


Declarative Dynamic Object Reclassification

Riccardo Sieve  

University of Oslo, Norway

Eduard Kamburjan  

IT University of Copenhagen, Denmark

Ferruccio Damiani  

University of Turin, Italy

Einar Broch Johnsen  

University of Oslo, Norway

Abstract

In object-oriented languages, dynamic object reclassification is a technique to change the class binding of an object at runtime. Current approaches express when and how to reclassify inside the program's business code, while maintaining internal consistency. These approaches are less suited for programs that need to be consistent with an external context, such as autonomous systems interacting with a knowledge base. This paper proposes declarative dynamic object reclassification, a novel technique that provides a separation of concerns between a program's business code and its adaptation logic for reclassification, expressed via a knowledge base. We present Featherweight Semantically Reflected Java, a minimal calculus for declarative dynamic object reclassification that enables the programmer to define consistency both internally (using a type system) and externally (using declarative classification queries). We use this calculus to study how internal and external consistency interact for declarative dynamic object reclassification. We further implement the technique by extending SMOL, a language for reflective programming via external knowledge bases.

2012 ACM Subject Classification Software and its engineering → Abstraction, modeling and modularity; Software and its engineering → Object oriented languages

Keywords and phrases Dynamic Object Reclassification, Dynamic Software Updates, Featherweight Java, Knowledge Bases, Semantic Reflection, Type Soundness

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.29

Supplementary Material *Software (Artifact)*: <https://doi.org/10.5281/zenodo.15275245>

Software (ECOOP 2025 Artifact Evaluation approved artifact):

<https://doi.org/10.4230/DARTS.11.2.6>

1 Introduction

In traditional object-oriented programming [54, 56], the behaviour of an object instantiated from a class is defined by the methods and properties of that class. However, in many real-world applications, both the object's state and the context in which it operates can change over time. Dynamic object reclassification [15, 16] and typestate-oriented programming [2, 20] are techniques within class-based object-oriented programming that enable *dynamic* class binding of objects at runtime, allowing objects to change their class while retaining their identity. These powerful programming abstractions are particularly useful when an object's context changes over time, and the object needs to adapt to the new context. When the class-binding of objects is dynamic, affected objects need to remain *consistent* with their expected capabilities. For example, consistency becomes critical for dynamic software updates in asynchronous, IoT-driven ecosystems where the local processing of class re-binding may get delayed [34].



© Riccardo Sieve, Eduard Kamburjan, Ferruccio Damiani, and Einar Broch Johnsen;

licensed under Creative Commons License CC-BY 4.0

39th European Conference on Object-Oriented Programming (ECOOP 2025).

Editors: Jonathan Aldrich and Alexandra Silva; Article No. 29; pp. 29:1–29:31



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



Previous approaches to such programming abstractions (e.g., [2, 8, 13–16, 20, 26]) handle the requirements for the reclassification process internally: programs describe classes and reclassification within the same language. This results in (1) a notion of consistency that does not capture the program’s context, (2) an operational and low-level view of the conditions for reclassification, and (3) a lack of separation of concerns between describing object behaviour and object reclassification. In contrast, autonomous and self-adaptive systems in, e.g., robotics and digital twins, typically need to adapt to a context represented by external knowledge bases containing information about both the environment and the system itself [7, 42].

This paper proposes a novel approach to dynamic object reclassification, which shifts the reclassification process from ensuring internal consistency, with an operational view of the adaptation logic, to ensuring external consistency, with a declarative view of the adaptation logic. The approach combines three main elements:

Knowledge Base: The context of the program is modelled as an external knowledge base, i.e., a logical representation of facts. This knowledge base evolves independently of the program and can be queried for Boolean results (“*is a certain formula implied by the knowledge base?*”) or retrieval (“*which values satisfy a given formula?*”).

Semantic Reflection expresses that a program can query its own state in context of the knowledge base; this is enabled by a process of lifting the runtime state of a program into the knowledge base [38].

Declarative Object Reclassification is defined via reclassification queries to the knowledge base. The first query defines when an object is *consistent* with a particular class; i.e., it is a *membership* or *classification* query. The second query that defines how to instantiate an object’s fields when reclassifying an object into a particular class; i.e., it is a *state retrieval* query. These queries make use of semantic reflection, as they uniformly access both the lifted program state and the context in the knowledge base.

In combination, restrictions on queries and on class inheritance ensure that reclassification preserves type safety. In short, the main contributions of this paper are as follows:

- We propose a declarative dynamic object reclassification technique that uses (1) reflection of the program state into a knowledge base, and (2) the additional context of an external knowledge base, to provide an interface for reclassification queries that encapsulates the adaptation logic from the programmer.
- We formalise our technique in Featherweight Semantically Reflected Java (FSRJ), a novel minimal core calculus in the spirit of Featherweight Java [31], which supports semantic reflection into knowledge bases.
- We prove type soundness for FSRJ and give a precise characterisation of *program coherence*, describing the conditions under which reclassification queries ensure type safety.
- We provide a prototype implementation of declarative dynamic object reclassification as an extension of SMOL [38], a language for self-adaptive digital twins [40] based on Knowledge Graphs [29], including a static check for program coherence.
- We evaluate our prototype implementation by application to GreenhouseDT, an open-source digital twin of a mini-greenhouse [41].

The remainder of this paper is structured as follows: Section 2 illustrates declarative dynamic object reclassification with an example before Section 3 describes Featherweight Semantically Reflected Java, the minimal calculus for reflection into a knowledge base and subsequent object reclassification, as well as the type system. Section 4 describes the implementation in the SMOL language and Section 5 the evaluation on a self-adaptive digital twin. Finally, Section 6 discusses related work and Section 7 concludes with a discussion.

```

Java
1 class Plant { int id; String species; }
2 abstract class Pump {
3   int id; int gpioPin; Plant plant;
4   void pump(){ ... }; /* uses gpioPin and waters the plant */
5 }
6 class NormalPump extends Pump { ... /* methods */ }
7 class OverheatingPump extends Pump { int maximal; ... /* methods */ }
8 class Main() {
9   Plant pl = new Plant(1, "Ocimum basilicum");
10  Pump pu = new NormalPump(2, 7, pl);
11  void loop() { while (true) { pu.pump(); System.wait(1); } }
12  public static void main(String[] args) { new Main().loop(); }
13 }

```

■ **Figure 1** Greenhouse digital twin, without domain knowledge and dynamic object reclassification.

2 Overview

This section introduces the programming challenges that we address with declarative dynamic object reclassification, and motivates different technical aspects of our solution. To ease readability in the examples in this section and in the formalisation of Section 3, programs are given with a Java-like syntax and knowledge bases in first-order logic (FOL).

2.1 Programming Challenges

Motivating Scenario. Let us consider as a running example a digital twin of a greenhouse which contains plants and water pumps. These are monitored through sensors measuring their physical *properties*: the height, oxygen level and soil moisture for each plant and the temperature for each pump. Each plant has an associated pump that waters the plant according to a watering profile. The concrete watering profile depends on the stage of the plant (e.g., seedling or mature plant) and the level of functionality of the pump. We first consider a simple configuration of the example in Sections 2 and 3, with one pump, two plants and a temperature monitor for the pump (see Figure 1); we later consider a larger configuration in Section 5, with three pumps, three plants and monitoring more properties.

In the digital twin, each plant is modelled by a `Plant` object and each pump by a `Pump` object. The `Plant` class contains the plant’s identifier and scientific `species` name, so further information about the plant can be retrieved from a *domain knowledge base*. The `Pump` class contains the pump’s identifier, a reference `p1` to the plant it is watering and a reference `gpioPin` to the output pin needed to operate the physical pump. The `pump` method implements the watering profile for the pump, which depends on the stage of the plant and whether the pump is fully operational or not: A pump is either operating normally (`NormalPump`) or overheated and in need of maintenance (`OverheatingPump`). To avoid overheating, an `OverheatingPump` should run at reduced power. Let us further assume that sensors (not specified here) update a *synchronisation knowledge base* containing the current state of the greenhouse. Thus, our program can retrieve information about the greenhouse from the knowledge base as needed.

The physical entities in the greenhouse evolve over time (e.g., a pump may threaten to overheat). The objects in the digital twin need to adapt their behaviour in accordance with the evolution of the physical entities they are twinning. To address this kind of problem, we propose a dynamic object reclassification technique that is *declarative*; i.e., the adaptation logic for reclassification is derived in the knowledge base. We now introduce a knowledge base for our example, and extend our program with constructs to interact with this knowledge base.

Knowledge Bases and Consistency. A knowledge base is used to organise and make contextual information accessible via an API. To keep the presentation abstract, we here represent knowledge in terms of first-order logic formulas and structure the knowledge base for our digital twin as follows: (1) *domain knowledge* about plants and pumps, see Figure 2a, including the description of the physical pumps and plants in the greenhouse (the so-called domain objects); and (2) *synchronisation knowledge* about the current value of the properties of these domain objects, see Figure 2b. Observe how *consistency* here comes into play: a `Pump` object in the digital twin should be classified in the subclass of `Pump` implementing the watering policy that is appropriate for the current stage of the pump domain object it twins and controls.

