Semantical Lifting: Programs, Digital Twins, Correctness

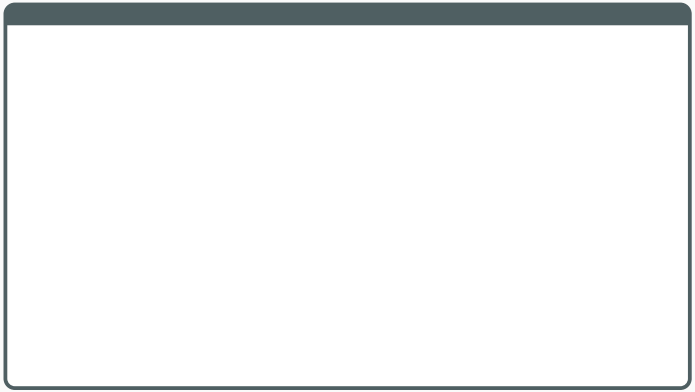
Eduard Kamburjan

WAKERS, Stellenbosch, 29.01.24

University of Oslo

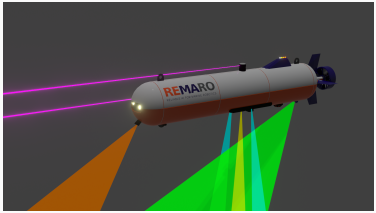
Knowledge Representation with Ontologies

Ontologies are logically formalized domain knowledge



Ontologies are logically formalized domain knowledge

- Intelligence for autonomous systems,
  e.g., for robotics

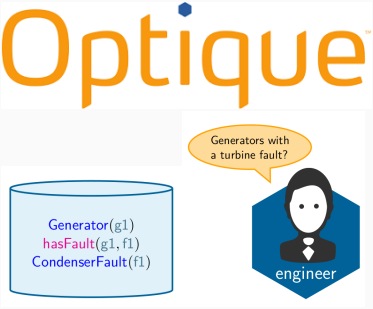


REMARO
RELIABLE AI FOR MARINE ROBOTICS

Knowledge Representation with Ontologies

Ontologies are logically formalized domain knowledge

- Intelligence for autonomous systems,
  e.g., for robotics
- Data access for domain experts
  e.g., in the energy industry



Ontologies are logically formalized domain knowledge

- Intelligence for autonomous systems,
  e.g., for robotics
- Data access for domain experts
  e.g., in the energy industry
- Reasoning for expert systems
  e.g., in the biomedical field

Ontologies are logically formalized domain knowledge

- Intelligence for autonomous systems,
  e.g., for robotics
- Data access for domain experts
  e.g., in the energy industry
- Reasoning for expert systems
  e.g., in the biomedical field
- Data integration
  e.g., as industrial standards

Knowledge Representation with Ontologies

Ontologies are logically formalized domain knowledge

- Intelligence for autonomous systems,
  e.g., for robotics
- Data access for domain experts
  e.g., in the energy industry
- Reasoning for expert systems
  e.g., in the biomedical field
- Data integration
  e.g., as industrial standards

IEEE SA
STANDARDS ASSOCIATION

READI

Data represented as *knowledge graphs*, tools summarized as semantic technologies

Problems for Programmers

How to use ontologies in programming?

- Make domain knowledge available to the programmer
- Reduce redundancy between program and other artifacts
- Simplify communication with users/domain experts

Problems for Programmers

How to use ontologies in programming?

- Make domain knowledge available to the programmer
- Reduce redundancy between program and other artifacts
- Simplify communication with users/domain experts

How to program applications around ontologies?

- Using multiple semantic technologies can be tricky
- Programmer must be aware of logical and formal pitfalls
- Correct interplay must be ensured manually

Tutorial Outline

- **Part I** Semantic Reflection in Programs

  How to add domain knowledge to imperative programming?

- **Part II** Self-Adaptation in Digital Twins

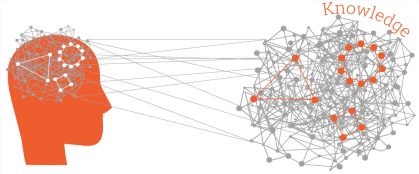
  How to use ontological asset models for structural self-adaptation?

- **Part III** Correctness and Verification

  How to ensure type safety and functional correctness?

Semantic Reflection in Programs

Knowledge Graphs



- Knowledge can be described ad hoc or in a structural manner
- Semantic Technologies/Knowledge Graph facilitate the description of structured knowledge, consistency checking and reasoning
- In this tutorial W3C standards:
  - For data: RDF (Resource description framework)
  - For knowledge: OWL (Web Ontology language)
  - For queries: SPARQL(an RDF query language)

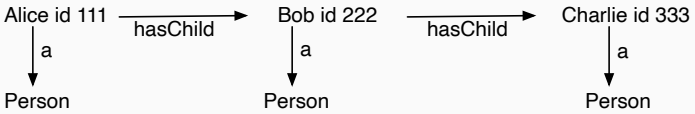
RDF (Resource description framework)

Data in **RDF** is expressed using a triple pattern, which consists of a *subject*, a *predicate*, and an *object*

RDF (Resource description framework)

Data in **RDF** is expressed using a triple pattern, which consists of a *subject*, a *predicate*, and an *object*

Example:



Turtle syntax: `Alice a Person.  Alice hasChild Bob.`

OWL (Web Ontology language)

OWL: Description Logic based language(s) to build ontologies, i.e., structured, general domain knowledge.

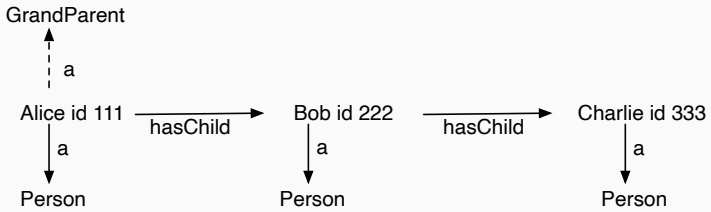
OWL (Web Ontology language)

OWL: Description Logic based language(s) to build ontologies, i.e., structured, general domain knowledge.

OWL (Web Ontology language)

OWL: Description Logic based language(s) to build ontologies, i.e., structured, general domain knowledge.

