Declarative Dynamic Object Reclassification

Riccardo Sieve [1] Eduard Kamburjan [2] Ferruccio Damiani [3] Einar Broch Johnsen [1]

[1] University of Oslo, Norway
{riccasi,einarj}@uio.no

[2] IT-University of Copenhagen, Denmark
eduard.kamburjan@itu.dk

[3] University of Turin, Italy
damiani@unito.it

IFIP WG2.2

Aachen, Germany, 24 September 2025

UNIVERSITY
OF OSLO

Talk Overview

Digital twins connect a model to a modeled system

- **Bidirectional link** between model and system, often combined with knowledge base
- **Typical use**: descriptive & predictive analysis
- Current research focuses mostly on systems engineering, these systems **lack theory** today

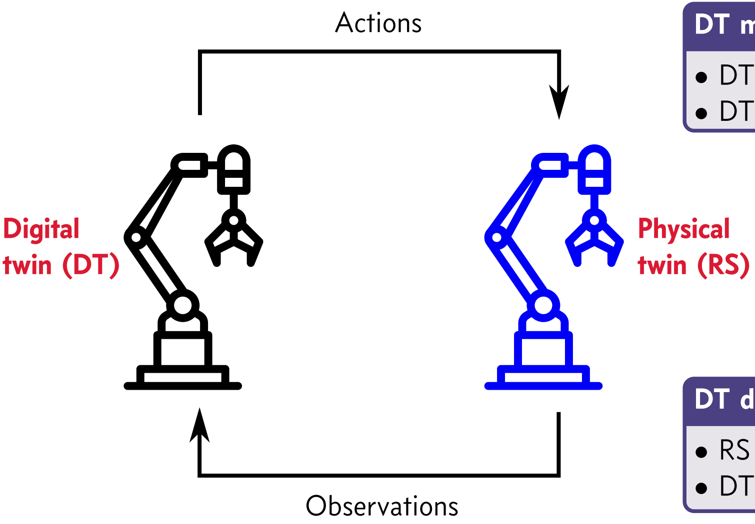
Programming with semantic reflection

- **SMOL**: semantically reflected micro-object language (https://smolang.org/)
- **Basic idea**: programs query an external knowledge base (e.g., sensor data, domain knowledge)
- **Semantic lifting**: function to encode a runtime state in a knowledge graph
- **Semantic reflection**: a program can use reasoners to infer properties about itself

Today: Recent work on formalizing some of these ideas for programming software evolution

Talk based on paper published at **ECOOP 2025** (link)

What is a Digital Twin?



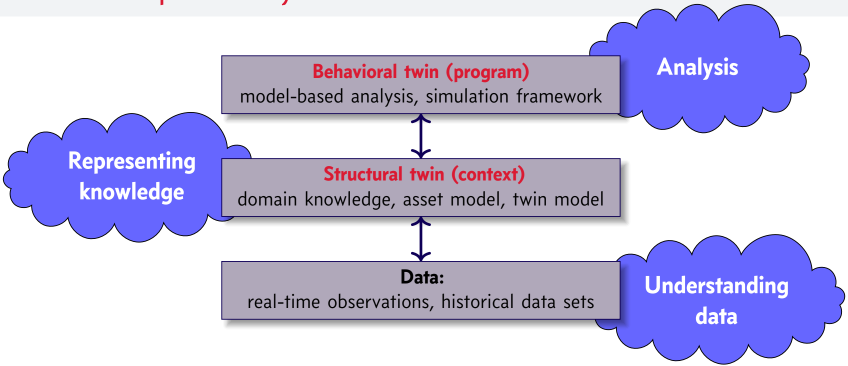
DT model in sync with real system

- DT is a “live replica” of RS
- DT is both model and control

DT depends on external context

- RS can also change over time
- DT must “catch up” with RS

Digital Twins: Conceptual Layers



Can we give guarantees for this kind of systems?