We use patterns from ontologies and knowledge representation to model domain knowledge, represent domain objects and domain classes as constants (nullary predicates), and prefix all predicates concerning the domain by `ctx_`. The *membership* of a domain object (e.g., `ctx_o`) in a domain class (e.g., `ctx_C`) is expressed by a binary predicate `ctx_in(ctx_o, ctx_C)`. the *value* x of a property `ctx_prop` for a domain object y by the predicate `ctx_prop(y, x)`. For instance, the temperature x of a pump y is expressed by the predicate `ctx_temp(y, x)`.

The formulas **E1–E4** in Figure 2a express general knowledge about pumps; the remaining formulas specific knowledge about domain objects (here, one plant and one pump). Formula **E1** states that a pump (`ctx_Pump`) is operating normally (`ctx_NormalPump`) if its temperature is $\leq 50^\circ\text{C}$, and **E2** that a pump is in danger of overheating (`ctx_OverheatingPump`) if its temperature is $> 50^\circ\text{C}$. Formulas **E3** and **E4** express that the property `ctx_id` uniquely identifies a domain object and that each pump can have only one temperature. The remaining formulas describe the plant and pump in our greenhouse, represented by constants `ctx_plant` and `ctx_pump`; `ctx_Plant` is the domain class of all plants. The last formula states that the pump `ctx_pump` has some temperature. The synchronisation knowledge base, in Figure 2b, contains, for each non-constant property `ctx_prop` of the system, an axiom that states the current value of `ctx_prop`. It contains a formula that states the actual temperature of `ctx_pump`.

To express consistency, we need to relate the `Pump` program objects to the current state of the actual pumps in the greenhouse, as represented by the `ctx_Pump` domain objects. To this aim, we represent the program code and its runtime configuration as knowledge bases. Figure 2c illustrates how the code in Figure 1 can be represented by a *code knowledge base*: predicate `isCls(x)` express whether x is a program program class; and `subclass(x, y)` express whether x is a subclass of y .

The representation of a program configuration in a knowledge base is known as *semantic lifting* [38], and uses a function defined in terms of the language to serialise runtime configurations into a *lifted-heap* knowledge base, where: `isObj(x)` express whether x is a program object (reference); `instOf(x, y)` whether x is an object instance of class y ; `in(x, y)` whether x is an object instance of a subclass of y ; and `C_f(x, y)` express whether x is an object in class C and its field f has value y . Figure 2d shows the lifted runtime configuration of the program in Figure 1 after execution of the expression `new Main()` in Line 14 and field initialisation in Lines 11 and 12: here, the program object ι_1 , which is stored in `p1`, is an instance of program class `Plant` and ι_2 is an instance of program class `NormalPump`; and `Pump_plant(ι_2, ι_1)` states that the field `plant` in ι_2 of program class `Plant` has value ι_1 .

Observe that the fact **D1** `ctx_in(ctx_pump, ctx_OverheatingPump)` can be derived in the knowledge base in Figure 2 (by instantiating the universal quantifier in the axioms **E2** and **E3** with `ctx_plant`), which shows that the lifted runtime configuration in Figure 2d (stating that ι_2 is an instance of `NormalPump`) is *not consistent* with the context. To enable

(E1) $\forall x. \text{ctx_in}(x, \text{ctx_NormalPump}) \Leftrightarrow (\text{ctx_in}(x, \text{ctx_Pump}) \wedge \exists y. \text{ctx_temp}(x, y) \wedge y \leq 50)$,
 (E2) $\forall x. \text{ctx_in}(x, \text{ctx_OverheatingPump}) \Leftrightarrow (\text{ctx_in}(x, \text{ctx_Pump}) \wedge \exists y. \text{ctx_temp}(x, y) \wedge y > 50)$,
 (E3) $\forall x, y, z. (\text{ctx_id}(x, z) \wedge \text{ctx_id}(y, z)) \Rightarrow x \doteq y$,
 (E4) $\forall x, y, z. (\text{ctx_temp}(x, y) \wedge \text{ctx_temp}(x, z)) \Rightarrow y \doteq z$,
 $\text{ctx_in}(\text{ctx_plant}, \text{ctx_Plant}), \quad \text{ctx_in}(\text{ctx_pump}, \text{ctx_Pump}),$
 $\text{ctx_id}(\text{ctx_plant}, 1), \quad \text{ctx_id}(\text{ctx_pump}, 2), \quad \exists x. \text{ctx_temp}(\text{ctx_pump}, x)$

(a) The *domain knowledge base* for the greenhouse.

$\text{ctx_temp}(\text{ctx_pump}, 52)$

(b) A *synchronisation knowledge base* describing the temperature of the pump.

$\text{isCls}(\text{Plant}), \quad \text{isCls}(\text{Pump}), \quad \text{isCls}(\text{NormalPump}), \quad \text{isCls}(\text{OverheatingPump}), \quad \text{isCls}(\text{Main}),$
 $\text{subclass}(\text{NormalPump}, \text{Pump}), \quad \text{subclass}(\text{OverheatingPump}, \text{Pump})$

(c) A snippet of the *code knowledge base* describing the code in Figure 1.

$\text{isObj}(\iota_1), \quad \text{isObj}(\iota_2), \quad \text{isObj}(\iota_3), \quad \text{instOf}(\iota_1, \text{Plant}), \quad \text{instOf}(\iota_2, \text{NormalPump}),$
 $\text{instOf}(\iota_3, \text{Main}), \quad \text{Plant_id}(\iota_1, 1), \quad \text{Plant_name}(\iota_1, \text{"Ocimum basilicum"}),$
 $\text{Pump_plant}(\iota_2, \iota_1), \quad \text{Pump_id}(\iota_2, 2), \quad \text{Pump_gpioPin}(\iota_2, 7)$

(d) The *lifted-heap knowledge base*, for the program's heap after the `Main` object has been initialised.

■ **Figure 2** Knowledge bases for the digital twin: domain, synchronisation, code and lifted heap.

declarative dynamic object reclassification, we need to connect program objects (e.g., ι_1 and ι_2) and classes (e.g., `Pump`, `NormalPump` and `OverheatingPump`) to corresponding domain objects (e.g., `ctx_plant` and `ctx_pump`) and classes (e.g., `ctx_Pump`, `ctx_NormalPump` and `ctx_OverheatingPump`) in the knowledge base. In particular, we face the following challenges:

- Ch1:** How can we relate program objects with the external knowledge base, and define *external consistency*, i.e., consistency between program and external knowledge base?
- Ch2:** How can we program reactions to changes in the consistency of the relation between program and knowledge base?
- Ch3:** How can we ensure that establishing external consistency does not break internal consistency, i.e., the typing of the program?

Knowledge Bases as Interoperability Layers. While our approach is general to dynamic reclassification, it is worth commenting on the role of the knowledge base, which is motivated by digital twins. In digital twins, the focus is not on highly complex low-level control, but on modularity, composition, and semantic interoperability of different components, where knowledge bases are widely used [42, 60]. Often, all communication between the program and the physical component is handled via the knowledge base. The program is acting as a controller *and* a coordinating component that acts on data that has been already ingested into the knowledge base. This approach, which lends to blackboard architectures [25], also motivates our terminology of *external consistency*: The program is consistent with an external component (namely the physical twin), as described by the synchronization knowledge base. From a model-based perspective, this corresponds to *intermodel consistency* (see, e.g., Feichtinger et al. [19]), in contrast to internal, or intramodel, consistency, which corresponds to type soundness.