Example:

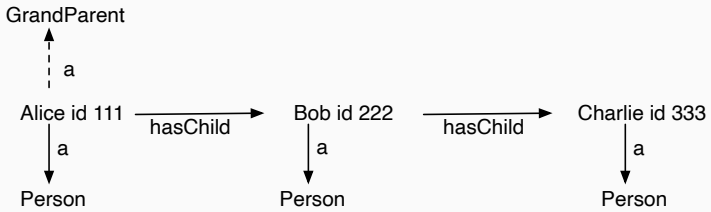


$\forall x \exists y \exists z \cdot hasChild(x, y) \wedge hasChild(y, z) \wedge Person(z) \implies GrandParent(x)$

OWL (Web Ontology language)

OWL: Description Logic based language(s) to build ontologies, i.e., structured, general domain knowledge.

Example:



$\forall x \exists y \exists z \cdot \mathit{hasChild}(x, y) \wedge \mathit{hasChild}(y, z) \wedge \mathit{Person}(z) \implies \mathit{GrandParent}(x)$

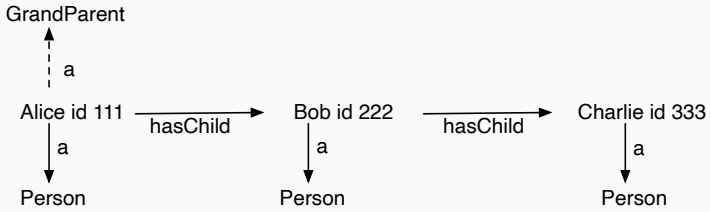
`hasChild` **some** (`hasChild` **some** `Person`) **subClassOf** `GrandParent`

SPARQL

SPARQL is an RDF query language:
a query language for databases stored in RDF format

SPARQL

SPARQL is an RDF query language:
a query language for databases stored in RDF format




```
SELECT ?x WHERE { ?x a :Person }

SELECT ?x ?y WHERE { ?x a :Person. ?x :hasChild ?y }

SELECT ?x WHERE { ?x a :GrandParent }
```

Assets as Knowledge Graphs

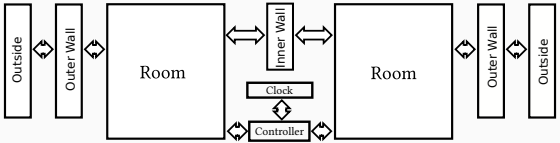
Asset Models

Asset models contain the current, past and designed structure of a facility, plus general knowledge for it. Aim: Use graph-based asset models to manage engineering lifecycle.

Assets as Knowledge Graphs

Asset Models

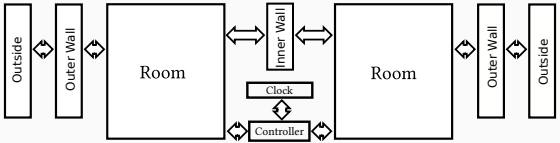
Asset models contain the current, past and designed structure of a facility, plus general knowledge for it. Aim: Use graph-based asset models to manage engineering lifecycle.



Assets as Knowledge Graphs

Asset Models

Asset models contain the current, past and designed structure of a facility, plus general knowledge for it. Aim: Use graph-based asset models to manage engineering lifecycle.

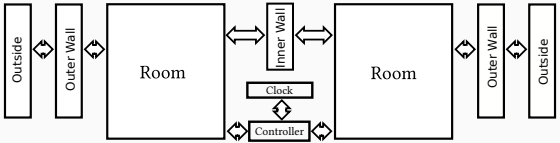



```
ast:heater1 a ast:Heater. ast:heater1 ast:in ast:room1.
ast:heater2 a ast:Heater. ast:heater2 ast:in ast:room2.
ast:heater1 ast:id 13. ast:heater2 ast:id 12.
ast:room1 ast:leftOf ast:room2.
```

Assets as Knowledge Graphs

Asset Models

Asset models contain the current, past and designed structure of a facility, plus general knowledge for it. Aim: Use graph-based asset models to manage engineering lifecycle.

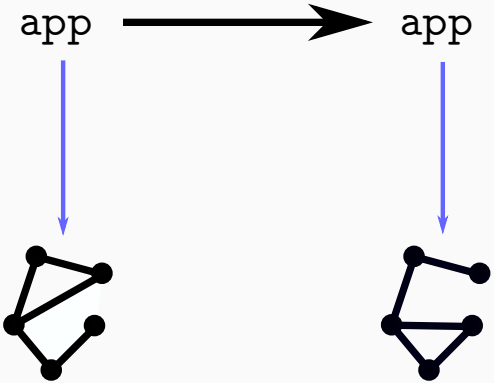



```
ast:heater1 a ast:Heater. ast:heater1 ast:in ast:room1.
ast:heater2 a ast:Heater. ast:heater2 ast:in ast:room2.
ast:heater1 ast:id 13. ast:heater2 ast:id 12.
ast:room1 ast:leftOf ast:room2.
```

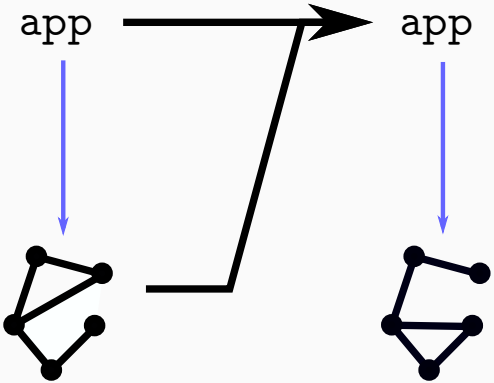
htLeftOf **subPropertyOf** `ast:in` **o** `ast:leftOf` **o inverse**(`ast:in`)

