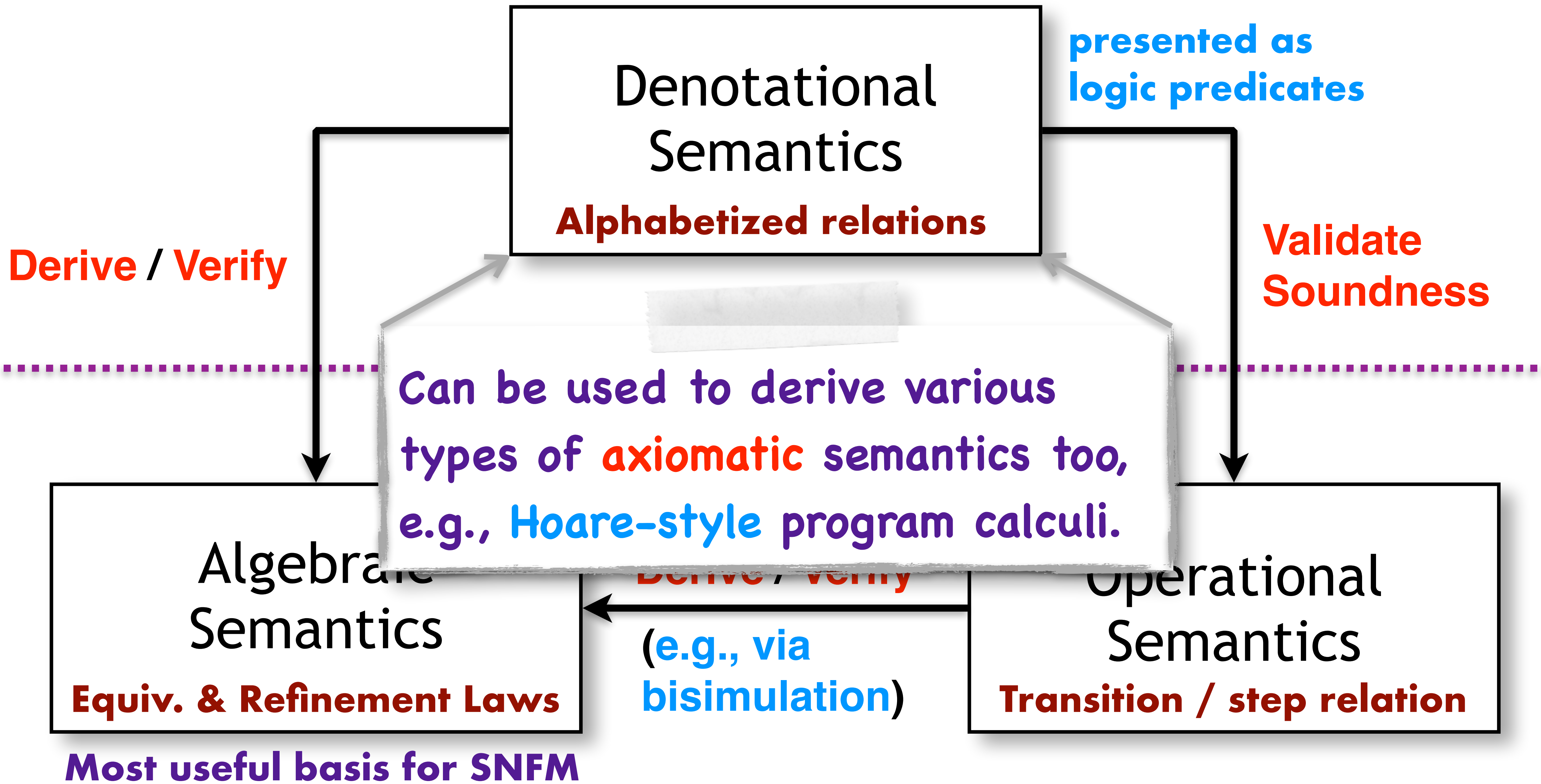


UTP Semantic Triangle



- Denotational semantics is considered the “gold standard” and point of reference for other semantic presentations.

UTP Semantic Triangle



- Denotational semantics is considered the "gold standard" and point of reference for other semantic presentations.

Review of Semantic Approaches

- **Denotational** semantics:
  - encapsulated by UTP theories ("**healthy**" predicate sets);
  - copes well with everything: iteration, recursion, non-determinism, refinement, and compositional development;
  - but carries the heavy burden of a mathematical model with it.
- **Algebraic** semantics:
  - especially useful for refactoring, refinement, code generation and optimization, as well as pattern-based design;
  - may be incomplete and less tractable in axiomatic frameworks.
- **Operational** semantics:
  - mimics abstract execution: more natural and intuitive
  - implicitly provides complexity measure (number of steps)
  - but more difficult to deal with iteration, nondeterminism and refinement (requires notion of bisimulation in proofs) …

Review of Semantic Approaches

- **Denotational** semantics:
  - encapsulated by UTP theories (“**healthy**” predicate sets);
  - copes well with everything: iteration, recursion, non-determinism, refinement, and compositional development;
  - but carries the heavy burden of a mathematical model with it.
- **Algebraic** semantics:
  - especially useful for refactoring, refinement, code generation and optimization, as well as pattern-based design;
  - may be incomplete and less tractable in axiomatic frameworks.
- **Operational** semantics:
  - mimics abstract execution: more natural and intuitive
  - implicitly provides complexity measure (number of steps)
  - but more difficult to deal with iteration, nondeterminism and refinement (requires notion of bisimulation in proofs) …

Review of Semantic Approaches

- **Denotational** semantics:
  - encapsulated by UTP theories ("**healthy**" predicate sets);
  - copes well with everything: iteration, recursion, non-determinism, refinement, and compositional development;
  - but carries the heavy burden of a mathematical model with it.
- **Algebraic** semantics:
  - especially useful for refactoring, refinement, code generation and optimization, as well as pattern-based design;
  - may be incomplete and less tractable in axiomatic frameworks.
- **Operational** semantics:
  - mimics abstract execution: more natural and intuitive
  - implicitly provides complexity measure (number of steps)
  - but more difficult to deal with iteration, nondeterminism and refinement (requires notion of bisimulation in proofs) …

Review of Semantic Approaches

- **Denotational** semantics:
  - encapsulated by UTP theories (“**healthy**” predicate sets);
  - copes well with everything: iteration, recursion, non-determinism, refinement
  - but carries … model with it.
- **Algebraic** se…
  - especially … le generation and optimizatio…
  - may be inc… tic frameworks.
- **Operational** semantics:
  - mimics abstract execution: more natural and intuitive
  - implicitly provides complexity measure (number of steps)
  - but more difficult to deal with iteration, nondeterminism and refinement (requires notion of bisimulation in proofs) …

We also have axiomatic semantics that translates postulates about programs into logical conjectures to be proved in a FOL/HOL.