Extended Java code

```

1 class Plant { int id; String species; }
2 abstract class Pump classifies λself. ctx_in(self, ctx_Pump) {
3   int id; int gpioPin; Plant plant;
4   void pump(){ ... }; /* uses gpioPin and waters the plant */
5 }
6 class NormalPump extends Pump
7   links λself. ctx_in(self, ctx_Pump) ∧ ∀x. Pump_id(self, x) ⇒ ctx_id(self, x)
8   classifies λself. ctx_in(self, ctx_NormalPump) { ... /* methods */ }
9 class OverheatingPump extends Pump
10  links λself. ctx_in(self, ctx_Pump) ∧ ∀x. Pump_id(self, x) ⇒ ctx_id(self, x)
11  classifies λself. ctx_in(self, ctx_OverheatingPump)
12  retrieves λself, maximal. ∃x. ctx_profile(self, x) ∧ ctx_maximalPower(x, maximal)
13  { int maximal; ... /* methods */ }
14 class Main {
15   Plant pl = new Plant(1, "Ocimum basilicum");
16   Pump pu = new NormalPump(2, 7, pl);
17   void loop() { while (true) { adapt(pu); pu.pump(); System.wait(1); } }
18   public static void main(String[] args) { new Main().loop(); }
19 }

```

(a) The greenhouse digital twin programmatic part with **links**, **classifies**, **retrieves** and **adapt**.

$$\begin{aligned} \forall y. \text{instOf}(y, \text{NormalPump}) &\Rightarrow (\text{ctx_in}(y, \text{ctx_Pump}) \wedge \forall x. \text{Pump_id}(y, x) \Rightarrow \text{ctx_id}(y, x)), \\ \forall y. \text{instOf}(y, \text{OverheatingPump}) &\Rightarrow (\text{ctx_in}(y, \text{ctx_Pump}) \wedge \forall x. \text{Pump_id}(y, x) \Rightarrow \text{ctx_id}(y, x)) \end{aligned}$$

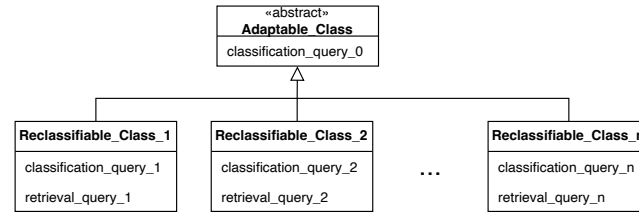
(b) Linkage knowledge base. Tautologies from the default **links** clauses of **Plant** and **Main** are omitted.

■ **Figure 3** The greenhouse digital twin programmatic part and the linkage knowledge base.

2.2 Declarative Dynamic Object Reclassification

Based on the knowledge base and scenario above, we now illustrate the mechanisms in the programming language to connect the program with its context at runtime, shown in Figure 3. The formalisation and properties for these mechanisms will be explored in Section 3.

Linkage. To address the first part of challenge *Ch1*, we propose to dynamically add knowledge about the dynamically created objects of the program’s runtime state to the knowledge base. Technically, we add to each non-**abstract** class **C** a **links** clause with a single-parameter λ -predicate $\lambda self. \phi$ to express that $\forall x. \text{instOf}(x, \mathbf{C}) \Rightarrow \phi[self := x]$ is added to the knowledge base (the $\lambda self. \text{true}$ here acts as default). At runtime, dynamically created objects are lifted into the lifted-heap knowledge base by adding a fact about their class membership, e.g., $\text{instOf}(\iota, \mathbf{C})$ for some object ι (cf. Section 2.1). In Figure 3 (Section 2.1), the program class **NormalPump** declares its objects to be members of domain class **ctx_NormalPump** and the domain id (**ctx_id**) and program id (**Pump_id**) to be the same. Similarly for **OverheatingPump** (Section 2.1). Linkage connects the objects of **C** to the context, using domain terminology (i.e., predicate symbols). Figure 3b shows the *linkage knowledge base* containing the added axioms. The first states that every program object that is an instance of **NormalPump**, is also a pump (i.e., a **ctx_Pump** in the domain context), and that the ids are the same. The same states the same for **OverheatingPump**. This links the program object to the domain knowledge, as one can use *domain* axioms (such as **E1** and **E2**) to reason about the *program* objects. This is done as follows. From the lifted heap in Figure 2d, we derive that ι_2 is member of **ctx_OverheatingPump** as follows: By lifting, we have $\text{instOf}(\iota_2, \text{NormalPump})$ and by linkage, $\text{ctx_id}(\iota_2, 2)$. Since **ctx_id** uniquely identifies an object (cf. **E3**), $\text{ctx_pump} \doteq \iota_2$. Applying this equality in the formula $\text{ctx_in}(\text{ctx_pump}, \text{ctx_OverheatingPump})$ (**D1**, see above), we obtain $\text{ctx_in}(\iota_2, \text{ctx_OverheatingPump})$.



■ **Figure 4** The structure of a class hierarchy for declarative dynamic object classification.

Consistency Declaration. To address the second part of challenge *Ch1*, we propose to characterise in the knowledge base when an object should belong to a particular class. To this aim, we let objects change between *reclassifiable* subtypes of an *adaptable* (abstract) class. Technically, we add a **classifies** clause with a single-parameter λ -predicate to these classes. If this λ -predicate holds when instantiated for a program object of the given program class, the program object is considered consistent with its context. In our example (see Figure 2), ι_2 is *not* consistent. The only classes with a **classifies** clause are

1. *adaptable* classes, i.e., **abstract** classes that extend classes with no **classifies** clause; and
2. *reclassifiable* classes, i.e., all the subclasses of the adaptable classes (introduced in point 1), that must be non-**abstract** and have no subclasses.

Remark that, as illustrated in Figure 4, each reclassifiable class must (1) directly inherit from an adaptable class; (2) not be inherited from; and (3) have only reclassifiable sibling classes.

Reclassification Queries. Reclassification is the process of changing the program class of an inconsistent program object to a program class that makes the program object consistent. To address challenge *Ch2*, we provide language support for programming such reclassification. The expression **adapt**(e) (Section 2.1) takes a program object that must be an instance of a reclassifiable class as argument, and checks whether this program object is consistent with the context; if this is not the case, it is reclassified. This is done by searching the knowledge base for a sibling to its current program class in the class hierarchy, that would make the object consistent. For the running example (see Figure 2), this would be the **OverheatingPump** class. Remark that the **adapt** expression is an explicit mechanism to ensure external consistency, but far from the only one. External consistency can also be established implicitly, either lazily, whenever an object is accessed, or periodically in the background, akin to a garbage collector.

To support state transfer for the reclassified object to the fields of its new program class, the programmer adds a **retrieves** clause to each reclassifiable class. This clause provides a λ -predicate with one parameter for the object in question and one for each field declared in the class. We can then infer values from the knowledge base that make the λ -predicate hold. Consider the **OverheatingPump** in Figure 3a: the reclassification query (Section 2.1) first retrieves the profile of the given pump, and then the maximal power it can use when in this stage.

Ensuring Consistency and Type Soundness. To address challenge *Ch3*, let us observe that it is crucial that the dynamic operations on the knowledge base maintain satisfiability of the knowledge base – otherwise, the program’s interactions with the knowledge base would allow incorrect information to be inferred, potentially violating type soundness. Technically, we first ensure that reclassifiable classes are not used as types; e.g., a variable of type **OperationalPump** is not allowed, as it may cause an error when accessed after reclassification. Both this restriction and the structure illustrated in Figure 4 are enforced by the type system; they guarantee that no unsafe access is possible.

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

■ **Figure 5** FSRJ syntax, where ϕ ranges over first-order logic (FOL) formulas, C over class names, f over field names, T over types (i.e., class names and the basic type `int`), m over method names, x over local variables (i.e., formal parameters to methods) and z over term variables (so $\lambda z.\phi$ is a unary λ -predicate and $\lambda z\overline{z}.\phi$ a λ -predicate of arity ≥ 1).

To ensure that the interaction between the program and the knowledge base does not introduce errors, we introduce a notion of *program coherence* roughly expressing that

1. every lifted object of a reclassifiable class can be classified as member of the adaptable superclass;
2. every lifted object that is member of an adaptable class can also be classified as member of one of the reclassifiable subclasses; and
3. the **retrieves** λ -predicate associates a value of the correct type to each field.

We say that an object is *hot* to mean that a method invoked on the object has an activation record on the stack, and *cold* otherwise. In the next section we will see that program coherence ensures that reclassification of cold objects is always successful.

3 FSRJ: Featherweight Semantically Reflected Java

This section provides a formal account of programming with semantic reflection in terms of *Featherweight Semantically Reflected Java (FSRJ)*, a minimal core calculus for class-based object-oriented languages with semantic reflection. Our purpose with FSRJ is to formalise the interplay between program execution, semantic reflection and the querying of knowledge bases for dynamic object reclassification. FSRJ is a variant of Featherweight Java (FJ) [31].¹

3.1 FSRJ Syntax

The syntax of FSRJ is given in Figure 5. We let $\overline{}$ represent (possibly empty) sequences and $[\cdot]$ represent optional elements. Operations on sequences of pairs are abbreviated in a similar way; e.g., $\overline{C} \overline{f}$ is shorthand for $C_1 f_1, \dots, C_n f_n$. The empty sequence is denoted by \bullet and the length of a sequence \overline{e} is denoted by $\#(\overline{e})$. The set of program variables includes the reserved name `this`, which cannot be used as a name of a field or a method's formal parameter.