Semantically Lifted Programs

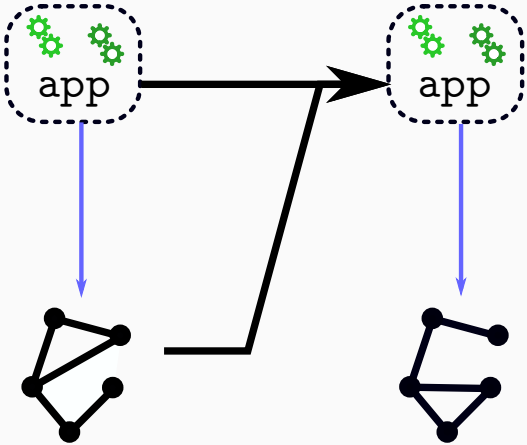
Semantically Lifted Programs



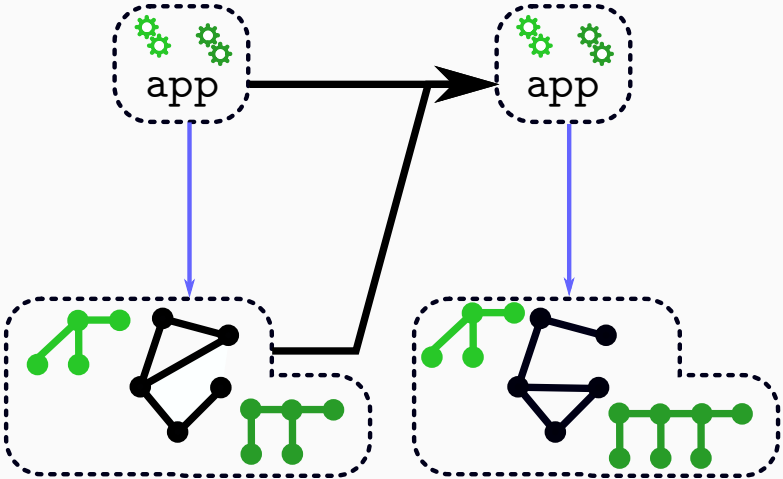
Semantically Lifted Programs



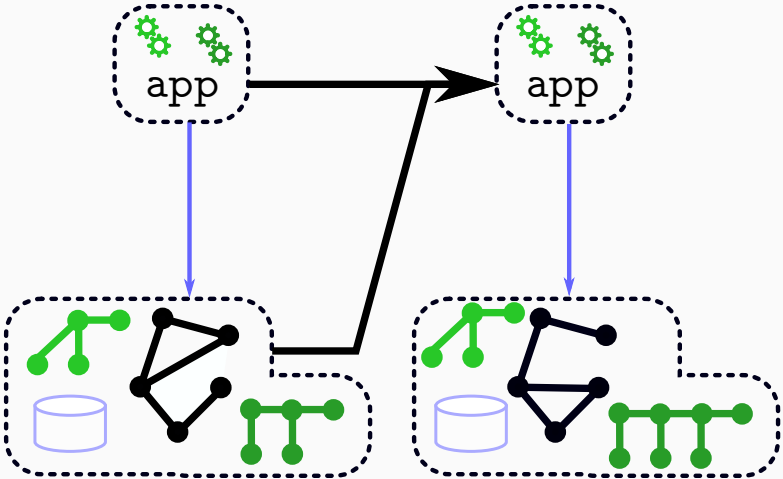
Semantically Lifted Programs



Semantically Lifted Programs



Semantically Lifted Programs



Direct Mapping of Program States

SMOL: Integration of Semantics and Semantic Technologies

Map each program state to a knowledge graph and allow program to operate on the KG. Implemented in SMOL.

```
1 class C (Int i) Unit inc(){ this.i = this.i + 1; } end
2 Main C c = new C(5);  Int i = c.inc(); end
```

Direct Mapping of Program States

SMOL: Integration of Semantics and Semantic Technologies

Map each program state to a knowledge graph and allow program to operate on the KG. Implemented in SMOL.

```
1 class C (Int i) Unit inc(){ this.i = this.i + 1; } end
2 Main C c = new C(5);  Int i = c.inc(); end
```

```
prog:C a prog:class. prog:C prog:hasField prog:i.
run:obj1 a prog:C.   run:obj1 prog:i 5.
run:proc1 a prog:process.
run:proc1 prog:runsOn run:obj1.
....
```

Semantic Reflection: Reasoning about oneself

```
1 class Building(List<Room> rooms) ... end
2 class Inspector(List<Building> buildings)
3  Unit inspectStreet(String street)
4   List<Building> l := access("SELECT ?x WHERE {?x a Villa. ?x :in %street}");
5   this.inspectAll(l);
6  end
7 end
```

Semantic Reflection: Reasoning about oneself

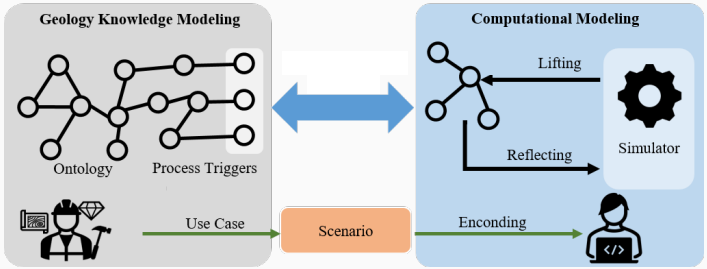
```
1 class Building(List<Room> rooms) ... end
2 class Inspector(List<Building> buildings)
3  Unit inspectStreet(String street)
4   List<Building> l := access("SELECT ?x WHERE {?x a Villa. ?x :in %street}");
5   this.inspectAll(l);
6  end
7 end
```

```
Villa EquivalentTo: rooms o length some xsd:int [>= 3]
```

Semantic Reflection: Reasoning about oneself – GeoSimulator

Case study of using SMOL for a geological simulator

- SMOL simulators describes the effects of the process
- SMOL state is interpreted through ontology
- Geological ontology describes under which conditions a geological process starts



[Qu, K., Torabi, and Giese, *Semantically triggered qualitative simulation of a geological process*, Appl. Comp. and Geosc., 2024]

Semantic Reflection: Reasoning about oneself – GeoSimulator

Modeling of a geological shale structure in SMOL

```
class ShaleUnit extends GeoUnit
 (Double temperature,
  Boolean hasKerogenSource,
  Int maturedUnits)
  models
  a GeoReservoirOntology_sedimentary_geological_object;
   location_of [a domain:amount_of_organic_matter];
   GeoCoreOntology_constituted_by [a domain:shale];
   has_quality [domain:datavalue %temperature; a domain:temperature].
end
```

Semantic Reflection: Reasoning about oneself – GeoSimulator

Resulting (part of the) knowledge graph

```
run:obj1 smol:models domain:obj1.
domain:obj1 a GeoReservoirOntology_sedimentary_geological_object;
   location_of [a domain:amount_of_organic_matter];
   GeoCoreOntology_constituted_by [a domain:shale];
   has_quality [domain:datavalue "10.0"^^xsd:Double; a domain:temperature].
```

Semantic Reflection: Reasoning about oneself – GeoSimulator

Simulation driver

```
1 List<ShaleUnit> fs =
2 member(smol:models some (participates_in some maturation_trigger))
3 while fs != null do
4   fs.content.mature(); fs = fs.next
5 end
```

For Mandal-Ekofisk field, simulation gives similar results as original study (2*mya* steps)

	SMOL	Cornford'94	Time Difference
Start M.	52*ma*	∼50*ma*	∼2*mya*
End M.	14*ma*	∼23*ma*	∼9*mya*
Crit. Moment	28*ma*	∼30*ma*	∼2*mya*

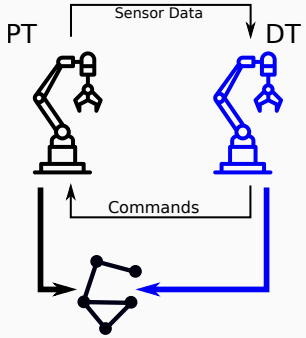
Self-Adaptation in Digital Twins

Structural Self-Adaptation

- We can access the sensors of the physical system,
- access the structure of the physical system, and
- simulate the digital design.