A Taste of UTP Models

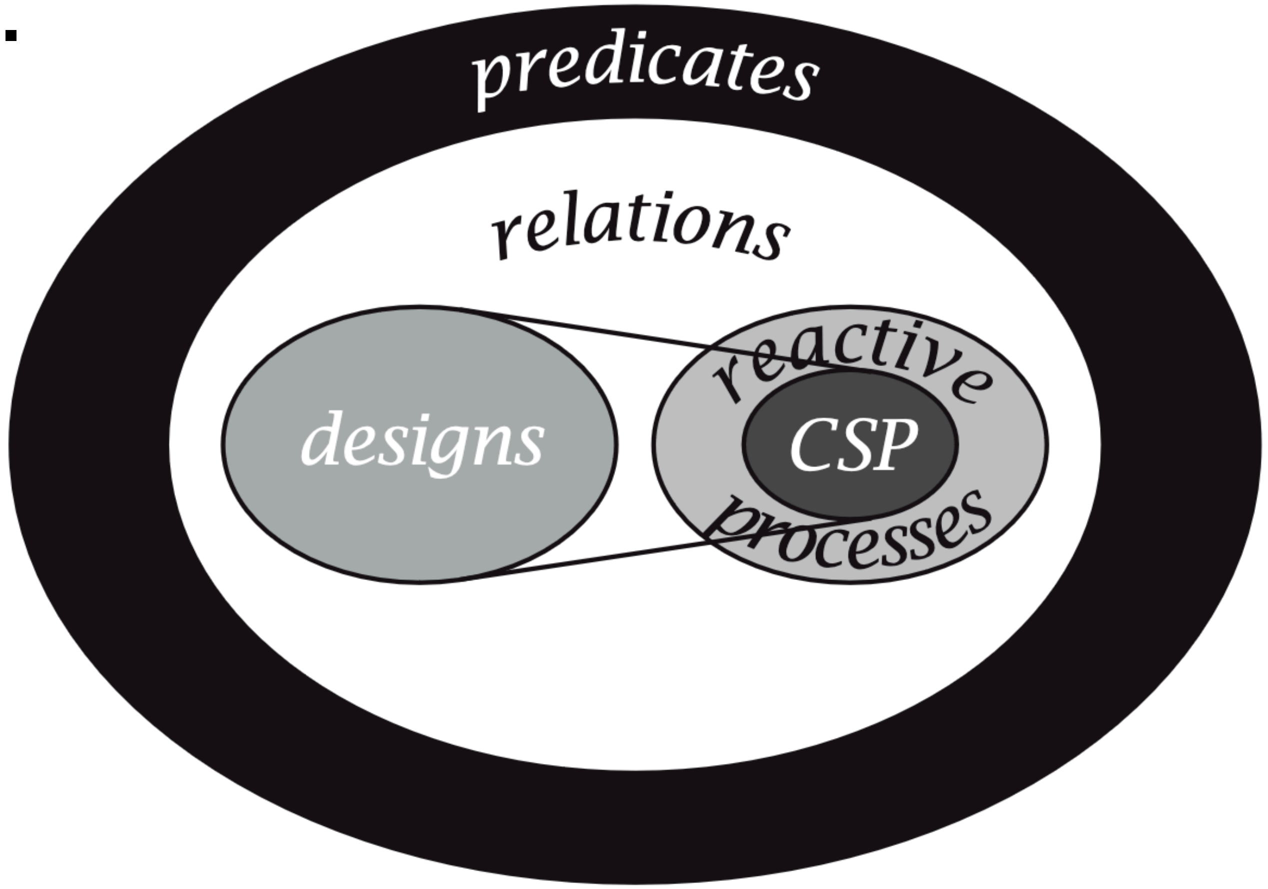
- Relational programs:
  - $x := 42 \stackrel{\text{def}}{=} x' = 42$ (constraining the after-state)
- Total-correctness "designs":
  - $x := y \div z \stackrel{\text{def}}{=} ok \wedge y \neq 0 \;\Rightarrow\; ok' \wedge z' = (x \text{ div } y)$
- Reactive programs (ACP, CSP, *Circus*, etc.):
  - $c \rightarrow skip \stackrel{\text{def}}{=}$
    $R(tr' = tr \wedge c \notin ref' \lhd wait' \rhd tr' = tr \mathbin{\hat{}} c\langle\rangle)$

A Taste of UTP Models

- Relational programs:
  - $x := 42 \stackrel{\text{def}}{=} x' = 42$ (constraining the after-state)
- Total-correctness "designs":
  - $x := y \div z \stackrel{\text{def}}{=} ok \land y \neq 0 \;\Rightarrow\; ok' \land z' = (x \text{ div } y)$
- Reactive programs (ACP, CSP, *Circus*, etc.):
  - $c \rightarrow skip \stackrel{\text{def}}{=}$
    $\mathbf{R}(tr' = tr \land c \notin ref' \lhd wait' \rhd tr' = tr \mathbin{\hat{}} \langle c \rangle)$
- **NOTE**: We only write the LHS. The RHS is typically hidden from the user and managed by the prover.