In FSRJ, a program Prg consists of a knowledge base \mathcal{K} , a list of class definitions $\overline{\text{CD}}$ and a main expression e . The knowledge base \mathcal{K} , called the *domain knowledge base* of Prg , is set of FOL formulas. A class definition **class** C [**extends** D] [**Links**] [**Adapt**] $\{\overline{\text{FD}} \overline{\text{MD}}\}$ consists of

¹ FSRJ does not technically extend FJ because cast-expressions are excluded; these are orthogonal to semantic reflection and object reclassification. Further, FSRJ is imperative (it features a field assignment expression), has no implicit root class (`Object`), and classes in FSRJ need not have a superclass and have an implicit constructor which initialises all the fields.

a class name C , an optional clause **extends** D to express inheritance (so D is the immediate superclass of C), an optional linkage clause $\text{Links} = \text{links } \lambda z. \phi$, an optional adaptation clause $\text{Adapt} = \text{classifies } \lambda z. \phi [\text{retrieves } \lambda z\bar{z}. \psi]$ (where, in turn, the **retrieves** part is optional), a list of field definitions \overline{FD} and a list of method definitions \overline{MD} . The *knowledge base*, formalising the context in which the program executes, is detailed in Section 3.2. The *linkage clause* expresses that every instance o of C is logically linked by $\lambda z. \phi$; i.e., given the address ι of o , the formula $\phi[z := \iota]$ holds in the context of the program. The *adaptation clause* expresses that if C has the immediate superclass D , then every instance o of an immediate subclass of D is reclassifiable to C by $\lambda z. \phi$ and $\lambda z\bar{z}. \psi$; i.e., given the address ι of o , if the formula $\phi[z := \iota]$ holds in the context of the program then there exist values \bar{v} for the fields of C such that the formula $\psi[z\bar{z} := \iota\bar{v}]$ holds in the context of the program. Types τ include names of declared classes and the basic type **int**. Field and method declarations are standard; the fields declared in a class are added to the ones declared in its superclasses, and are assumed to have distinct names (i.e., no field shadowing is allowed). Method names are assumed to be unique in the class; i.e., no method overloading is allowed. It is, however, possible to override methods from the superclass. All fields and methods are public, and each class has an implicit constructor that takes a parameter for each field (inherited or defined) in the class and initialises all the fields.

We assume that the elements of lists of named entities (i.e., class definitions, field definitions, method definitions, method formal parameter declarations) have different names. To simplify the presentation, we use a program Prg as a mapping from class names C to the corresponding class definitions CD ; and use a class definition CD as a mapping that maps field and method names to field and method definitions, may map the keyword **extends** to the name of its superclass D , may map each of the keywords **links** and **classifies** to a unary λ -predicate of arity one and the keyword **retrieves** to a λ -predicate of arity $n + 1$, where n is the number of fields (defined or inherited) in the class. We assume that programs Prg satisfy the following *well-formedness conditions*:

1. For a class name C appearing anywhere in Prg , we have $C \in \text{dom}(\text{Prg})$.
2. There are no cycles in the transitive closure of the immediate **extends** relation.

The *subclassing* relation $<$ is the reflexive and transitive closure of the immediate **extends** relation. The *subtyping* relation \leq extends $<$ by adding reflexivity on basic types, i.e., $\text{int} \leq \text{int}$.

In the sequel, we let the metavariable a range over names and the metavariable AD over definitions. The lookup of the definition of a field or method a in a class C is denoted by $\text{aDef}(C)(a)$. For each class in $\text{dom}(\text{Prg})$, the function $\text{aDef}(C)$ is defined as follows:

$$\text{aDef}(C)(a) = \begin{cases} \text{Prg}(C)(a) & \text{if } a \in \text{dom}(\text{Prg}(C)) \\ \text{aDef}(D)(a) & \text{if } a \notin \text{dom}(\text{Prg}(C)) \text{ and } \text{Prg}(C)(\text{extends}) = D \end{cases}$$

Given a field definition $\text{FD} = \tau \text{ f}$ and a method definition $\text{MD} = \tau \text{ m } (\bar{\tau} \bar{x}) \{ \dots \}$, let $\text{signature}(\text{FD})$ denote the type τ of field f and $\text{signature}(\text{MD})$ the type $\bar{\tau} \rightarrow \tau$ of method m .

► **Example 1** (Greenhouse digital twin in FSRJ). Let us consider the encoding of the Greenhouse example (Figure 3a) in FSRJ. Sequential composition, which is not part of the syntax of FSRJ, can be encoded. For $e_1; \dots; e_n$ of $n \geq 1$ expressions, define an auxiliary class **Encode** with a method $\text{T}_n \text{ sqT1_T2_} \dots \text{T}_n (\text{T}_1 x_1, \dots, \text{T}_n x_n) \{ \text{return } e_n; \}$, where T_i is the type of e_i ($1 \leq i \leq n$). Namely, $e_1; \dots; e_n$ can be expressed as **new Encode().sqT1...Tn**(e_1, \dots, e_n).

Then, if we ignore the keyword **abstract**, consider **void** as syntactic sugar for **int**, consider the string literal "Ocinum basilicum" as syntactic sugar for an integer literal, and consider **System.wait(1)** as syntactic sugar for **new System().wait(1)** where **System** is a

class providing a method `int wait(int x) {...}`, we have that the Java-like code fragment in Figure 3a can be turned into an FSRJ code fragment by replacing the code of the class `Main` (in lines 16-21) by the following FSRJ code:

FSRJ

```
class Main { Plant pl; Pump pu;
  int loop(){ return new Encode().scPump_Plant_int_int_int(pu.plant=this.pl,
    adapt(pu), pu.pump(), System.wait(1), this.loop()); }
} // Main expression:
new Main(new Plant(1,"Ocimum basilicum"), new NormalPump(2,7,null)).loop()
```

3.2 A FOL Representation of Knowledge Bases

To avoid the technicalities of knowledge representation languages in FSRJ, the knowledge base \mathcal{K} of an FSRJ program is given as a set of closed FOL formulas over a given signature. Formally, signatures and formulas over signatures are defined as follows.

► **Definition 2 (Signatures).** A signature is a set of predicate symbols $\Sigma = \Sigma_0 \cup \Sigma_1 \cup \Sigma_2$, where Σ_0 is a set of 0-ary predicate symbols (constants), Σ_1 a set of unary predicate symbols, and Σ_2 a set of binary predicate symbols (relations). Let c range over Σ_0 , P over Σ_1 and R over Σ_2 .

► **Definition 3 (FOL formulas and terms over a signature).** Let term variables z range over constants in a signature Σ . The formulas ϕ and terms t over Σ are defined as follows:

$$\begin{aligned} \phi &::= \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi \mid \phi \oplus \phi \mid \phi \Rightarrow \phi \mid \phi \Leftrightarrow \phi \mid \exists z. \phi \mid \forall z. \phi \mid P(t) \mid R(t, t) \mid t \doteq t \\ t &::= z \mid c \end{aligned}$$

Thus, in a λ -predicate $\lambda z\bar{z}.\psi$, $z\bar{z}$ is a (non-empty) list of term variables and ϕ a FOL formula. We denote by $\text{FV}(t)$ the free variables in a term t and by $\text{FV}(\phi)$ the free variables in a formula ϕ . The substitution $\phi[z := t]$ replaces all free occurrences of the term variable z by the term t in the formula ϕ (we omit its inductive definition here). Given a knowledge base \mathcal{K} and a formula ψ , we let $\mathcal{K} \Rightarrow \psi$ abbreviate $(\bigwedge_{\phi \in \mathcal{K}} \phi) \Rightarrow \psi$.

Recall that the semantics of FOL with equality defines the values of terms to be individuals of a domain with an identity-relation, and the values of (well-formed) formulas to be truth values [4]. Thus, the semantics of a formula over a signature Σ is defined relative to an equality-model $\mathcal{M} = \langle \mathcal{D}, \mathcal{I}, \approx \rangle$ and a variable assignment β that maps term variables to elements in the domain \mathcal{D} , such that: (1) the interpretation $\mathcal{I} : \Sigma \rightarrow \mathcal{D}^n$ maps symbols in Σ_2 to binary relations over $\mathcal{D} \times \mathcal{D}$, symbols in Σ_1 to unary predicates over \mathcal{D} , and symbols in Σ_0 to distinct elements in \mathcal{D} ; and (2) the interpretation $\mathcal{I}(\doteq)$ is the equality-relation \approx on \mathcal{D} . In the following definition of satisfiability, we omit the operators \forall , \wedge , \oplus (which denotes exclusive or), \Rightarrow and \Leftrightarrow , since their semantics can be derived by the semantics of the other operators.