Structural Self-Adaptation

- We can access the sensors of the physical system,
- access the structure of the physical system, and
- simulate the digital design.

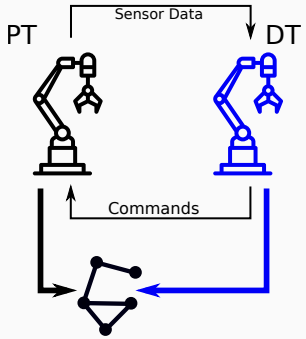


Structural Self-Adaptation

- We can access the sensors of the physical system,
- access the structure of the physical system, and
- simulate the digital design.

Putting it all together

- Compare simulations to sensors
- Compare digital with physical structure
- Self-adapt to changes in physical system

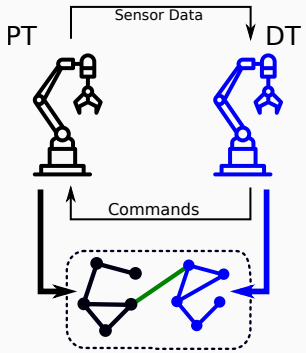


Structural Self-Adaptation

- We can access the sensors of the physical system,
- access the structure of the physical system, and
- simulate the digital design.

Putting it all together

- Compare simulations to sensors
- Compare digital with physical structure
  How to formalize consistency?
- Self-adapt to changes in physical system

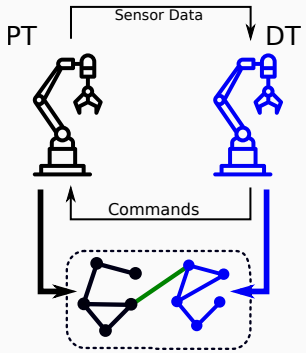


Structural Self-Adaptation

- We can access the sensors of the physical system,
- access the structure of the physical system, and
- simulate the digital design.

Putting it all together

- Compare simulations to sensors
- Compare digital with physical structure
  How to formalize consistency?
- Self-adapt to changes in physical system
  How to repair?



Self-Adaptation (I)

Digital Twins: Self-Adaptation

Self-adaptation means to *automatically* reestablish some property of a system, by reacting to outside stimuli. For Digital Twins, the "outside" is the physical system.

Two kinds of self-adaptation to reestablish the *twinning* property:

- Behavioral self-adaptation if sensors and simulators mismatch
- Structural self-adaptation if *structures* mismatch

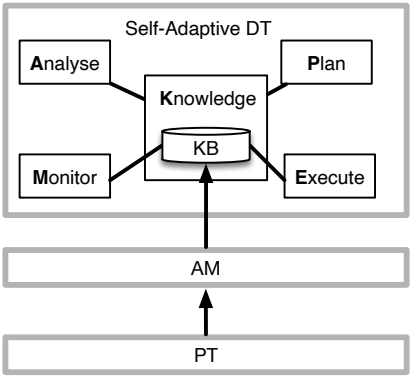
MAPE-K

MAPE-K is an established conceptual framework to structure self-adaptive systems.

MAPE-K

MAPE-K is an established conceptual framework to structure self-adaptive systems.

- A **K**nowledge component keeps track of information and goals for the self-adaptation loop:
- **M**onitor the situation
- **A**nalyze whether the situation requires adaptation
- **P**lan the adaptation
- **E**xecute the plan



Self-Adaptation (II)

Behavioral Self-Adaptation

Simulated (=expected) behavior of certain components does not match the real (=measured) behavior of the sensors.

- Monitor sensors
- Analyze the relation to simulation
- Plan repair by, e.g., finding new simulation parameters
- Exchange simulators or send signal to physical system

Reasons

- Sensor drift
- Modeling errors
- Faults
- Unexpected events

Self-Adaptation (III)

Structural Self-Adaptation

Simulated structure of digital system does not match real (= expressed in asset model) structure.

Self-Adaptation (III)

Structural Self-Adaptation

Simulated structure of digital system does not match real (= expressed in asset model) structure.

Semantically Lifted Programs

We need to express the program structure, so we can *uniformly* access it together with the asset model. How to apply semantic web technologies on programs? $\Rightarrow$ Semantical lifting.

Self-Adaptation (III)

Structural Self-Adaptation

Simulated (= lifted) structure of digital system does not match real (= expressed in asset model) structure.

Semantically Lifted Programs

We need to express the program structure, so we can *uniformly* access it together with the asset model. How to apply semantic web technologies on programs? $\Rightarrow$ Semantical lifting.

Semantical lifting is a mechanism to automatically generate the knowledge graph of a program state.

Self-Adaptation

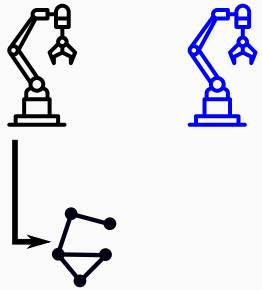
Repair

To self-adapt we must (1) detect broken twinning and (2) repair it.

Repair

To self-adapt we must (1) detect broken twinning and (2) repair it.

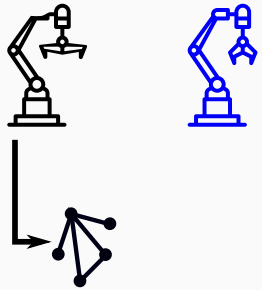
- Access PT structure through asset model



Repair

To self-adapt we must (1) detect broken twinning and (2) repair it.

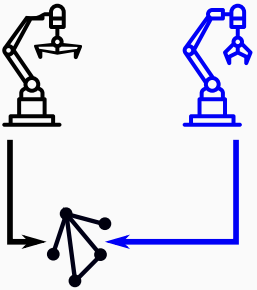
- Access PT structure through asset model
- Changes of PT are visible in asset model



Repair

To self-adapt we must (1) detect broken twinning and (2) repair it.

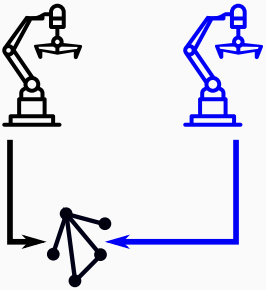
- Access PT structure through asset model
- Changes of PT are visible in asset model
- Asset model accessible directly to DT



Repair

To self-adapt we must (1) detect broken twinning and (2) repair it.