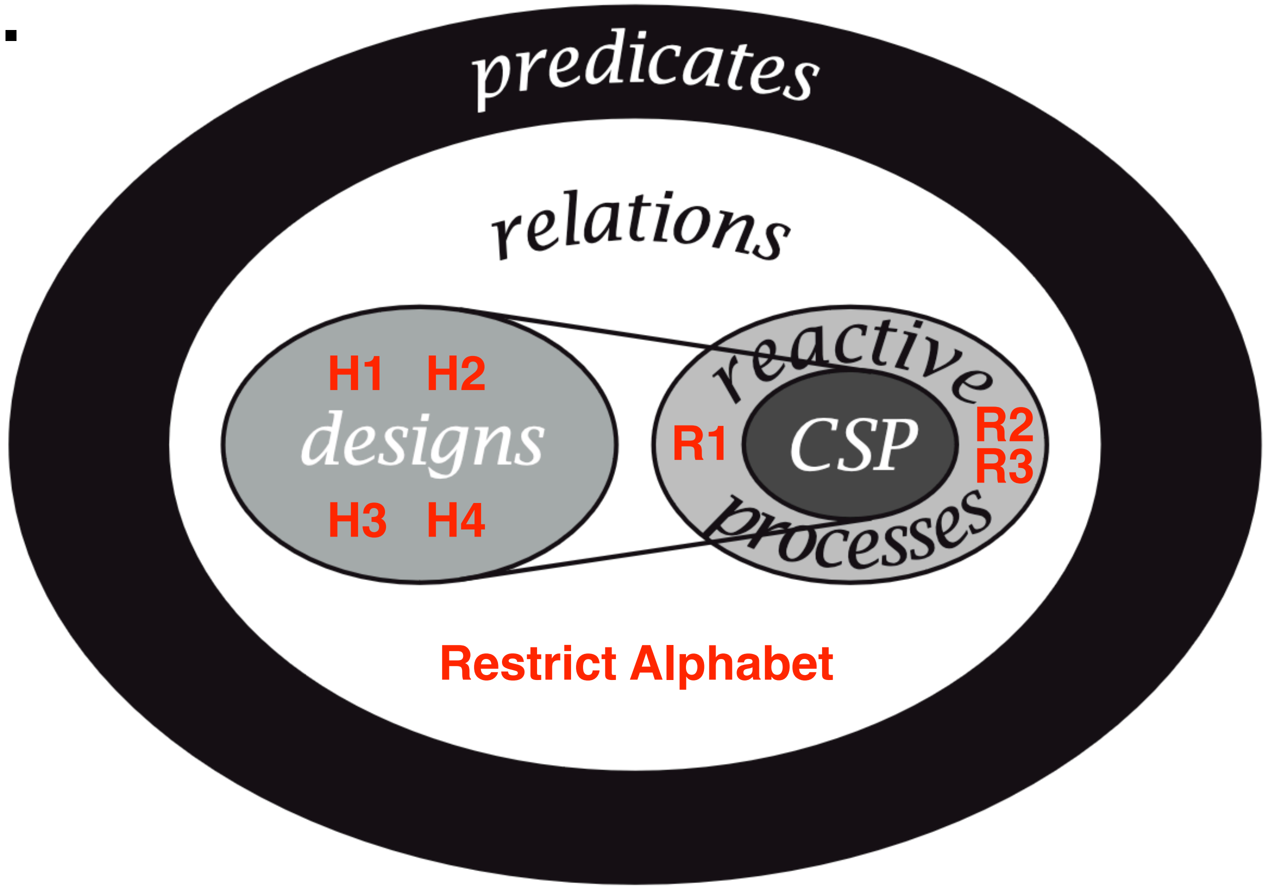
UTP Theories

- UTP Theories are characterized by **healthiness conditions** (HC).
- Define a subset of the permissible predicates.
- Combinators of theories works via their HCs and alphabets.



UTP Theories

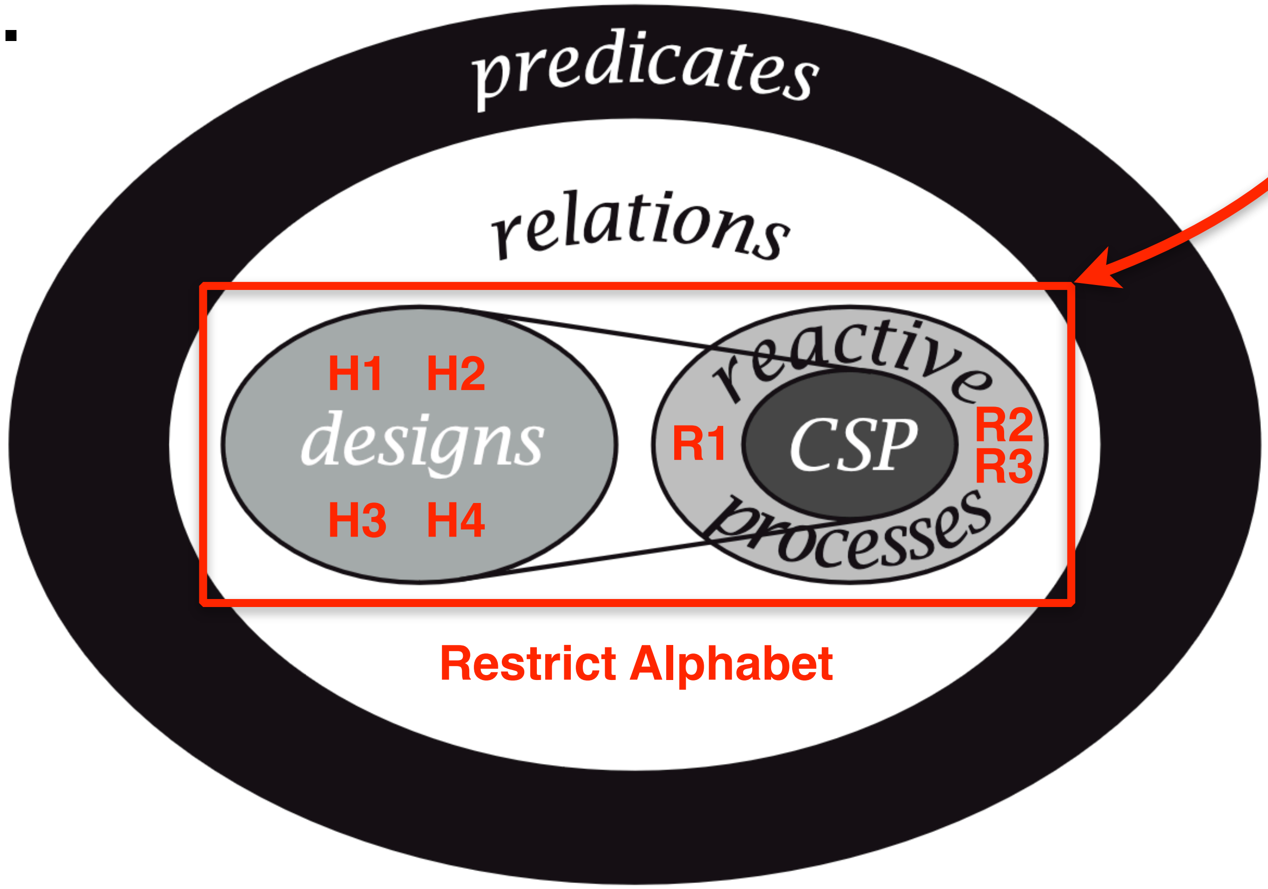
- UTP Theories are characterized by **healthiness conditions** (HC).
- Define a subset of the permissible predicates.
- Combinators of theories works via their HCs and alphabets.



UTP Theories

- UTP Theories are charac[...] **conditions** (HC).
- Define a subset of the pe[...]
- Combinators of theories [...] alphabets.