► **Definition 4 (Satisfiability).** Let Σ be signature, $\mathcal{M} = \langle \mathcal{D}, \mathcal{I}, \approx \rangle$ an equality-model such that $\mathcal{I} : \Sigma \rightarrow \mathcal{D}$, and β a variable assignment. The values of terms t with respect to β in \mathcal{M} , denoted $\llbracket t \rrbracket_{\mathcal{M}, \beta}$, are defined by $\llbracket z \rrbracket_{\mathcal{M}, \beta} = \beta(z)$ and $\llbracket c \rrbracket_{\mathcal{M}, \beta} = \mathcal{I}(c)$. The relation $\mathcal{M}, \beta \models \phi$, expressing that a formula ϕ is satisfied by \mathcal{M} and β , is defined inductively as follows:

$$\begin{aligned} \mathcal{M}, \beta \models \neg\phi & \quad \text{iff } \mathcal{M}, \beta \not\models \phi \\ \mathcal{M}, \beta \models \phi_1 \vee \phi_2 & \quad \text{iff } \mathcal{M}, \beta \models \phi_1 \text{ or } \mathcal{M}, \beta \models \phi_2 \\ \mathcal{M}, \beta \models \exists x. \phi & \quad \text{iff } \mathcal{M}, \beta[x \mapsto d] \models \phi \text{ for some } d \in \mathcal{D} \\ \mathcal{M}, \beta \models P(t) & \quad \text{iff } \llbracket t \rrbracket_{\mathcal{M}, \beta} \in \mathcal{M}(P) \\ \mathcal{M}, \beta \models R(t_1, t_2) & \quad \text{iff } (\llbracket t_1 \rrbracket_{\mathcal{M}, \beta}, \llbracket t_2 \rrbracket_{\mathcal{M}, \beta}) \in \mathcal{M}(R) \end{aligned}$$

A formula ϕ over Σ is (1) satisfiable if $\mathcal{M}, \beta \models \phi$ holds for some \mathcal{M} and β , and (2) valid, denoted $\models \phi$, when ϕ is satisfiable for every \mathcal{M} and variable assignment.

A knowledge base \mathcal{K} is satisfiable/valid if $\bigwedge_{\phi \in \mathcal{K}} \phi$ is satisfiable/valid.

Given a knowledge base \mathcal{K} and a formula ψ , we write $\mathcal{K} \models \psi$ as shorthand for $\models \mathcal{K} \Rightarrow \psi$.

► **Example 5** (Domain knowledge base for the Greenhouse digital twin in FSRJ). The domain knowledge base for the FSRJ Greenhouse digital twin is given in Figure 2a.

3.3 Program Typing

In this section we present the type system for FSRJ, detailing especially on the language aspects related to knowledge bases, semantic reflection and dynamic object reclassification.

The function `type`, that extracts the type from a field or method definition, is defined by: `type(T f) = T` and `type(T m(\bar{T} \bar{x}) { \dots }) = $\bar{T} \rightarrow T$. The lookup of the type of a field or method a in the program is defined by: aType(C)(a) = type(aDef(C)(a)). With respect to reclassification, we say that a class $c \in \text{dom}(\text{Prg})$ is:`

- *standard*, written `Std(c)`, if (1) `links` $\in \text{dom}(\text{Prg}(c))$; (2) `classifies` $\notin \text{dom}(\text{Prg}(c))$; (3) `retrieves` $\notin \text{dom}(\text{Prg}(c))$; and (4) if `c` has a superclass, then this superclass is standard (i.e., if `Prg(c)(extends) = B` for some `B`, then `Std(B)`).
- *adaptable*, written `Adp(c)`, if (1) `links` $\notin \text{dom}(\text{Prg}(c))$; (2) `classifies` $\in \text{dom}(\text{Prg}(c))$; (3) `retrieves` $\notin \text{dom}(\text{Prg}(c))$; and (4) if `c` has a superclass, then this superclass is standard (i.e., if `Prg(c)(extends) = B` for some `B`, then `Std(B)`).
- *reclassifiable*, written `Rcl(c)`, if (1) `links` $\in \text{dom}(\text{Prg}(c))$; (2) `classifies` $\in \text{dom}(\text{Prg}(c))$; (3) `retrieves` $\in \text{dom}(\text{Prg}(c))$; and (4) `c` has a superclass which is adaptable (that is, `Prg(c)(extends) = B` and `Adp(B)`, for some `B`).

By inspecting the program, it is possible to check that there are no cycles in the transitive closure of the `extends` relation; that, for each class `c` in `dom(Prg)`, the names of the fields in `c` are distinct from the names of inherited fields; that each class is either standard, or adaptable, or reclassifiable; and that each adaptable class has at least a subclass (which is necessarily reclassifiable). We write `sane(Prg)` to express that the program `Prg` satisfies the well-formedness conditions in Section 3.1 and that the following sanity conditions:

1. `c1 <: c2` implies that, for all method names `m`, if `aType(c2)(m)` is defined, then `aType(c1)(m) = aType(c2)(m)`.
2. `c1 <: c2` and `c1 ≠ c2` imply that, for all field names `f`, if `f ∈ dom(Prg(c1))` then `f ∉ dom(Prg(c2))`.
3. For every `c ∈ dom(Prg)`, exactly one of `Std(c)`, `Adp(c)` or `Rcl(c)` holds.
4. For every `c ∈ dom(Prg)`, if `Adp(c)` then there exists `d ∈ dom(Prg)` s.t. `Prg(d)(extends) = c`.
5. Reclassifiable class names are not used as field types, as method parameters or return types.

The typing rules and field lookup rule for FSRJ are given in Figure 6. Rule T-knowledge enforces that the domain knowledge base consists of closed formulas. The typing of expressions is standard, except for rule T-new, which enforces new objects to be typed either by an adaptable or a standard class, and rule T-adapt, which enforces the `adapt`-expression and the argument of `adapt` to be typed by an adaptable class `c`. This allows the argument of `adapt` to be dynamically reclassified to any subclass of `c`.

The typing of `links` clauses ensures that the linkage formula only contains one free variable `z`, which ensures that the formula can be closed by instantiating `z`. Similarly, the typing of the `adapts` clause ensures that the free variables of the `classifies`- and `retrieves`-formulas

$$\begin{array}{c}
\textbf{Knowledge base typing} \quad \boxed{\vdash \mathcal{K} \text{ OK}} \\
\text{(T-knowledge)} \quad \frac{\text{FV}(\phi) = \emptyset \text{ for all } \phi \in \mathcal{K}}{\vdash \mathcal{K} \text{ OK}} \\
\\
\textbf{Expression Typing} \quad \boxed{\Gamma \vdash e : T} \\
\text{(T-var)} \quad \Gamma \vdash x : \Gamma(x) \quad \text{(T-int)} \quad \Gamma \vdash n : \text{Int} \\
\text{(T-null)} \quad \Gamma \vdash \text{null} : \perp \\
\text{(T-field)} \quad \frac{\Gamma \vdash e : C \quad \text{aType}(C)(f) = T \quad (e = \text{this}) \vee \neg \text{Rcl}(C)}{\Gamma \vdash e.f : T} \\
\text{(T-invoke)} \quad \frac{\Gamma \vdash e_0 : C_0 \quad (e_0 = \text{this}) \vee \neg \text{Rcl}(C_0) \quad \Gamma \vdash \bar{e} : \bar{S} \quad \text{aType}(m, C_0) = \bar{T} \rightarrow T \quad \bar{S} \leq \bar{T}}{\Gamma \vdash e_0.m(\bar{e}) : T} \\
\text{(T-new)} \quad \frac{\neg \text{Adp}(C) \quad \bar{T}\bar{f} = \text{fields}(C) \quad \Gamma \vdash \bar{e} : \bar{S} \quad \bar{S} \leq \bar{T} \quad D = \begin{cases} \text{Prg}(C)(\text{extends}) & \text{if } \text{Rcl}(C) \\ C & \text{otherwise} \end{cases}}{\Gamma \vdash \text{new } C(\bar{e}) : D} \\
\text{(T-assign)} \quad \frac{\Gamma \vdash e_0.f : T \quad \Gamma \vdash e_1 : S \quad S \leq T}{\Gamma \vdash e_0.f = e_1 : T} \\
\text{(T-adapt)} \quad \frac{\Gamma \vdash e : C \quad \text{Adp}(C)}{\Gamma \vdash \text{adapt}(e) : C} \\
\\
\textbf{Links typing} \quad \boxed{\vdash \text{links OK}} \\
\text{(T-links)} \quad \frac{\text{FV}(\phi) = \{z\}}{\vdash \text{links } \lambda z.\phi \text{ OK}} \\
\\
\textbf{Adapts typing} \quad \boxed{\text{this} : C \vdash \text{adapts OK}} \\
\text{(T-adapts)} \quad \frac{[\text{fields}(C) = T_1 f_1, \dots, T_n f_n] \quad \text{FV}(\phi) = \{z\} \quad [\text{FV}(\psi) = \{z, z_1, \dots, z_n\}]}{\text{this} : C \vdash \text{classifies } \lambda z.\phi \quad [\text{retrieves } \lambda z z.\psi] \text{ OK}} \\
\\
\textbf{Method definition typing} \quad \boxed{\text{this} : C \vdash \text{MD OK}} \\
\text{(T-method)} \quad \frac{\text{this} : C, \bar{x} : \bar{T} \vdash e : S \quad S \leq T}{\text{this} : C \vdash T.m(\bar{T}\bar{x})\{\text{return } e;\} \text{ OK}} \\
\\
\textbf{Class definition typing} \quad \boxed{\vdash \text{CD OK}} \\
\text{(T-class)} \quad \frac{\text{this} : C \vdash \bar{\text{MD}} \text{ OK} \quad [\vdash \text{links OK}] \quad [\text{this} : C \vdash \text{adapts OK}]}{\vdash \text{class } C[\text{extends } D][\text{links}][\text{adapts}]\{\bar{\text{FD}} \bar{\text{MD}}\} \text{ OK}} \\
\\
\textbf{Program typing} \quad \boxed{\vdash \text{Prg OK}} \\
\text{(T-program)} \quad \frac{\text{Prg} = \mathcal{K} \bar{\text{CD}} e \quad \text{sane}(\text{Prg}) \quad \vdash \mathcal{K} \text{ OK} \quad \vdash \bar{\text{CD}} \text{ OK} \quad \bullet \vdash e : S}{\vdash \text{Prg OK}} \\
\\
\textbf{Field lookup} \\
\frac{[\text{fields}(D) = \bar{T}' \bar{f}'] \quad \text{Prg}(C) = \text{class } C[\text{extends } D] \dots \{\bar{T} \bar{f}; \bar{\text{MD}}\}}{\text{fields}(C) = [\bar{T}' \bar{f}',] \bar{T} \bar{f}}
\end{array}$$