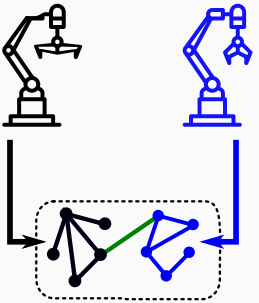
- Access PT structure through asset model
- Changes of PT are visible in asset model
- Asset model accessible directly to DT



Repair

To self-adapt we must (1) detect broken twinning and (2) repair it.

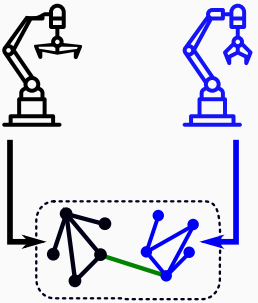
- Access PT structure through asset model
- Changes of PT are visible in asset model
- Asset model accessible directly to DT
- Detect changes through combined knowledge graph



Repair

To self-adapt we must (1) detect broken twinning and (2) repair it.

- Access PT structure through asset model
- Changes of PT are visible in asset model
- Asset model accessible directly to DT
- Detect changes through combined knowledge graph

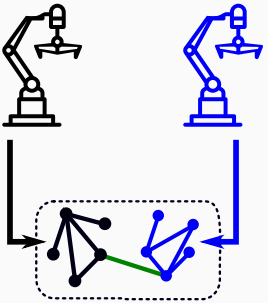


Self-Adaptation

Repair

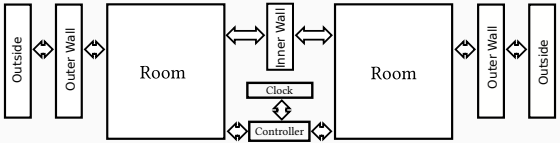
To self-adapt we must (1) detect broken twinning and (2) repair it.

- Access PT structure through asset model
- Changes of PT are visible in asset model
- Asset model accessible directly to DT
- Detect changes through combined knowledge graph
- Information for repair available there!



Example

Back to digital twins



- Monitor consistency
- Monitor twinning
- Adapt to addition of new rooms

Model Description

```
<fmiModelDescription fmiVersion="2.0" modelName="Example" ...>
  <CoSimulation needsExecutionTool="true" .../>
  <ModelVariables>
    <ScalarVariable name="p" variability="continuous"
                    causality ="parameter">
      <Real start="0.0"/>
    </ScalarVariable>
    <ScalarVariable name="input" variability="continuous"
                    causality ="input">
      <Real start="0.0"/>
    </ScalarVariable>
    <ScalarVariable name="val" variability="continuous"
                    causality ="output" initial ="calculated">
      <Real/>
  </ModelVariables>
  <ModelStructure> ... </ModelStructure>
</fmiModelDescription>
```

SMOL and FMI

Functional Mock-Up Objects (FMOs)

Tight integration of simulation units using FMI into programs.

```
1 //setup
2 FMO[out Double val] shadow =
3     simulate("Sim.fmu", input=sys.val, p=1.0);
4 FMO[out Double val] sys = simulate("Realsys.fmu");
5 Monitor m = new Monitor(sys,shadow); m.run(1.0);
```

SMOL and FMI

Functional Mock-Up Objects (FMOs)

Tight integration of simulation units using FMI into programs.

```
1 //setup
2 FMO[out Double val] shadow =
3     simulate("Sim.fmu", input=sys.val, p=1.0);
4 FMO[out Double val] sys = simulate("Realsys.fmu");
5 Monitor m = new Monitor(sys,shadow); m.run(1.0);
```

Integration

- Type of FMO directly checked against model description
- Variables become fields, functions become methods
- Causality reflected in type

SMOL and FMI

Functional Mock-Up Interface (FMI)

Standard for (co-)simulation units, called function mock-up units (FMUs). Can also serve as interface to sensors and actuators.

SMOL and FMI

Functional Mock-Up Interface (FMI)

Standard for (co-)simulation units, called function mock-up units (FMUs). Can also serve as interface to sensors and actuators.

```
1 //simplified shadow
2 class Monitor(FMO[out Double val] sys,
3               FMO[out Double val] shadow)
4   Unit run(Double threshold)
5     while shadow != null do
6       sys.doStep(1.0);  shadow.doStep(1.0);
7       if(sys.val - shadow.val >= threshold) then ... end
8     end ...
```

Is this twinning something? Is this setup correctly?

SMOL and FMI

SMOL with FMOs

FMOs are objects, so they are part of the knowledge graph.

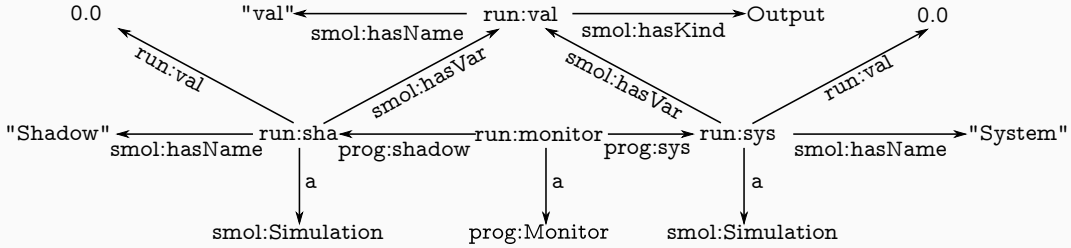
```
1 class Monitor(FMO[out Double val] sys,
2               FMO[out Double val] shadow)
```

SMOL and FMI

SMOL with FMOs

FMOs are objects, so they are part of the knowledge graph.

```
1 class Monitor(FMO[out Double val] sys,
2               FMO[out Double val] shadow)
```



Using the Semantical Lifting

SPARQL

Define structural requirements as queries in SPARQL on *combined* knowledge graph, to use domain constraints on digital twin.

Query to detect non-sensical setups:

```
SELECT ?room WHERE { ?ctrl a prog:Controller.
                     ?ctrl prog:Controller_left ?room.
                     ?ctrl prog:Controller_right ?room }
```

Using the Semantical Lifting

SPARQL

Define structural requirements as queries in SPARQL on *combined* knowledge graph, to use domain constraints on digital twin.

Query to check structural consistency for heaters:

```
SELECT * WHERE { ?o1 prog:Room_id ?id1. ?h1 asset:id ?id1.
                 ?o2 prog:Room_id ?id2. ?h2 asset:id ?id2.
                 ?h1 htLeftOf ?h2.
                 ?c a prog:Controller.
                 ?c prog:Controller_left ?o1.
                 ?c prog:Controller_right ?o2}
```

Demo

Inconsistent Twinning

Self-Adapting to Structural Drift

Detecting Structural Drift

The previous query can detect that some mismatch between asset model and program state exists.