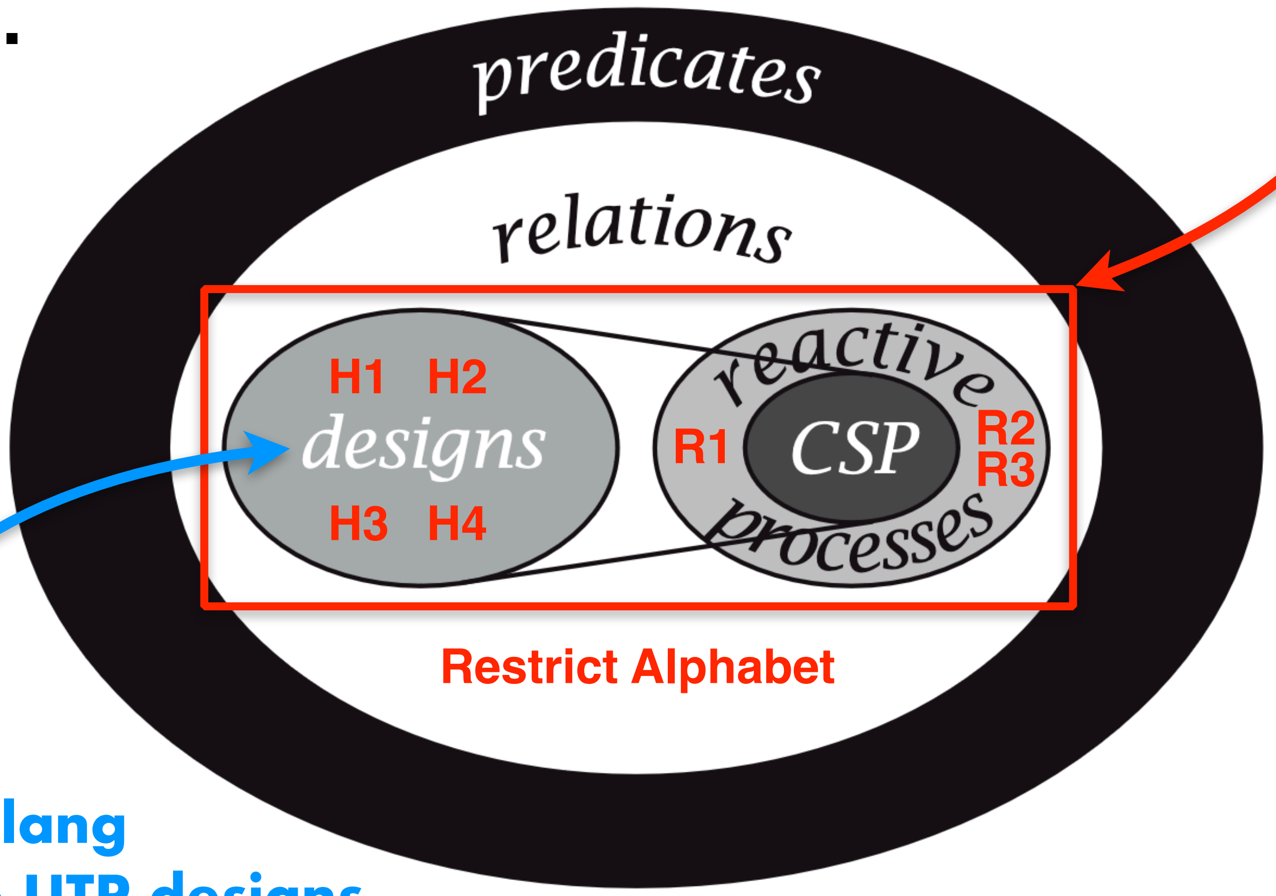
All of these are useful to give a semantic model of AADL components …



UTP Theories

- UTP Theories are charac **conditions** (HC).
- Define a subset of the pe
- Combinators of theories alphabets.

All of these are useful to give a semantic model of AADL components …



The GUMBO/HAMR/Slang semantics is similar to UTP designs

UTP vs AADL Refinement

- AADL's use of refinement is a little different from UTP's:
  - ➡ component type **C** and its implementation **I** are encoded in the **same** architectural model $M \mathrel{\widehat{=}} (\mathbf{C} \parallel \mathbf{I})$.
  - ➡ … rather than being separate computations.
- Besides, there may be more than on implementation of a single component type C: $M \mathrel{\widehat{=}} (\mathbf{C} \parallel \mathbf{I_1} \parallel \mathbf{I_2} \parallel \cdots)$.
- Hence, we trade the binary refinement relation:

  $C \sqsubseteq I$ for a UTP healthiness conditions $H_{\sqsubseteq}(M)$.
- Healthiness conditions form a **layered hierarchy** with successively stronger notions of refinement.
  - Structural / topological refinement at the top.
  - Behavioral refinement (**core** & **annexes**) below.

Current State of our Work

- So far, we have focused on mechanizing the structural (**declarative**) model of AADL in Isabelle/HOL as a baseline for further work.
- Includes core entities, such as Components, Properties, Features, Ports, Connections, Flows, Implementations.
- Formalization of **legality** and **consistency** rules.
- Emphasis on *traceability* and *hyperlinks* to an abridged version of the SAE AADL standard (version C).
- Supports code generation into Scala (JVM-based).
- Preliminary work on also generating the **instance** model from the **Ecore** meta-model description of OSATE2.

Current State of our Work

- So far, we have focused on mechanizing the structural (**declarative**) model of AADL in Isabelle/HOL as a baseline for further work.
- Includes co… , Properties, Features, P… …mentations.
- Formalizati… …rules.
- Emphasis c… …an abridged version of t… …C).
- Supports code generation into Scala (JVM-based).
- Preliminary work on also generating the **instance** model from the **Ecore** meta-model description of OSATE2.

The choice of Isabelle is motivated by using the Isabelle/UTP framework for UTP mechanization parts.