- DT engineering: typically takes a "systems" perspective on these challenges
- In our work, we think of a DT as a complex, dynamic model management problem
- Techniques from self-adaptive systems for autonomous lifecycle management
- Here, we aim to explore this problem from a PL/FM perspective

What is Dynamic Object Reclassification?

Mechanisms that allow class definitions to evolve at runtime

- **Dynamic Software Updates** [3]: external support for patches, no explicit adaptation logic
- **Dynamic Object Reclassification** [2] and **Typestate-Oriented Programming** [1]:
  techniques within the program to support dynamically changing class definitions,
  the adaptation logic programmed as part of the application

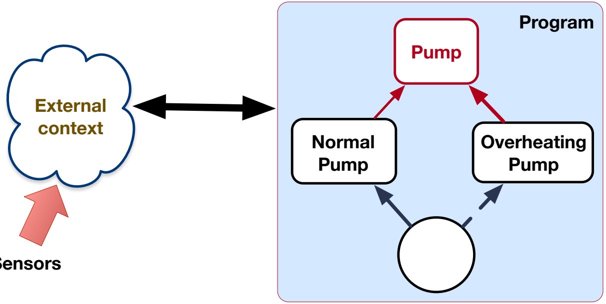
Self-adaptive systems typically need to dynamically adapt to an external context

- This problem lacks support in programming abstractions today
- Can we provide programming abstractions for object reclassification for this purpose?

[1] Aldrich, Sunshine, Saini, Sparks. *Typestate-oriented programming*. OOPSLA 2009
[2] Drossopoulou, Damiani, Dezani-Ciancaglini, Giannini. *Fickle: Dynamic object re-classification*. ECOOP 2001
[3] Hicks, Moore, Nettles. *Dynamic software updating*. PLDI 2001

Motivating Scenario

Consider a digital twin of a greenhouse



A digital twin connects a program to physical system

1. The plants are monitored through **sensors** measuring their physical properties
2. The concrete **watering profile** depends on the stage of the plant (e.g., seedling, mature, ...) and the level of functionality of the pumps (e.g., normal, overheating, ...)

Greenhouse: External Context Formalised as a Knowledge Base

- Let us consider a greenhouse with one plant, and one pump to water the plant
- Remember that pumps may overheat — we here focus on the pump

Domain knowledge:

$$(\textbf{E1})\ \forall x.\ \text{ctx\_in}(x, \text{ctx\_NormalPump}) \Leftrightarrow \left(\text{ctx\_in}(x, \text{ctx\_Pump}) \wedge \exists y.\ \text{ctx\_temp}(x, y) \wedge y \leq 50\right),$$
$$(\textbf{E2})\ \forall x.\ \text{ctx\_in}(x, \text{ctx\_OverheatingPump}) \Leftrightarrow \left(\text{ctx\_in}(x, \text{ctx\_Pump}) \wedge \exists y.\ \text{ctx\_temp}(x, y) \wedge y > 50\right),$$
$$(\textbf{E3})\ \forall x, y, z.\ (\text{ctx\_id}(x, z) \wedge \text{ctx\_id}(y, z)) \Rightarrow x \doteq y,$$
$$(\textbf{E4})\ \forall x, y, z.\ (\text{ctx\_temp}(x, y) \wedge \text{ctx\_temp}(x, z)) \Rightarrow y \doteq z,$$

$$\text{ctx\_in}(\text{ctx\_plant}, \text{ctx\_Plant}), \quad \text{ctx\_in}(\text{ctx\_pump}, \text{ctx\_Pump}),$$
$$\text{ctx\_id}(\text{ctx\_plant}, 1), \quad \text{ctx\_id}(\text{ctx\_pump}, 2), \quad \exists x. \text{ctx\_temp}(\text{ctx\_pump}, x)$$