How to detect where the mismatch is and how to repair it?

- Retrieve all assets, and their connections by id (**M**)
- Remove all ids present in the digital twin
- If any id is left, assets needs to be twinned (**A**)
- Find kind of defect to plan repair (**P**)
- Execute repair according to connections (**E**)
- Monitor connections using previous query
- (And v.v. to detect twins that must be removed)

Example: Adding a New Room

- Get all (asset) rooms and their neighboring walls
- Remove all (twinned) rooms with the same id
- Use the information about walls to
- Assumption: at least one new room is next to an existing one

```
1 class RoomAsrt(String room, String wallLt, String wallRt) end
2 ....
3 List<RoomAsrt> newRooms =
4  construct(" SELECT ?room ?wallLt ?wallRt WHERE
5   { ?x a asset:Room;
6        asset:right [asset:Wall_id ?wallRt];
7        asset:left [asset:Wall_id ?wallLt]; asset:Room_id ?room.
8     FILTER NOT EXISTS {?y a prog:Room; prog:Room_id ?room.} }");
```

Demo

`Repair`

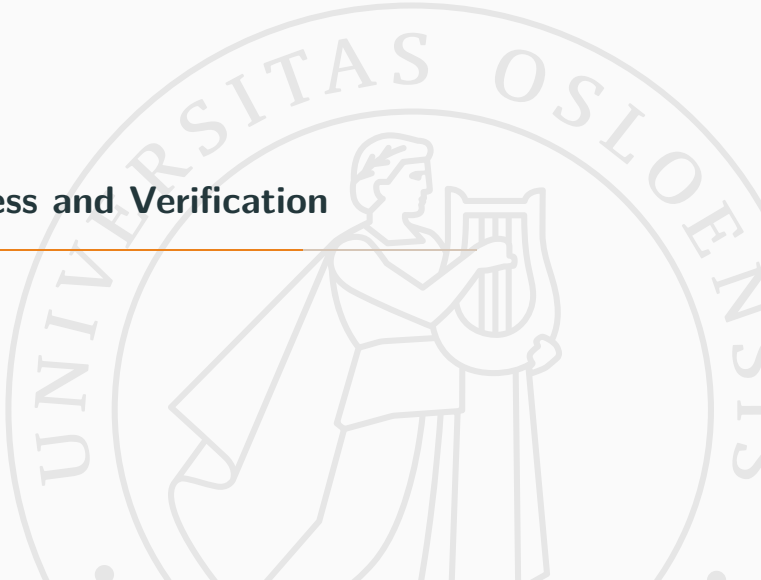
Remarks

Assumptions

- We know all the possible modifications up-front
  E.g., how to deal with a heater getting new features?
- We know how to always correct structural drift
- Changes do not happen faster than we can repair

Monitoring is still needed to (a) ensure that repairs work correctly, and (b) detect loss of twinning due to, e.g., unexpected structural drift.

Correctness and Verification



Type Safety and Interfaces

```
1 class Building(List<Room> rooms) ... end
2 class Inspector(List<Building> buildings)
3  Unit inspectStreet(String street)
4   List<Building> l := access("SELECT ?x WHERE {?x a Villa. ?x :in %street}");
5   this.inspectAll(l);
6  end
7 end
```

```
Villa EquivalentTo: rooms o length some xsd:int [>= 3]
```

Is this type safe?

Type Safety and Interfaces

```
1 class Building(List<Room> rooms) ... end
2 class Inspector(List<Building> buildings)
3  Unit inspectStreet(String street)
4   List<Building> l := access("SELECT ?x WHERE {?x a Villa. ?x :in %street}");
5   this.inspectAll(l);
6  end
7 end
```

```
Villa EquivalentTo: rooms o length some xsd:int [>= 3]
```

Is this type safe?

- Depends on the ontology – it is safe if every villa is a building
- Requires reasoning, e.g., about the domain of `rooms`

Type Safety for Semantic Reflection

Types & subject reduction

- SMOL is statically typed, . . .

Type Safety for Semantic Reflection

Types & subject reduction

- SMOL is statically typed, ... even with an untyped query language

Type Safety for Semantic Reflection

Types & subject reduction

- SMOL is statically typed, ... even with an untyped query language
- We can guarantee safe query access if ontology $\mathcal{K}$ is known

Type Safety for Semantic Reflection

Types & subject reduction

- SMOL is statically typed, ... even with an untyped query language
- We can guarantee safe query access if ontology $\mathcal{K}$ is known

$$\frac{\mathit{answers}(\texttt{Q}) \subseteq \mathit{members}(\texttt{C})}{\Gamma \vdash \texttt{List<C> l:=access(Q); : Unit}}$$

Type Safety for Semantic Reflection

Types & subject reduction

- SMOL is statically typed, ... even with an untyped query language
- We can guarantee safe query access if ontology $\mathcal{K}$ is known

$$\frac{\texttt{Q} \subseteq \{\texttt{?x a prog:C.}\}}{\Gamma \vdash \texttt{List<C> l:=access(Q); : Unit}}$$

Queries

- Query containment (wrt. entailment) becomes our subtyping relation
- (More) tractable if query translates into DL concept

Type Safety for Semantic Reflection

Types & subject reduction

- SMOL is statically typed, ... even with an untyped query language
- We can guarantee safe query access if ontology $\mathcal{K}$ is known

$$\frac{\texttt{Q} \subseteq \{\texttt{?x a prog:C.}\}}{\Gamma \vdash \texttt{List<C> l:=access(Q); : Unit}}$$

$$\frac{\mathcal{K} \vdash \Phi(\texttt{Q}) \sqsubseteq \texttt{prog:C}}{\Gamma \vdash \texttt{List<C> l:=access(Q); : Unit}}$$

Queries

- Query containment (wrt. entailment) becomes our subtyping relation
- (More) tractable if query translates into DL concept