Isabelle/HOL Theory Extract

- AADL **component type** and **implementation** encoding:

```
subsection ‹Component Types›

record component_type =
  name       :: "classifier"
  category   :: "component_category"
  properties :: "property ⇀ property_value"
  features   :: "name ⇀ feature"

subsection ‹Implementations›

record implementation =
  name          :: "classifier"
  category      :: "component_category"
  subcomponents :: "name ⇀ classifier"
  properties    :: "property ⇀ property_value"
  connections   :: "name ⇀ connection"
```

```
datatype component_category =
    system
  | abstract
  | software "software_category"
  | hardware "hardware_category"
```

```
datatype software_category =
    process
  | thread
  | thread_group
  | subprogram
  | subprogram_group
  | data
```

Isabelle/HOL Theory Extract

- AADL **component type** and **implementation** encoding:

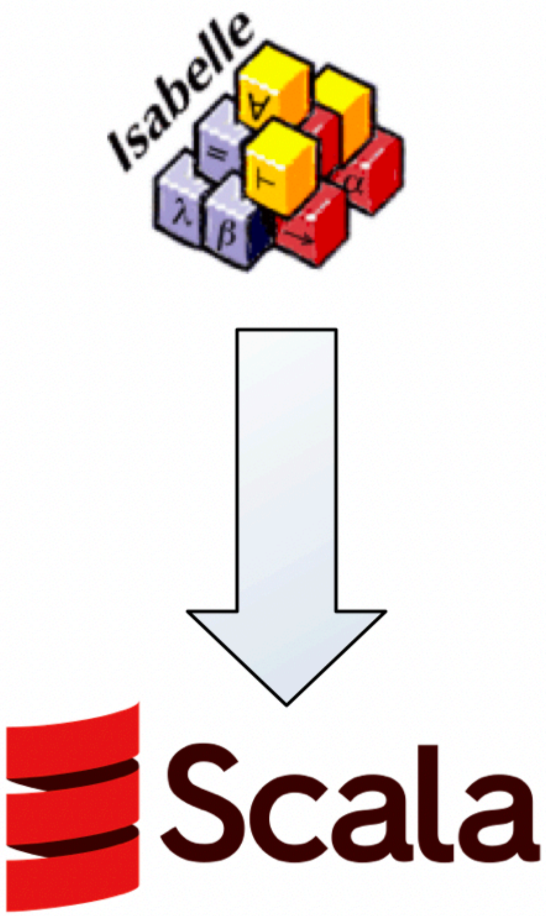
```
subsection ‹Component Types›

record component_type =
  name       :: "classifier"
  category   :: "component_category"
  properties :: "property ⇀ property_value"
  features   :: "name ⇀ feature"

subsection ‹Implementations›

record implementation =
  name          :: "classifier"
  category      :: "component_category"
  subcomponents :: "name ⇀ classifier"
  properties    :: "property ⇀ property_value"
  connections   :: "name ⇀ connection"
```

Records are by default extendible in Isabelle/HOL. We use the extension type to add behavioral models!

```
datatype software_category =
    process
  | thread
  | thread_group
  | subprogram
  | subprogram_group
  | data
```

Code Generation Example

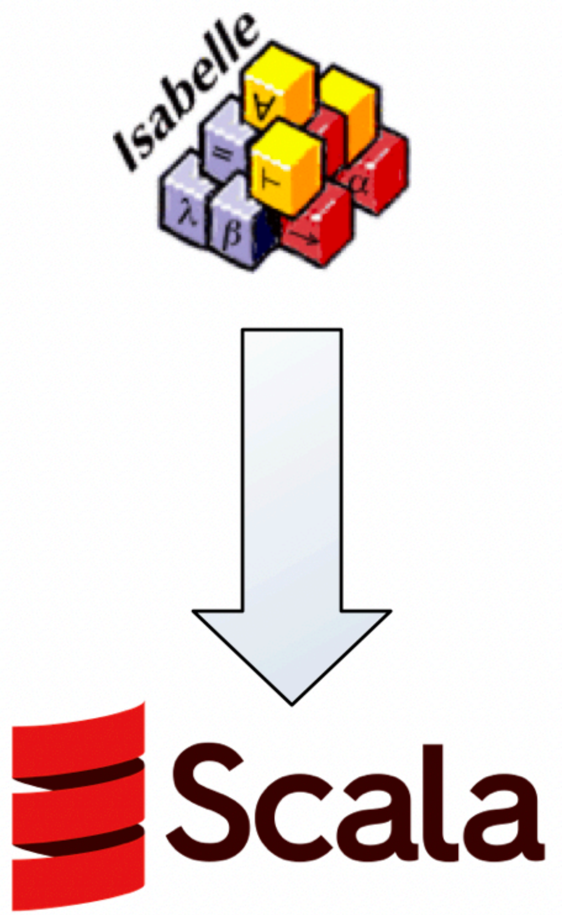
Example for record types component_type and implementation:




```
abstract sealed class component_type_ext[A]
final case class component_type_exta[A](
  a: classifier_ext[Unit],
  b: component_category,
  c: Map[(property_ext[Unit]), property_value],
  d: Map[String, (feature_ext[Unit])], e: A)
extends component_type_ext[A]

abstract sealed class implementation_ext[A]
final case class implementation_exta[A](
  a: classifier_ext[Unit],
  b: component_category,
  c: Map[String,  (classifier_ext[Unit])],
  d: Map[(property_ext[Unit]),  property_value],
  e: Map[String,  connection],
  f: A)
extends implementation_ext[A]
```