■ **Figure 6** FSRJ: typing rules and field lookup rule.

are known. The typing rules for method and class definitions and for programs are standard recursively checking the different syntactic elements; the latter rule additionally checks that the program is sane. In the type system, the auxiliary function `fields(C)` retrieves all field definitions of a given class `C` in the program `Prg`, including the inherited ones. To type the `null` value, we use the special type \perp , which is not a class name, not part of the program and a subtype of any class name. In the sequel, we use the following convention:

- the metavariable T denotes either a non-reclassifiable class name or `int`; and
- the metavariable S denotes either a non-reclassifiable class name, `int` or \perp .

We say that a program `Prg` is well-typed to mean that $\vdash \text{Prg OK}$ holds.

► **Lemma 6** (Syntactic guarantees for well-typed programs). *Let $\vdash \text{Prg OK}$. Then $\text{sane}(\text{Prg})$ and if `new C(...)` appears in `Prg` then `C` is not adaptable.*

3.4 Code, Linkage and Program Knowledge Bases

The purpose of this section is to explain how to formalise program-specific knowledge bases in FOL. Given a well-typed program $\text{Prg} = \mathcal{K} \bar{\text{CD}} e$ with domain knowledge base \mathcal{K} , we first define a knowledge base $\mathcal{K}^{\text{code}}$ that reflects part of the code of `Prg` in FOL. The knowledge base $\mathcal{K}^{\text{code}}$ includes general axioms and symbols common to all FSRJ programs, as well as knowledge specifically derived from `Prg`, such as knowledge about declared classes and fields.

- A1. $\forall x. (x \doteq \text{int}) \oplus (x \doteq \text{null}) \oplus \text{isInt}(x) \oplus \text{isObj}(x) \oplus \text{isCls}(x),$
- A2. $\forall x. \text{isCls}(x) \Leftrightarrow (\text{isStd}(x) \oplus \text{isAdp}(x) \oplus \text{isRcl}(x))$
- A3. $\text{isInt}(n)$ for all $n \in \mathbb{Z}_k$
- A4. $\text{isCls}(c)$ for all $c \in Cls$
- A5. $\text{isStd}(c)$ for all $c \in Cls$ such that $\text{Std}(c)$
- A6. $\text{isAdp}(c)$ for all $c \in Cls$ such that $\text{Adp}(c)$
- A7. $\text{isRcl}(c)$ for all $c \in Cls$ such that $\text{Rcl}(c)$
- A8. $\forall x. (\bigwedge_{n \in \mathbb{Z}_k} \neg(x \doteq n)) \Rightarrow \neg \text{isInt}(x)$
- A9. $\forall x. (\bigwedge_{c \in Cls} \neg(x \doteq c)) \Rightarrow \neg \text{isCls}(x)$
- A10. $\forall x, y. \text{subclass}(x, y) \Rightarrow (\text{isCls}(x) \wedge \text{isCls}(y))$
- A11. $\text{subclass}(c, c')$ for all $c, c' \in Cls$ such that $c <: c'$
- A12. $\neg \text{subclass}(c, c')$ for all $c, c' \in Cls$ such that $c \not<: c'$
- A13. $\forall x, y. \text{compatible}(x, y) \Leftrightarrow \exists z. \text{subclass}(x, z) \wedge \text{subclass}(y, z) \wedge \text{isAdp}(z)$
- A14. $\forall x, y. (\text{instOf}(x, y) \vee \text{in}(x, y)) \Rightarrow (\text{isObj}(x) \wedge \text{isCls}(y))$
- A15. $\forall x. \text{isObj}(x) \Rightarrow \exists y. \text{isCls}(y) \wedge \text{instOf}(x, y) \wedge \neg \text{isAdp}(y)$
- A16. $\forall x, y, z. (\text{instOf}(x, y) \wedge \text{instOf}(x, z)) \Rightarrow (y \doteq z)$
- A17. $\forall x, y. \text{in}(x, y) \Leftrightarrow \exists z. \text{instOf}(x, z) \wedge \text{subclass}(z, y)$
- A18. $\forall x, y. \text{hasType}(x, y) \Leftrightarrow (\text{in}(x, y) \vee (\text{isInt}(x) \wedge (y \doteq \text{int})) \vee ((x \doteq \text{null}) \wedge \text{isCls}(y)))$
- A19. $\forall x. \text{in}(x, c) \Leftrightarrow \exists y. c_f(x, y)$ for all $c \in Cls$ and $c_f \in Fls(c)$
- A20. $\forall x, y, z. (c_f(x, y) \wedge c_f(x, z)) \Rightarrow (y \doteq z)$ for all $c \in Cls$ and $c_f \in Fls(c)$
- A21. $\forall x, y. c_f(x, y) \Rightarrow \text{in}(x, c) \wedge \text{hasType}(y, \text{type}(c_f))$ for all $c \in Cls$ and $c_f \in Fls(c)$

■ **Figure 7** The code knowledge base.

► **Definition 7** (Code knowledge bases). *Given a well-typed FSRJ program Prg with domain knowledge base \mathcal{K} over the signature Σ , let $Cls = \text{dom}(\text{Prg})$ be the set of class names in Prg , $Fls(c)$ the set of qualified field names $c.f$ declared (not inherited) in class c , $\text{type}(c.f)$ the type of the field $c.f$, and $\mathbb{Z}_k = \{z \mid -k \leq z \leq k\}$ (where k is natural number) a set of integers. We denote by Σ^{code} the following signature (where subscripts denote the arity of non-constant symbols):*

$$\begin{aligned} & \mathbb{Z}_k \cup \{\text{int}, \text{null}\} \cup Cls \cup \{c_f_2 \mid c.f \in Fls(c)\} \\ & \cup \{\text{isInt}_1, \text{isObj}_1, \text{isCls}_1, \text{isStd}_1, \text{isAdp}_1, \text{isRcl}_1, \text{subclass}_2, \text{in}_2, \text{compatible}_2, \text{instOf}_2, \text{hasType}_2\} \end{aligned}$$

and (without loss of generality) assume that $\Sigma \cap \Sigma^{\text{code}} = \mathbb{Z}_k \cup \{\doteq\}$. The code knowledge base $\mathcal{K}^{\text{code}}$ of Prg is the set consisting of the formulas over Σ^{code} in Figure 7.

We briefly discuss the formulas of $\mathcal{K}^{\text{code}}$. Axiom A1 expresses that each element in the domain must be either the type name `int`, the value `null`, an integer value, an object or a class; and A2 that each class is either standard, adaptable or reclassifiable. Axioms A3 declare that each constant $n \in \mathbb{Z}_k$ is an integer and A4 declare that each constant $c \in Cls$ is a class. Axioms A5–A7 declare the kind (standard, adaptable, reclassifiable) of each constant $c \in Cls$. Axioms A8 express that each integer in the domain is mapped to a constant $n \in \mathbb{Z}_k$ and A9 that each class in the domain is mapped to a constant $c \in Cls$. Axioms A10–A12 express that the `subclass` relation relates classes to classes and capture the `<:` relation of Prg . Axiom A13 expresses that two (reclassifiable) classes are `compatible` if and only if they are both subclasses of the same adaptable class. Axiom A14 expresses that both the `instOf` and the `in` relations relate objects with classes. Axioms A15 and A16 declare that each object is an `instOf` of exactly one non-adaptable class. Axiom A17 declares that an object is in a class if and only if it is an `instOf` of a subclass of that class. Axiom A18 declares that only `null`,

integers and objects have types, and which are their types. Axioms A19–A21 express that the predicate $\mathbf{c_f}$ models the field \mathbf{f} of objects of any subclass of \mathbf{c} , each object has exactly the fields (declared or inherited) in its class, and each field contains exactly one value of the declared type.