Type Safety for Semantic Reflection

Types & subject reduction

- SMOL is statically typed, ... even with an untyped query language
- We can guarantee safe query access if ontology $\mathcal{K}$ is known

$$\frac{\texttt{Q} \subseteq \{\texttt{?x a prog:C.}\}}{\Gamma \vdash \texttt{List<C> l:=access(Q); : Unit}}$$

$$\frac{\mathcal{K} \vdash \Phi(\texttt{Q}) \sqsubseteq \texttt{prog:C}}{\Gamma \vdash \texttt{List<C> l:=access(Q); : Unit}}$$

$$\frac{\exists C.\ \exists \bar{y}.(\phi) \sqsubseteq^{\mathcal{K}} C \sqsubseteq^{\mathcal{K}} \mathit{Class}_{\mathtt{T}'} \quad \Gamma \vdash \texttt{l} : \texttt{List<T'>} \quad \Gamma \vdash \texttt{e}_i : \texttt{T}_i}{\Gamma \vdash^{\mathcal{K}}_{\mathsf{er}} \texttt{l:=access}(\exists \bar{y}.\,\phi, \texttt{e}_1, \ldots, \texttt{e}_n) \ : \texttt{Unit}}$$

Queries

- Query containment (wrt. entailment) becomes our subtyping relation
- (More) tractable if query translates into DL concept

Functional Correctness

So far, we can

- integrate knowledge intro control flow at runtime,
- use combined knowledge graph to check for consistency at runtime, and
- ensure statically that runtime queries results are representable.

Functional Correctness

So far, we can

- integrate knowledge intro control flow at runtime,
- use combined knowledge graph to check for consistency at runtime, and
- ensure statically that runtime queries results are representable.

Can we use ontologies also for specification of behavior and static verification?

What is a Car?

Suppose you model the assembly process of a car

```
1 procedure addWheels(p) nrWheels := p end
```

What is a Car?

Suppose you model the assembly process of a car

```
1 procedure addWheels(p) nrWheels := p end
```

Programmer

This procedure sets the number of wheels in a car to the value of p.

$\{\top\}\mathit{addWheels}(p)\{\texttt{nrWheels} \doteq \texttt{p}\}$

Subject Matter Expert

I want that in the end of this step, the car has 4 wheels.

$\{\top\}\mathit{addWheels}(p)\{\texttt{HasFourWheels}(c)\}$

What is a Car?

Suppose you model the assembly process of a car

```
1 procedure addWheels(p) nrWheels := p end
```

Programmer

This procedure sets the number of wheels in a car to the value of `p`.

$\{\top\}\mathit{addWheels}(p)\{\texttt{nrWheels} \doteq \texttt{p}\}$

Subject Matter Expert

I want that in the end of this step, the car has 4 wheels.

$\{\top\}\mathit{addWheels}(p)\{\texttt{HasFourWheels}(c)\}$

How to enable both of them to specify that?

- SME does not know about how the car c is encoded
- Programmer does not know what it means for a car to be small.

Lifted Specification

$$\left\{\begin{array}{c} - \\ \mathtt{p} \doteq 4 \end{array}\right\} \mathtt{addWheels(p)} \left\{\begin{array}{c} \mathtt{HasFourWheels}(c) \\ - \end{array}\right\}$$

- Upper component specifies lifted state
- Lower component specifies non-lifted state

Lifted Specification

$$\left\{\begin{array}{c} - \\ \mathtt{p} \doteq 4 \end{array}\right\} \mathtt{addWheels(p)} \left\{\begin{array}{c} \mathtt{HasFourWheels}(c) \\ - \end{array}\right\}$$

- Upper component specifies lifted state
- Lower component specifies non-lifted state

$$\mathtt{HasBody} \sqsubseteq \mathtt{HasChassis} \sqsubseteq \mathtt{Car} \qquad \exists \mathtt{doors}.\exists \mathtt{hasValue}.2 \equiv \mathtt{HasTwoDoors}$$

$$\exists \mathtt{wheels}.\exists \mathtt{hasValue}.4 \equiv \mathtt{HasFourWheels}$$

Lifted Specification

$$\left\{\begin{array}{c} - \\ \mathtt{p} \doteq 4 \end{array}\right\} \mathtt{addWheels(p)} \left\{\begin{array}{c} \mathtt{HasFourWheels}(c) \\ - \end{array}\right\}$$

- Upper component specifies lifted state
- Lower component specifies non-lifted state

$$\mathtt{HasBody} \sqsubseteq \mathtt{HasChassis} \sqsubseteq \mathtt{Car} \qquad \exists \mathtt{doors}.\exists \mathtt{hasValue}.2 \equiv \mathtt{HasTwoDoors}$$

$$\exists \mathtt{wheels}.\exists \mathtt{hasValue}.4 \equiv \mathtt{HasFourWheels}$$

We can lift the state at runtime, but at compile-time we need to lift the specification...

Specification

State Lifting

Function μ from runtime states to knowledge graphs.

Specification Lifting

Function $\widehat{\mu}$ from program assertions to axioms. Must be compatible to state lifting:

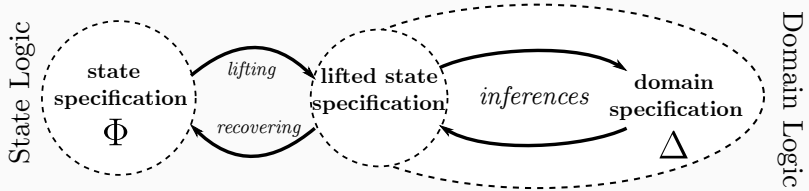
$$\sigma \models \phi \rightarrow \mu(\sigma) \models \widehat{\mu}(\phi)$$

This is not enough for the specification – the lifted specification cannot be used for domain specification

$$\widehat{\mu}(\texttt{nrWheels} \doteq 4) = \{\texttt{hasValue}(\texttt{nrWheels}, 4), \quad \texttt{wheels}(c, \texttt{nrWheels})\}$$

But we need to derive the state assertions from `HasFourWheels`

A Signature Perspective



Kernel and Generator

Let Σ be the signature of the domain specification.