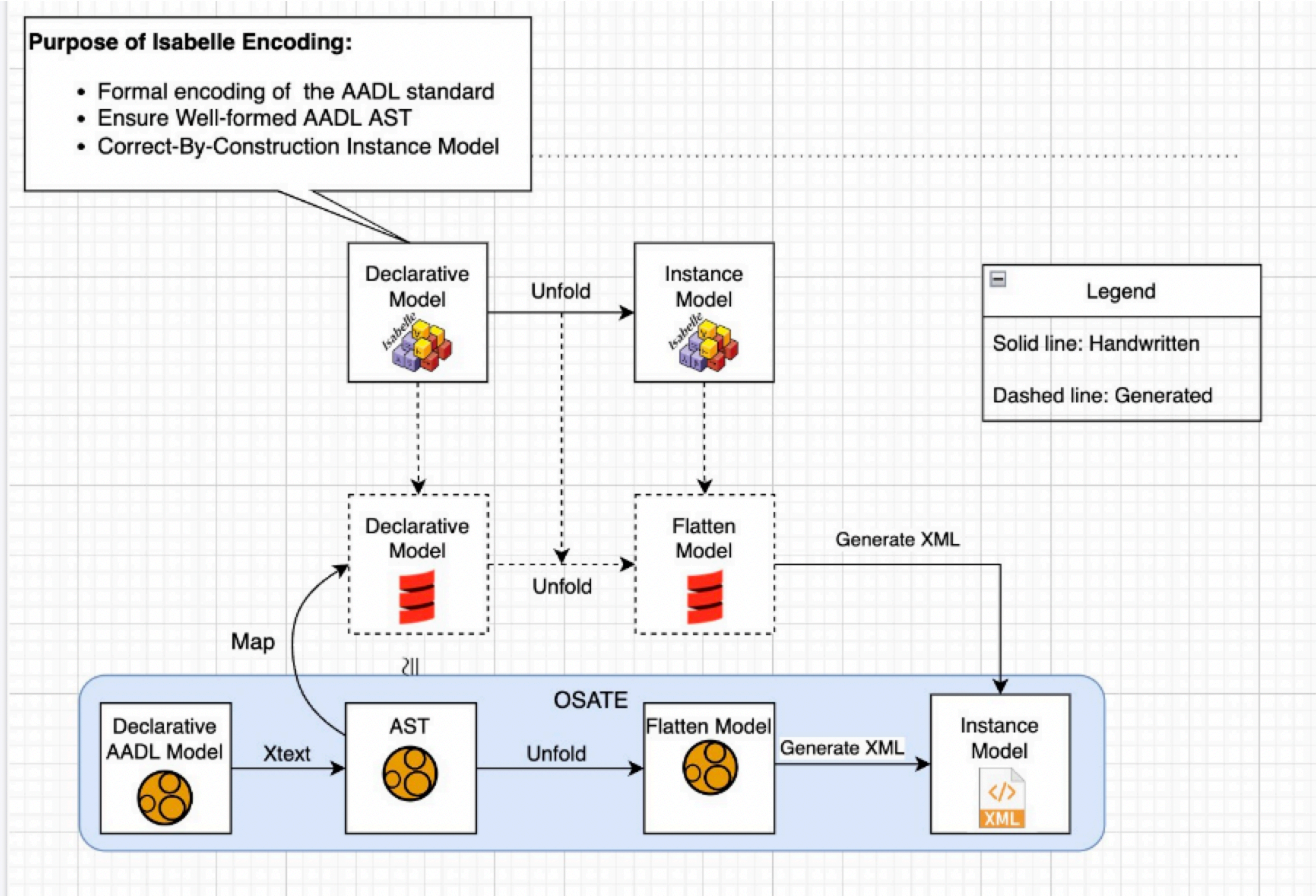
Code Generation Example (cont'd)

Translation of the earlier legality rule (wf_port):

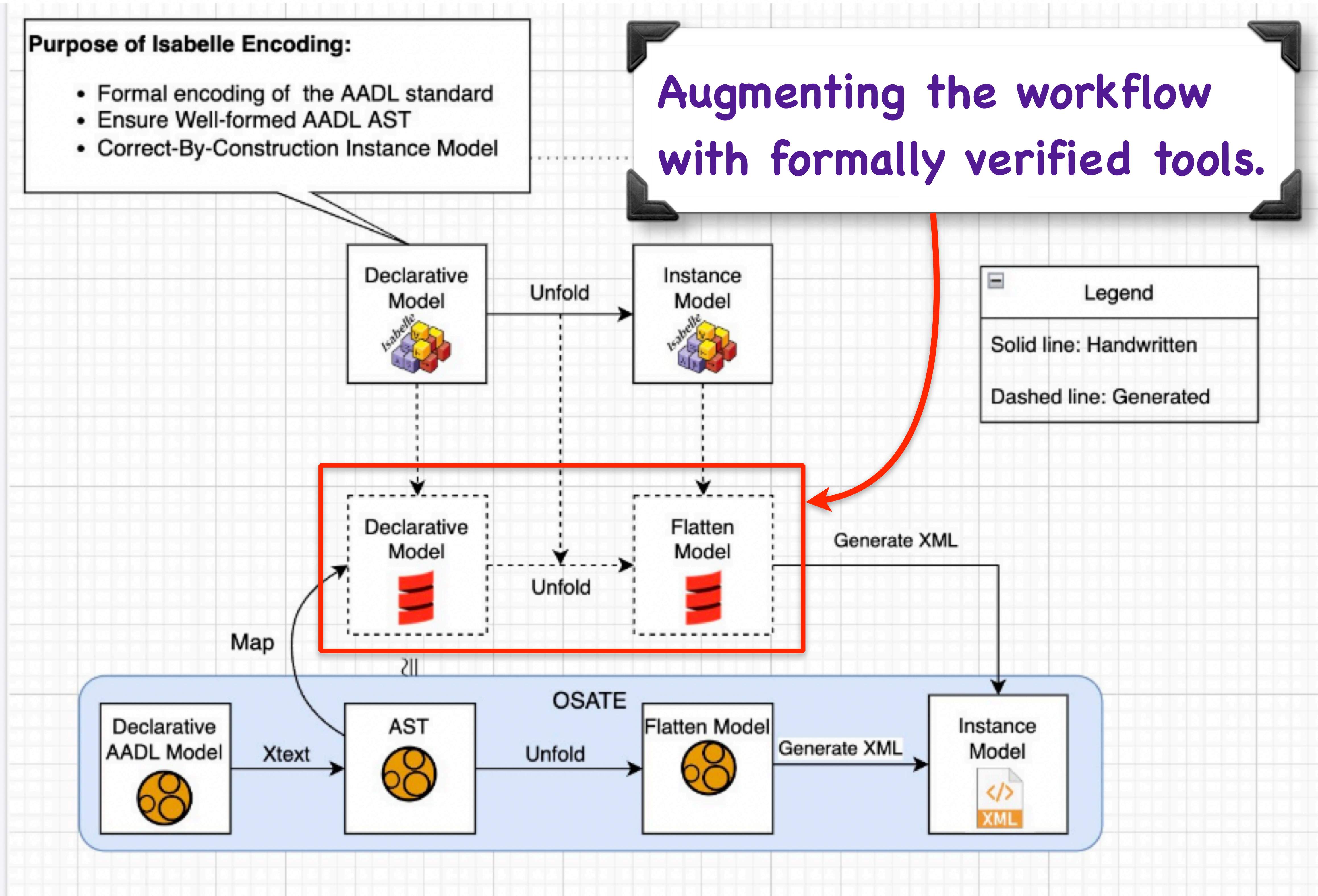



```
def wf_port(m: aadl_model_ext[Unit],
            c: component_type_ext[Unit],
            p: port_spec_ext[Unit]): Boolean =
  ((((((((Set.empty[component_category] +
    (hardware(device()))) +
    (hardware(virtual_processor()))) +
    (hardware(processor()))) +
    (software(thread_group()))) +
    (software(thread()))) +
    (software(process()))) +
    (abstracta())) +
    (system())) contains (category[Unit](c))
```