► **Example 8** (Code knowledge base for the Greenhouse digital twin in FSRJ). A subset of the code knowledge base for the FSRJ program in Example 1 (namely, the subset containing axioms A4 and A11) is given in Figure 2c.

We define the linkage knowledge base to express that each object satisfies the predicate specified by the **links** clause of its class.

► **Definition 9** (Linkage knowledge bases). *Let Prg be a well-typed FSRJ program with domain knowledge base \mathcal{K} over Σ . Let $\text{Cls} = \text{dom}(\text{Prg})$. The linkage knowledge base $\mathcal{K}^{\text{link}}$ of Prg is the set of formulas over the signature $\Sigma \cup \Sigma^{\text{code}}$ defined by $\forall x. \text{instOf}(x, \mathbf{c}) \Rightarrow \phi[z := x]$ for all $\mathbf{c} \in \text{Cls}$ and $\text{Prg}(\mathbf{c})(\mathbf{links}) = \lambda z. \phi$.*

► **Example 10** (Linkage knowledge base for the Greenhouse digital twin in FSRJ). The linkage knowledge base for the FSRJ program of Example 1 is given in Figure 3b.

Observe that by combining Σ and Σ^{code} , predicates in $\mathcal{K}^{\text{link}}$ may express connections between an FSRJ program’s runtime values (e.g., instances of a class \mathbf{c}) and the domain knowledge base. We define a knowledge base that comprises the three knowledge bases defined above:

► **Definition 11** (Program knowledge bases). *Let Prg be a well-typed FSRJ program with domain knowledge base \mathcal{K} over the signature Σ . The program knowledge base of Prg is the knowledge base $\mathcal{K}^{\text{Prg}} = \mathcal{K} \cup \mathcal{K}^{\text{code}} \cup \mathcal{K}^{\text{link}}$ over the signature $\Sigma \cup \Sigma^{\text{code}}$.*

► **Lemma 12** (Satisfiability of the program knowledge base). *For every well-typed program Prg with domain knowledge base \mathcal{K} over the signature Σ . If \mathcal{K} is satisfiable, then the knowledge base $\mathcal{K}^{\text{Prg}} = \mathcal{K} \cup \mathcal{K}^{\text{code}} \cup \mathcal{K}^{\text{link}}$ over the signature $\Sigma \cup \Sigma^{\text{code}}$ is satisfiable.*

3.5 Program Coherence

This section considers requirements on FSRJ programs Prg to ensure that reasoning with the program knowledge base \mathcal{K}^{Prg} is in fact meaningful for declarative object reclassification. These requirements are needed to ensure type soundness; even with a satisfiable domain knowledge base, linking axioms can make the knowledge base incoherent (e.g., define a linkage predicate to introduce false into the knowledge base). Below, let $\lambda z. \phi^{\mathbf{c}} = \text{Prg}(\mathbf{c})(\mathbf{classifies})$ and $\text{Children}(\mathbf{c}) = \{\mathbf{d} \in \text{dom}(\text{Prg}) \mid \text{Prg}(\mathbf{d})(\mathbf{extends}) = \mathbf{c}\}$ for classes \mathbf{c} . We say that a well-typed program Prg with domain knowledge base \mathcal{K} over a signature Σ is *coherent* if:

Coh1: \mathcal{K} is satisfiable. This requirement ensures that models exist for the domain knowledge base.

Coh2: $\mathcal{K} \cup \mathcal{K}^{\text{code}} \models \mathcal{K}^{\text{links}}$. This requirement ensures that the user-defined linkage axioms exclude no interpretation that satisfies the union of the domain and code knowledge bases $\mathcal{K} \cup \mathcal{K}^{\text{code}}$; thus ensuring that each object satisfies the linkage predicate of its class.

Coh3: In every model satisfying \mathcal{K}^{Prg} , for every object x instance of a reclassifiable subclass of an adaptable class \mathbf{c} , the classification predicate $\phi^{\mathbf{c}}$ of \mathbf{c} must hold for x :

$$\mathcal{K}^{\text{Prg}} \models \forall x. \text{in}(x, \mathbf{c}) \Rightarrow \phi^{\mathbf{c}}[z := x], \text{ for all } \mathbf{c} \in \text{Cls} \text{ such that } \text{isAdp}(\mathbf{c}).$$

This requirement ensures that the classification-predicates in the knowledge base respect the subclass hierarchy in the program.

Coh4: In every model satisfying \mathcal{K}^{Prg} , for every object x in an adaptable class \mathbf{c} , classification predicate $\phi^{\mathbf{D}}$ of one of its subclasses \mathbf{D} must hold for x :

$$\mathcal{K}^{\text{Prg}} \models \forall x. \text{in}(x, \mathbf{c}) \Rightarrow \bigvee_{\mathbf{D} \in \text{Children}(\mathbf{c})} \phi^{\mathbf{D}}[z := x], \text{ for all } \mathbf{c} \in \text{Cls s.t. isAdp}(\mathbf{c}).$$

This requirement ensures that for every instance of a subclass of \mathbf{c} , the classification query of at least one of the subclasses of \mathbf{c} always holds.

Coh5: In every model satisfying \mathcal{K}^{Prg} , for every object x in an adaptable class \mathbf{c} and subclass \mathbf{D} of \mathbf{c} , the **retrieves** predicate of \mathbf{D} must hold for x when its **classifies** predicate holds:

$$\mathcal{K}^{\text{Prg}} \models \forall x. (\text{in}(x, \mathbf{c}) \wedge \phi^{\mathbf{D}}[z := x]) \Rightarrow ((\exists \bar{y}. \psi^{\mathbf{D}}[z \bar{z} := x \bar{y}]) \wedge (\forall \bar{y}. \psi^{\mathbf{D}}[z \bar{z} := x \bar{y}] \Rightarrow \text{hasType}(\bar{y}, \bar{\mathbf{T}}))),$$

for all $\mathbf{c}, \mathbf{D} \in \text{Cls}$ such that $\text{isAdp}(\mathbf{c})$ and $\text{subclass}(\mathbf{D}, \mathbf{c})$

where $\lambda z, \bar{z}. \psi^{\mathbf{D}} = \text{Prg}(\mathbf{D})(\text{retrieves})$, $\text{fields}(\mathbf{D}) = \bar{\mathbf{T}} \bar{\mathbf{f}}$ (for some $\bar{\mathbf{f}}$), and $\text{hasType}(\bar{y}, \bar{\mathbf{T}})$ is shorthand for $\text{hasType}(y_1, \tau_1) \wedge \dots \wedge \text{hasType}(y_n, \tau_n)$. This requirement ensures that a reclassified object can be instantiated correctly.

Conditions **Coh1** and **Coh2** are basic requirements on the domain knowledge base \mathcal{K} and the linkage axioms. They ensure that reasoning over the combined knowledge base is possible. In contrast, **Coh3–Coh5** are required for type soundness (cf. Theorem 22): first **Coh3** ensures that subclassing in the program and the knowledge base are aligned; then **Coh4** ensures that reclassification always succeeds in finding a target class \mathbf{D} ; and finally **Coh5** ensures that values for the fields in the target class \mathbf{D} can always be retrieved in the knowledge base and respect the declared types of the fields of class \mathbf{D} in the program.

3.6 Heap Lifting, Synchronisation and Operational Semantics

To model the operational semantics of FSRJ, we introduce the concepts of *address* (of an object in a heap), *stack*, *values*, *object*, *heap*, *lifted-heap knowledge base*, *environment*, *synchronisation knowledge base*, *overall knowledge base*, *runtime expressions* and *runtime configurations*. *Addresses*, ranged over by the metavariable ι , are elements of the denumerable set I . *Stacks* are sequences $\bar{\iota}$, recording the addresses of objects on which methods under execution have been invoked (the address of the object of the most recent invocation is rightmost). *Values*, ranged over by the metavariable v , are either addresses, integers, or the special value `null`. *Objects* are tuples $\langle \mathbf{C}, \bar{\mathbf{f}} = \bar{v} \rangle$, where \mathbf{C} is the class of the object, $\bar{\mathbf{f}}$ the names and \bar{v} the values of the fields. A heap \mathcal{H} maps addresses to objects; the empty heap is denoted by \emptyset . A value v occurs in \mathcal{H} if $v \in \{\iota, \bar{v}\}$ for some $\mathcal{H}(\iota) = \langle \mathbf{C}, \bar{\mathbf{f}} = \bar{v} \rangle$, where $\iota \in \text{dom}(\mathcal{H})$.

► **Definition 13** (Heaps for well-typed programs). *A heap \mathcal{H} for a well-typed FSRJ program Prg is a mapping from addresses to objects such that*

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

The following definition formalises the concept of *lifting a heap* into a knowledge base.