- The kernel of $\widehat{\mu}$ is a signature **ker** $\widehat{\mu} \subseteq \Sigma$.
- A core generator α maps axioms Δ to axioms $\alpha(\Delta)$ with $\alpha(\Delta) \models \Delta$

- Kernel generator can either implement deduction, or abduction
- In case of abduction: ABox abduction with signature abducibles

[K. and Gurov, *A Hoare Logic for Domain Specification*, 2024]

Rules (I)

- First you generate the kernel
- Additional premise trivial if α is deductive

$$\text{pre-core}\ \frac{\Delta_2 \models^{\mathsf{K}} \alpha(\Delta_2) \qquad \mathbf{K} \vdash \left\{\begin{smallmatrix}\Delta_1\\ \phi_1\end{smallmatrix}\right\} s \left\{\begin{smallmatrix}\Delta_2, \alpha(\Delta_2)\\ \phi_2\end{smallmatrix}\right\}}{\mathbf{K} \vdash \left\{\begin{smallmatrix}\Delta_1\\ \phi_1\end{smallmatrix}\right\} s \left\{\begin{smallmatrix}\Delta_2\\ \phi_2\end{smallmatrix}\right\}}$$

Rules (I)

- First you generate the kernel
- Additional premise trivial if α is deductive

$$\text{pre-core}\ \frac{\Delta_2 \models^{\mathsf{K}} \alpha(\Delta_2) \qquad \mathbf{K} \vdash \left\{\begin{smallmatrix}\Delta_1\\ \Phi_1\end{smallmatrix}\right\} s \left\{\begin{smallmatrix}\Delta_2, \alpha(\Delta_2)\\ \Phi_2\end{smallmatrix}\right\}}{\mathbf{K} \vdash \left\{\begin{smallmatrix}\Delta_1\\ \Phi_1\end{smallmatrix}\right\} s \left\{\begin{smallmatrix}\Delta_2\\ \Phi_2\end{smallmatrix}\right\}}$$

- Second you generate state assertions from the kernel axioms

$$\text{post-inv}\ \frac{\mathbf{K} \vdash \left\{\begin{smallmatrix}\Delta_1\\ \Phi_1\end{smallmatrix}\right\} s \left\{\begin{smallmatrix}\Delta, \Delta_2\\ \Phi_2 \wedge \widehat{\mu}^{-1}(\Delta_2)\end{smallmatrix}\right\}}{\mathbf{K} \vdash \left\{\begin{smallmatrix}\Delta_1\\ \Phi_1\end{smallmatrix}\right\} s \left\{\begin{smallmatrix}\Delta, \Delta_2\\ \Phi_2\end{smallmatrix}\right\}}\ \mathsf{sig}(\Delta_2) \subseteq \mathbf{ker}\ \widehat{\mu}$$

Rules (I)

- First you generate the kernel
- Additional premise trivial if α is deductive

$$\text{pre-core}\ \frac{\Delta_2 \models^{\mathsf{K}} \alpha(\Delta_2) \qquad \mathbf{K} \vdash \left\{\begin{smallmatrix}\Delta_1\\ \Phi_1\end{smallmatrix}\right\} s \left\{\begin{smallmatrix}\Delta_2, \alpha(\Delta_2)\\ \Phi_2\end{smallmatrix}\right\}}{\mathbf{K} \vdash \left\{\begin{smallmatrix}\Delta_1\\ \Phi_1\end{smallmatrix}\right\} s \left\{\begin{smallmatrix}\Delta_2\\ \Phi_2\end{smallmatrix}\right\}}$$

- Second you generate state assertions from the kernel axioms

$$\text{post-inv}\ \frac{\mathbf{K} \vdash \left\{\begin{smallmatrix}\Delta_1\\ \Phi_1\end{smallmatrix}\right\} s \left\{\begin{smallmatrix}\Delta, \Delta_2\\ \Phi_2 \wedge \widehat{\mu}^{-1}(\Delta_2)\end{smallmatrix}\right\}}{\mathbf{K} \vdash \left\{\begin{smallmatrix}\Delta_1\\ \Phi_1\end{smallmatrix}\right\} s \left\{\begin{smallmatrix}\Delta, \Delta_2\\ \Phi_2\end{smallmatrix}\right\}} \ \mathsf{sig}(\Delta_2) \subseteq \mathbf{ker}\ \widehat{\mu}$$

- Same for precondition
- On state assertions, we can now use standard Hoare rules

A Car is a `Car`

- Standard Hoare calculus rules must check that specifications are consistent, and
- remove all domain knowledge, as it may have changed

$$\text{var}\ \frac{\widehat{\mu}(\Phi) \models^{\mathsf{K}} \Delta}{\mathsf{K} \vdash \left\{ {}^{\emptyset}_{\Phi[\mathtt{v} \setminus \mathtt{expr}]} \right\} \mathtt{v} \ := \ \mathtt{expr} \left\{ {}^{\Delta}_{\Phi} \right\}}$$

$$\text{skip}\ \frac{}{\mathsf{K} \vdash \left\{ {}^{\Delta}_{\Phi} \right\} \mathsf{skip} \left\{ {}^{\Delta}_{\Phi} \right\}}$$

A Car is a `Car`

- Standard Hoare calculus rules must check that specifications are consistent, and
- remove all domain knowledge, as it may have changed

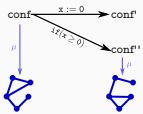
$$\text{var}\ \frac{\widehat{\mu}(\Phi) \models^{\mathsf{K}} \Delta}{\mathbf{K} \vdash \{^{\emptyset}_{\Phi[\mathtt{v}\setminus\mathtt{expr}]}\}\mathtt{v} := \mathtt{expr}\{^{\Delta}_{\Phi}\}} \qquad \text{skip}\ \frac{}{\mathbf{K} \vdash \{^{\Delta}_{\Phi}\}\mathtt{skip}\{^{\Delta}_{\Phi}\}}$$

But now, we can prove that our program does the right thing:

$$\dfrac{\dfrac{\dfrac{\mathtt{hasValue}(\mathtt{wheelsVar},4) \models^{\mathsf{K}} \mathtt{HasFourWheels}(c), \mathtt{hasValue}(\mathtt{wheelsVar},4)}{\mathbf{K} \vdash \{^{-}_{\mathtt{p}\doteq 4}\}\mathtt{nrWheels} := \mathtt{p}\{^{\mathtt{HasFourWheels}(c),\mathtt{hasValue}(\mathtt{wheelsVar},4)}_{\mathtt{nrWheels}\doteq 4}\}}}{\mathbf{K} \vdash \{^{-}_{\mathtt{p}\doteq 4}\}\mathtt{nrWheels} := \mathtt{p}\{^{\mathtt{HasFourWheels}(c),\mathtt{hasValue}(\mathtt{wheelsVar},4)}_{-}\}}}{\mathbf{K} \vdash \{^{-}_{\mathtt{p}\doteq 4}\}\mathtt{nrWheels} := \mathtt{p}\{^{\mathtt{HasFourWheels}(c)}_{-}\}}$$

Conclusion

Summary

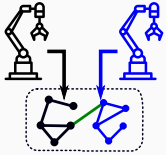


Semantic Lifting and Programming

Interpret program state as knowledge graph and connect with external graph knowledge and data using standard tools.

Semantic Lifting and Digital Twins

Interpret program state as knowledge graph to connect with asset model, express structural correctness with graph queries.



Semantic Lifting and Correctness

Semantical lifting on specification can be used to reason about correctness w.r.t. domain specification.