Synchronisation knowledge:

$$\text{ctx\_temp}(\text{ctx\_pump}, 52)$$

Greenhouse — Program

```
class Plant { int id; String species; }
abstract class Pump {
  int id; int gpioPin; Plant plant;
  void pump(){ ... }; /* uses gpioPin and waters the plant */
}
class NormalPump extends Pump { ... /* methods */ }
class OverheatingPump extends Pump { int maximal;
  ... /* methods */ }
class Main() {
  Plant pl = new Plant(1, "Ocimum basilicum");
  Pump pu = new NormalPump(2, 7, pl);
  void loop() { while (true) { pu.pump(); System.wait(1); } }
  public static void main(String[] args) { new Main().loop(); }
}
```

Heap KB:

isObj(ι_1),
isObj(ι_2),
isObj(ι_3),
instOf(ι_1, Plant),
instOf(ι_2, NormalPump),
instOf(ι_3, Main),
Plant_id(ι_1, 1),
Plant_name(ι_1, "Ocimum basilicum"),
Pump_plant(ι_2, ι_1),
Pump_id(ι_2, 2),
Pump_gpioPin(ι_2, 7)

But ...the Pump ι_2 should actually be an OverheatingPump!

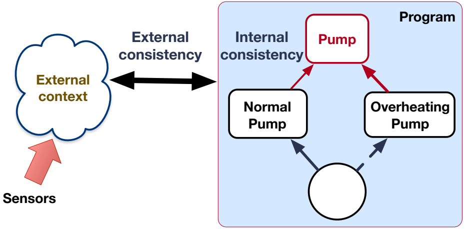
External vs. Internal Consistency

As the physical system evolves, we must ensure the consistency of the program

Internal consistency: Property of the program state (e.g., configurations are well-typed)

External consistency: Relates the program to the external knowledge base

- We address this problem by **dynamic object reclassification**, where external consistency is captured by the adaptation logic
  - Consider an abstract class Pump with two subclasses NormalPump and OverheatingPump. When should a Pump object change class to maintain external consistency?



Instead of entangling this complex adaptation logic in the business code of the program, we express the adaptation logic directly as an inference problem in the knowledge base

Challenges

Declarative dynamic object reclassification separates the adaptation logic of reclassification and the business logic of the program, by expressing the adaptation logic in the knowledge base.

- **Ch1:** How to relate program objects and KB in terms of an **external consistency relation**?
- **Ch2:** How to program **reactions to changes** in the consistency relation between program and KB?
- **Ch3:** How to ensure that establishing external consistency does not **break internal consistency**?

What is a knowledge base?

- The KB is a logical representation of facts and evolves independently of the program
- The KB combines domain knowledge with observations of the physical system
- The KB can be queried for Boolean results ("is a certain formula implied by the KB?") or retrieval ("which values satisfy a given formula?")

Declarative Dynamic Object Reclassification

We realise declarative dynamic object reclassification by combining the following two techniques:

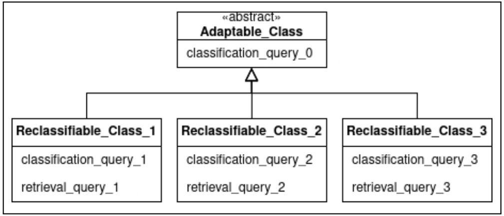
Semantic Reflection

- Allows a program to query its own state in the context of a KB
- **Linkage** connects instances of a program class to the context (defined in class declarations)
- **Lifting** dynamically extends the KB by adding facts about the class membership of runtime objects

Declarative Object Reclassification via queries to the KB

- A **membership** or **classification query** expresses when an object is consistent with a particular class
- A **retrieval query** expresses how to instantiate an object's fields when reclassifying the object
- The queries use semantic reflection to uniformly access both lifted program state and context
- We add **programmatic support** for membership and retrieval queries