Use Case: Adding Formality to OSATE2



Use Case: Adding Formality to OSATE2

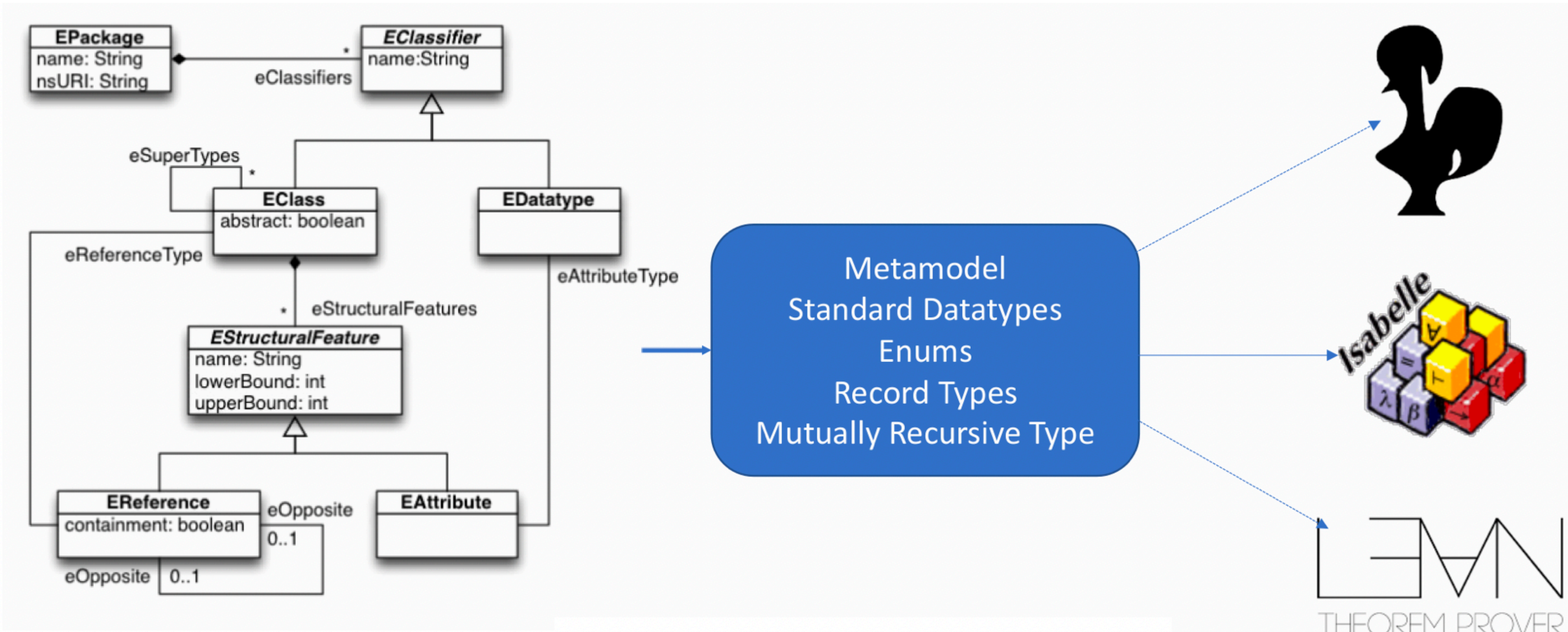


Caveats for Future Work

- Our mantra: "not to confine ourselves to a particular proof system, mechanization framework, or tool".
  - Considerable work that has already been done to formalize and mechanism the semantics of AADL in both Coq and Isabelle/HOL. (Jerome, KSU, etc.)
  - Fundamentally, both are suitable target platforms and fulfill the needs, and so is PVS, Lean, etc …
  - We opted for Isabelle/HOL solely since there already is an *elaborate mechanization* of UTP.
- Lean into **existing formalization** where a lot of work has already been accomplished (make meaningful additions rather than re-inventing the wheel …).

From Ecore to Formal Models

- Generation of a suitable **meta-model** to target different theorem provers:

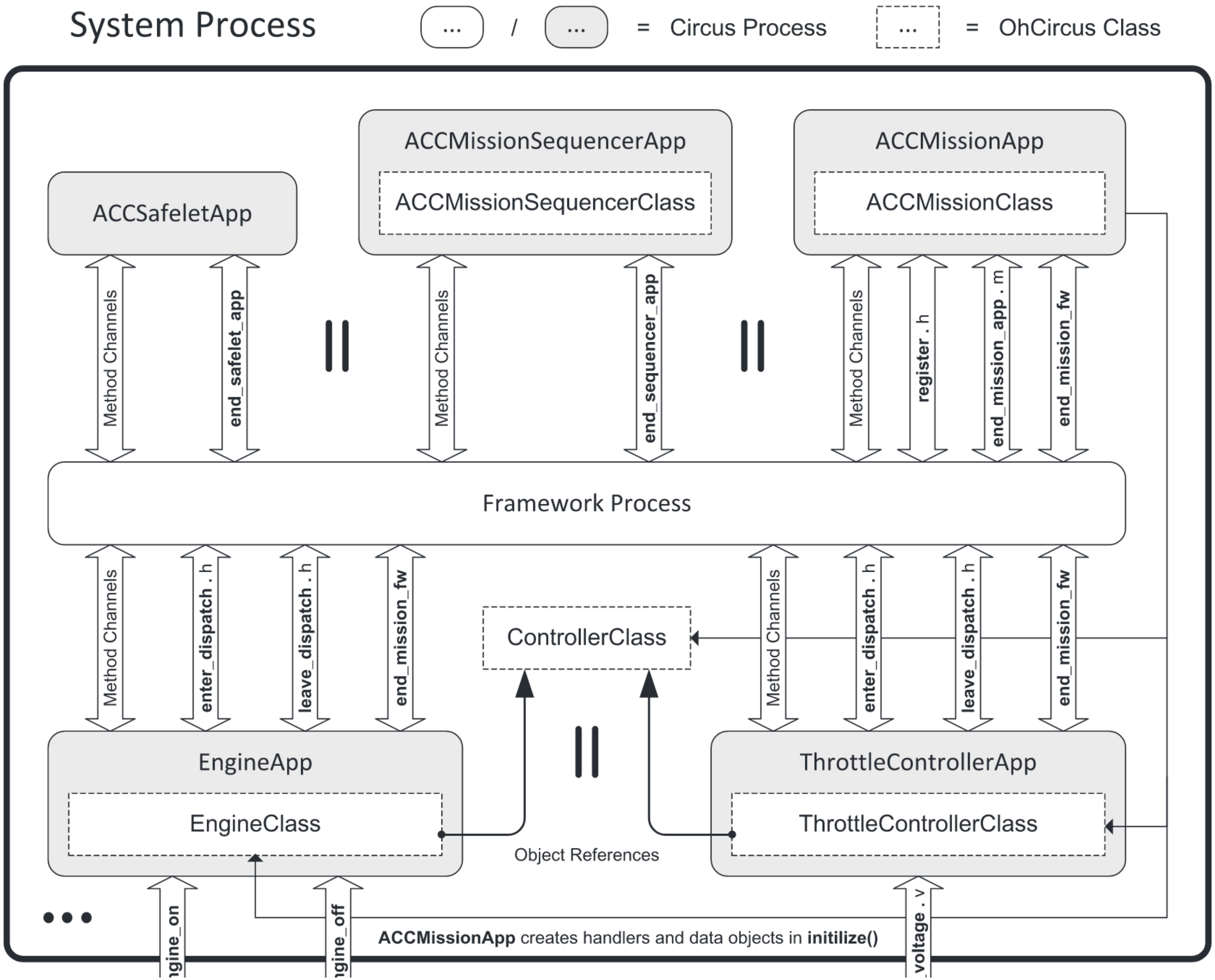


The meta-model is our starting point for model generation.