Adaptable and Reclassifiable Classes



class Pump (**domain** String id, **Int** GpioPin, Plant plant) { **Unit** pump(){ /* ... */} }
class NormalPump **extends** Pump ()
  links $\lambda self.\ \text{ctx\_in}(self, \text{ctx\_Pump}) \wedge \forall x.\ \text{Pump\_id}(self, x) \Rightarrow \text{ctx\_id}(self, x)$
  classifies $\lambda self.\ \text{ctx\_in}(self, \text{ctx\_NormalPump})$ { /* *methods* */ }
class OverheatingPump **extends** Pump (**Int** maximal)
  links $\lambda self.\ \text{ctx\_in}(self, \text{ctx\_Pump}) \wedge \forall x.\ \text{Pump\_id}(self, x) \Rightarrow \text{ctx\_id}(self, x)$
  classifies $\lambda self.\ \text{ctx\_in}(self, \text{ctx\_OverheatingPump})$
  retrieves $\lambda self, maximal.\ \exists x.\ \text{ctx\_profile}(self, x)$
      $\wedge\ \text{ctx\_maximalPower}(x, maximal)$ { /* *methods* */ }

FSRJ: Featherweight Semantically Reflected Java

- Formalise declarative dynamic object reclassification as a **featherweight calculus**
- Define a **FOL axiomatisation of the KB** for FSRJ programs
- Express **program coherence**: requirements to ensure that reasoning over the KB is meaningful
- Define operational semantics, including **heap lifting** and **querying**
- Prove **type soundness** for FSRJ

FSRJ Syntax

Prg ::=	$\mathcal{K}\ \overline{\text{CD}}$ e	Program
$\mathcal{K}$ ::=	$\{\overline{\phi}\}$	Knowledge base
CD ::=	**class** C [**extends** C] [Links] [Adapt] $\{\overline{\text{FD}}\ \overline{\text{MD}}\}$	Class
FD ::=	T f;	Field
T ::=	C | int	Type
MD ::=	MH {**return** e; }	Method
MH ::=	T m($\overline{\text{T}}\ \overline{\text{x}}$)	Method header
e ::=	x | n | e.f | e.m($\overline{\text{e}}$) | **new** C($\overline{\text{e}}$) | e.f = e | **null** | **adapt**(e)	Expression
Links ::=	**links** $\lambda z.\phi$	Linkage
Adapt ::=	**classifies** $\lambda z.\phi$ [**retrieves** $\lambda z\bar{z}.\phi$];	Adaptation

Here, $\phi \in$ FOL and $\lambda z.\phi$ binds the term variable z in ϕ

Program Typing

We say that a class $C \in \mathrm{dom}(Prg)$ is

- *standard* if *C* has a **links** declaration but no **classifies** or **retrieves** and superclasses are standard
- *adaptable* if *C* has **classifies** but no **links** or **retrieves**, and superclasses are standard
- *reclassifiable* if *C* has **links**, **classifies** and **retrieves**, and an *adaptable* superclass

We assume some **sanity conditions** on programs, e.g., all classes fall into the above categories, all adaptable classes have a reclassifiable subclass, etc

$$\text{(T-new)}\ \frac{\begin{array}{ll}\neg\mathrm{Adp}(C) & \overline{T}\,\overline{f} = \mathrm{fields}(C) \\ \Gamma \vdash \overline{e} : \overline{S} & \overline{S} \leq \overline{T} \\ \multicolumn{2}{c}{D = \begin{cases} \mathrm{Prg}(C)(\mathrm{extends}) & \text{if } \mathrm{Rcl}(C) \\ C & \text{otherwise}\end{cases}}\end{array}}{\Gamma \vdash \mathbf{new}\ C(\overline{e}) : D}$$

$$\text{(T-adapt)}\ \frac{\Gamma \vdash e : C \qquad \mathrm{Adp}(C)}{\Gamma \vdash \mathbf{adapt}(e) : C}$$

A FOL Representation of Knowledge Bases

Given a KB $\mathcal{K}$, we want to infer statements

$$\mathcal{K} \implies \phi[z := \iota],$$

where ι refers to a runtime object

We **parametrise the knowledge base** over an **environment** $\mathcal{E}$ and a runtime **heap** $\mathcal{H}$:

$$\mathcal{K}(\mathcal{H},\mathcal{E}) = \mathcal{K}^{\mathsf{Prg}} \cup \mathcal{K}^{\mathsf{heap}}(\mathcal{H}) \cup \mathcal{K}^{\mathsf{sync}}(\mathcal{E})$$

$\mathcal{K}^{\mathsf{Prg}}$ is **generated** from programs Prg:

- Axioms describing the class structure
- Axioms describing the terms (values, objects, classes)
- Closure and disjointness axioms

$A2.$ $\forall x.\ \mathsf{isCls}(x) \Leftrightarrow (\mathsf{isStd}(x) \oplus \mathsf{isAdp}(x) \oplus \mathsf{isRcl}(x))$

$A4.$ $\mathsf{isCls}(\mathsf{C})$ for all $\mathsf{C} \in \mathit{Cls}$

$A6.$ $\mathsf{isAdp}(\mathsf{C})$ for all $\mathsf{C} \in \mathit{Cls}$ such that $\mathsf{Adp}(\mathsf{C})$

$A11.$ $\mathsf{subclass}(\mathsf{C},\mathsf{C}')$ for all $\mathsf{C},\mathsf{C}' \in \mathit{Cls}$ such that $\mathsf{C} <: \mathsf{C}'$

$A17.$ $\forall x,y.\ \mathsf{in}(x,y) \Leftrightarrow \exists z.\mathsf{instOf}(x,z) \wedge \mathsf{subclass}(z,y)$

$A21.$ $\forall x,y.\ \mathsf{C\_f}(x,y) \Rightarrow \mathsf{in}(x,\mathsf{C}) \wedge \mathsf{hasType}(y,\mathsf{type}(\mathsf{C\_f}))$ for all $\mathsf{C} \in \mathit{Cls}$ and $\mathsf{C.f} \in \mathit{Fls}(\mathsf{C})$

Heap Lifting

$$\mathcal{H}(\iota) = \langle \mathsf{C}, \mathsf{f}_1 = v_1, \ldots, \mathsf{f}_n = v_n \rangle$$

A **heap** $\mathcal{H}$ for a well-typed FSRJ program Prg is a mapping from addresses to objects such that

- $\{\iota \mid \iota \text{ occurs in } \mathcal{H}\} = \mathrm{dom}(\mathcal{H})$, and
- $\mathcal{H}$ only contains instances of non-adaptable classes defined in Prg.

The **lifted-heap knowledge base** $\mathcal{K}^{\text{heap}}(\mathcal{H})$ is then given by

$L1.$ $\mathsf{instOf}(\iota, \mathsf{C})$ for all $\iota \in \mathrm{dom}(\mathcal{H})$

$L2.$ $\mathsf{C}_j\_\mathsf{f}_j(\iota, v_j)$ for all $\iota \in \mathrm{dom}(\mathcal{H})$ and all $\mathsf{C}_j\_\mathsf{f}_j$ such that $\mathsf{C} <: \mathsf{C}_j$, $\mathsf{C}_j.\mathsf{f}_j \in \mathit{Fls}(\mathsf{C}_j)$ and $1 \leq j \leq n$

$L3.$ $\forall x.\ \left(\bigwedge_{\iota \in \mathrm{dom}(\mathcal{H})} (x \neq \iota)\right) \Rightarrow \neg\mathsf{isObj}(x)$

Operational Semantics

Reduction relation has the form

$\mathcal{H} \mid \bar{\iota} \mid e \to \mathcal{H}' \mid \bar{\iota}' \mid e'$

where

- $\mathcal{H}$ is the heap
- $\bar{\iota}$ is the stack
- e is the expression to be evaluated