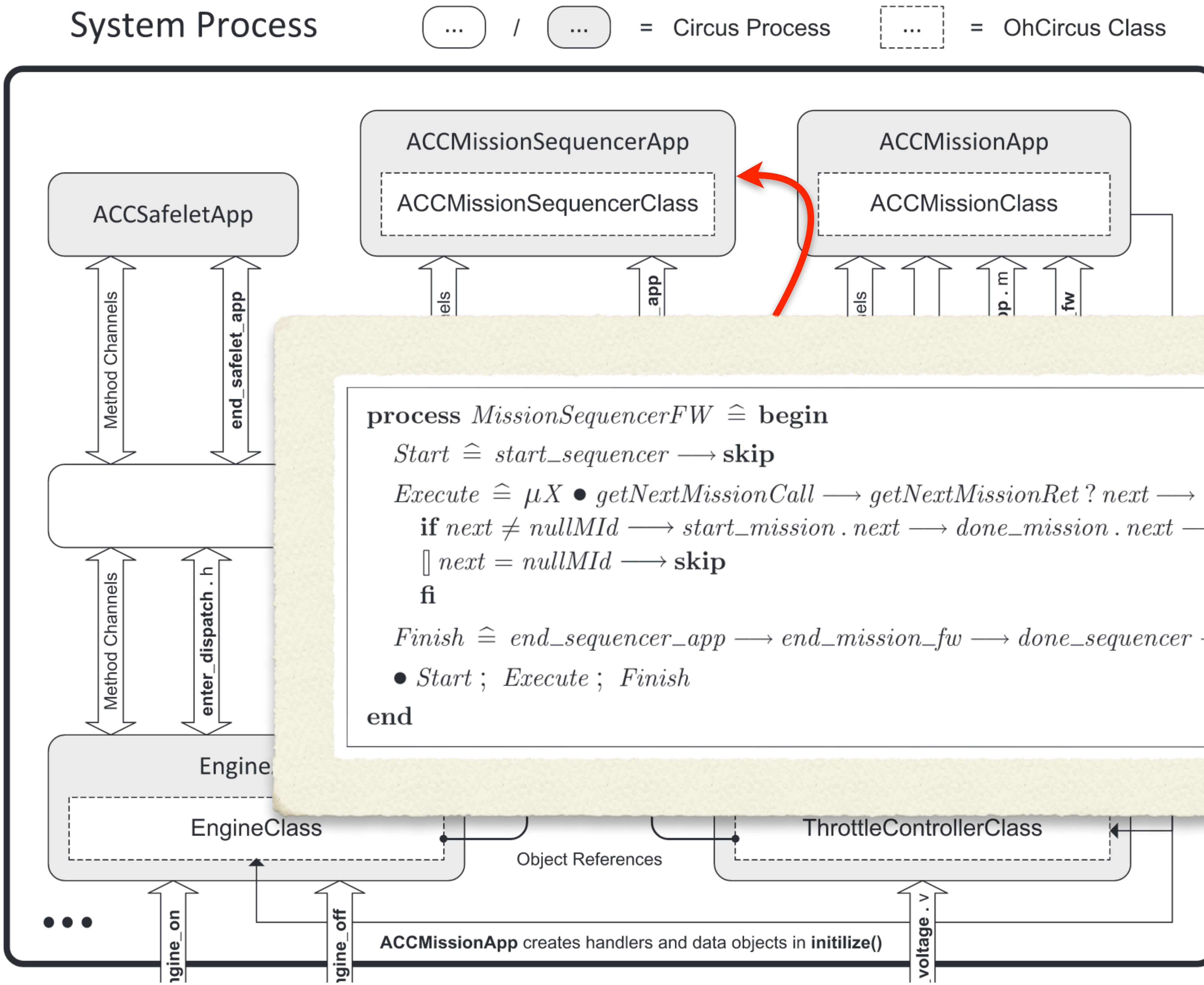
Conclusion

- We have sketched a vision here that still needs to be **validated** through implementation and examples…
- A first step will be to integrate a notion of **reactive computation** (as reactive design contracts and/or interaction trees) with the structural model.
- The incremental strengthening of refinement via **HC** poses some new challenges to proof engineering.
- Among other things, we aim to enable AADL **system engineering** and verification of **architectural patterns**, in addition to code-level verification (existing tools such as HAMR/Slang already do a brilliant job of that).

Addendum: Safety-Critical Java



Addendum: Safety-Critical Java



process $\mathit{MissionSequencerFW} \mathrel{\widehat=}$ **begin**

$\quad \mathit{Start} \mathrel{\widehat=} \mathit{start\_sequencer} \longrightarrow \mathbf{skip}$

$\quad \mathit{Execute} \mathrel{\widehat=} \mu X \bullet \mathit{getNextMissionCall} \longrightarrow \mathit{getNextMissionRet}\,?\,\mathit{next} \longrightarrow$

$\qquad \mathbf{if}\ \mathit{next} \neq \mathit{nullMId} \longrightarrow \mathit{start\_mission}\,.\,\mathit{next} \longrightarrow \mathit{done\_mission}\,.\,\mathit{next} \longrightarrow X$

$\qquad [\!]\ \mathit{next} = \mathit{nullMId} \longrightarrow \mathbf{skip}$

$\qquad \mathbf{fi}$

$\quad \mathit{Finish} \mathrel{\widehat=} \mathit{end\_sequencer\_app} \longrightarrow \mathit{end\_mission\_fw} \longrightarrow \mathit{done\_sequencer} \longrightarrow \mathbf{skip}$

$\quad \bullet\ \mathit{Start}\ ;\ \mathit{Execute}\ ;\ \mathit{Finish}$

end