Recall $\mathcal{K}(\mathcal{H}, \mathcal{E})$ denotes the KB for the heap $\mathcal{H}$ with some environment $\mathcal{E}$

$$\text{(R-adapt-y)}\ \frac{\begin{array}{ccc} & & \mathcal{H}(\iota) = \langle \mathsf{D}, \bar{\mathsf{f}} = \bar{v} \rangle \\ \iota \notin \bar{\iota} & \mathsf{D} \neq \mathsf{D}' & \mathcal{K}(\mathcal{H}, \mathcal{E}) \models \mathsf{compatible}(\mathsf{D}, \mathsf{D}') \\ \multicolumn{2}{c}{\mathbf{classifies}(\mathsf{D}) = \lambda z.\phi} & \mathcal{K}(\mathcal{H}, \mathcal{E}) \not\models \phi[z := \iota] \\ \multicolumn{2}{c}{\mathbf{classifies}(\mathsf{D}') = \lambda z.\phi'} & \mathcal{K}(\mathcal{H}, \mathcal{E}) \models \phi'[z := \iota] \\ \multicolumn{2}{c}{\mathbf{retrieves}(\mathsf{D}') = \lambda z\bar{z}.\psi'} & \mathcal{K}(\mathcal{H}, \mathcal{E}) \models \psi'[z\bar{z} := \iota\bar{v}'] \end{array}}{\mathcal{H} \mid \bar{\iota} \mid \mathbf{adapt}(\iota) \to \mathcal{H}[\iota \mapsto \langle \mathsf{D}', \bar{\mathsf{f}}' = \bar{v}' \rangle] \mid \bar{\iota} \mid \iota}$$

Program Coherence & Type Soundness for Coherent Programs (1)

We need to make sure that the KB is meaningful for declarative object reclassification

A well-typed program with knowledge base $\mathcal{K}^{\text{domain}}$ is *coherent* if:

- **Coh1:** $\mathcal{K}^{\text{domain}}$ is satisfiable
- **Coh2:** each object satisfies the linkage predicate of its class
- **Coh3:** classification-predicates in the KB respect the subclass hierarchy
- **Coh4:** for every instance of a subclass of C, the classification query of at least one of the subclasses of C always holds.
- **Coh5:** reclassified object can be instantiated correctly

Program Coherence & Type Soundness for Coherent Programs (2)

Program coherence and type soundness are mutually dependent:

- Reclassification of a coherent program results in well-typed runtime state
- Heap lifting from a well-typed runtime state maintains program coherence

For coherent programs, we obtain the following type soundness results:

Theorem (Subject reduction)

If $\Theta \Vdash \mathcal{H} \mid \bar{\iota} \mid e : \mathsf{S}$ and $\mathcal{H} \mid \bar{\iota} \mid e \to \mathcal{H}' \mid \bar{\iota}' \mid e'$ then there exists $\Theta' \supseteq \Theta$ such that $\Theta' \Vdash \mathcal{H}' \mid \bar{\iota}' \mid e' : \mathsf{S}'$ for some S' such that $\mathsf{S}' \leq \mathsf{S}$.

We can further characterize the normal forms (**progress**) and thereby obtain **type soundness**

Conclusion

Declarative dynamic object reclassification introduces a separation of concerns between adaptation logic and business code of evolving programs

Technical solution combines KBs, semantic reflection and reclassification queries

Design Choices

- **DC1:** The adaptation logic is expressed in a declarative way, leveraging domain knowledge
- **DC2:** The application logic is expressed by standard class-based object-oriented code
- **DC3:** Adaptation works on (cold) objects in isolation and hot object adaptation gets stuck

Prototype Implementation

- **SMOL:** Semantically reflected language implemented in Kotlin with virtualised heap lifting
- **KB:** Formalised in description logics (decidable fragments of FOL)
- **Program coherence:** statically checkable (up to values of sensor data)