► **Definition 14** (Lifted-heap knowledge base). *Given a heap \mathcal{H} for a well-typed FSRJ program Prg , the lifted-heap knowledge base $\mathcal{K}^{\text{heap}}(\mathcal{H})$, which lifts \mathcal{H} to FOL, is the set consisting of the following formulas over the signature $\Sigma^{\text{heap}}(\mathcal{H}) = \Sigma^{\text{code}} \cup \text{dom}(\mathcal{H})$:*

- L1. $\text{instOf}(\iota, \mathbf{c})$ for all $\iota \in \text{dom}(\mathcal{H})$ such that $\mathcal{H}(\iota) = \langle \mathbf{c}, \mathbf{f}_1 = v_1, \dots, \mathbf{f}_n = v_n \rangle$
- L2. $\mathbf{C}_j\text{-}\mathbf{f}_j(\iota, v_j)$ for all $\iota \in \text{dom}(\mathcal{H})$ and for all $\mathbf{C}_j\text{-}\mathbf{f}_j$ such that $\mathcal{H}(\iota) = \langle \mathbf{c}, \mathbf{f}_1 = v_1, \dots, \mathbf{f}_n = v_n \rangle$, $\mathbf{c} <: \mathbf{C}_j$, $\mathbf{C}_j.\mathbf{f}_j \in \text{Fls}(\mathbf{C}_j)$ and $1 \leq j \leq n$
- L3. $\forall x. (\bigwedge_{\iota \in \text{dom}(\mathcal{H})} (x \neq \iota)) \Rightarrow \neg \text{isObj}(x)$

The following definition formalises the concept of stating the values of the current properties of the physical assets into a knowledge base.

► **Definition 15** (Environment and synchronisation knowledge base). *An environment \mathcal{E} for a well-typed FSRJ program Prg is a reference to a mapping that provides suitable current values for the properties of the assets described in the domain knowledge base of Prg . The synchronisation knowledge base $\mathcal{K}^{\text{sync}}(\mathcal{E})$ of Prg provides a witness for each formula $\exists x.\text{ctx_prop}(\text{ctx_asset}, x)$ in \mathcal{K} which states that the property ctx_prop of the asset ctx_asset has some value x .*

We now define a knowledge base that combines the two knowledge bases defined above, which model the dynamic part of the system (the program heap and the current values of the properties physical assets, respectively) with the domain and code knowledge base, which model the static part of the system (domain knowledge and the structure of the code, respectively). The combined knowledge base is operationalised in the operational semantics of FSRJ (see rules R-adapt-y and R-adapt-n in Figure 8).

► **Definition 16** (Overall knowledge base). *Given a heap \mathcal{H} and an environment \mathcal{E} for a well-typed FSRJ program Prg , the overall knowledge base $\mathcal{K}^{\text{overall}}(\mathcal{H}, \mathcal{E}) = \mathcal{K}^{\text{Prg}} \cup \mathcal{K}^{\text{heap}}(\mathcal{H}) \cup \mathcal{K}^{\text{sync}}(\mathcal{E})$ of Prg , over the signature $\Sigma^{\text{Prg}} \cup \Sigma^{\text{heap}}(\mathcal{H})$, represents domain knowledge, code, heap and value of the asset properties for the program Prg with heap \mathcal{H} and environment \mathcal{E} .*

We briefly discuss the formulas of $\mathcal{K}^{\text{heap}}(\mathcal{H})$. The axiom *L1* introduces individuals ι as instances of classes \mathbf{c} (Axiom *A14* of Definition 7 then allows us to infer $\text{isObj}(\iota)$). Axiom *L2* populates the relation $\mathbf{c_f}$ for all fields \mathbf{f} of the classes \mathbf{c} . Axiom *L3* states that only addresses from the heap can be considered as objects in the knowledge base $\mathcal{K}^{\text{overall}}(\mathcal{H}, \mathcal{E})$ (thus, *L3* is similar to the closure axioms *A8* and *A9* for integers and classes in Figure 7).

► **Example 17** (Lifted-heap and synchronisation knowledge bases for the Greenhouse digital twin in FSRJ). The lifted-heap and synchronisation knowledge bases for the FSRJ program in Figure 3a and Example 1 after the first execution of the `pu.plant=this.pl` expression in the body of the `loop` method, are given in Figures 2b and 2d, respectively.

Recall that (by Lemma 12) the program knowledge base \mathcal{K}^{Prg} of a well-typed program Prg is satisfiable. The following theorem shows that, if \mathcal{K}^{Prg} satisfies conditions *Coh1* and *Coh2* for program coherence (see Section 3.5), then its extension $\mathcal{K}^{\text{overall}}(\mathcal{H}, \mathcal{E})$ with a lifted-heap knowledge base $\mathcal{K}^{\text{heap}}(\mathcal{H})$ and synchronization knowledge base $\mathcal{K}^{\text{sync}}(\mathcal{E})$ for Prg remains satisfiable (thus, reasoning over $\mathcal{K}^{\text{overall}}(\mathcal{H}, \mathcal{E})$ is possible).

► **Theorem 18** (Satisfiability of the overall knowledge base). *Let \mathcal{H} be a heap and \mathcal{E} an environment for the well-typed program Prg with domain knowledge base \mathcal{K} is over the signature Σ , and let $\mathcal{K}^{\text{Prg}} = \mathcal{K} \cup \mathcal{K}^{\text{code}} \cup \mathcal{K}^{\text{link}}$ be the program knowledge base of Prg . If \mathcal{K} is satisfiable and $\mathcal{K} \cup \mathcal{K}^{\text{code}} \models \mathcal{K}^{\text{links}}$, then the overall knowledge base $\mathcal{K}^{\text{overall}}(\mathcal{H}, \mathcal{E})$ is satisfiable.*

Runtime expressions are ranged over by e . Their syntax is obtained from the syntax of expressions (cf. Figure 5) by removing variables \mathbf{x} (recall that `this` is a variable) and adding addresses ι and `ret`-expressions (of the form `ret(e)`, that represent the execution of a method). A *runtime configuration* (*configuration*, for short) is a triple consisting of a heap, a stack and a runtime expression, i.e., $\mathcal{H} \mid \bar{\iota} \mid e$. The reduction relation has the form $\mathcal{H} \mid \bar{\iota} \mid e \rightarrow \mathcal{H}' \mid \bar{\iota}' \mid e'$, where the configuration $\mathcal{H} \mid \bar{\iota} \mid e$ gets reduced to $\mathcal{H}' \mid \bar{\iota}' \mid e'$ in one step. A configuration $\mathcal{H} \mid \bar{\iota} \mid e$ that cannot be reduced is called a *normal form*. The initial configuration of a program is $\emptyset \mid \bullet \mid \mathbf{e}$, where \mathbf{e} is the main expression of the program. Remark that configurations only

Computation rules		
	(R-ret) $\mathcal{H} \mid \bar{\iota} \mid \text{ret}(v) \rightarrow \mathcal{H} \mid \bar{\iota} \mid v$	
(R-field)	$\frac{\mathbf{f} = v \in \bar{\mathbf{f}} = \bar{v} \quad \mathcal{H}(\iota) = \langle \mathbf{C}, \bar{\mathbf{f}} = \bar{v} \rangle}{\mathcal{H} \mid \bar{\iota} \mid \iota.\mathbf{f} \rightarrow \mathcal{H} \mid \bar{\iota} \mid v}$	(R-new)
		$\frac{\iota \notin \text{dom}(\mathcal{H}) \quad \text{fields}(\mathbf{C}) = \bar{\mathbf{T}} \bar{\mathbf{f}}}{\mathcal{H} \mid \bar{\iota} \mid \text{new } \mathbf{C}(\bar{v}) \rightarrow \mathcal{H} \cup \left\{ \iota \mapsto \langle \mathbf{C}, \bar{\mathbf{f}} = \bar{v} \rangle \right\} \mid \bar{\iota} \mid \iota}$
(R-invoke)	$\frac{\mathcal{H}(\iota) = \langle \mathbf{C}, \dots \rangle \quad \text{mBody}(\mathbf{m}, \mathbf{C}) = (\bar{\mathbf{x}}, \mathbf{e}_0)}{\mathcal{H} \mid \bar{\iota} \mid \iota.\mathbf{m}(\bar{v}) \rightarrow \mathcal{H} \mid \bar{\iota} \mid \text{ret}([\bar{\mathbf{x}} \leftarrow \bar{v}, \text{this} \leftarrow \iota] \mathbf{e}_0)}$	(R-assign)
		$\frac{\mathcal{H}(\iota) = \langle \mathbf{C}, \dots, \mathbf{f}_i = v_i, \dots \rangle}{\mathcal{H} \mid \bar{\iota} \mid \iota.\mathbf{f}_i = v \rightarrow \mathcal{H}[\iota \mapsto \langle \mathbf{C}, \dots, \mathbf{f}_i = v, \dots \rangle] \mid \bar{\iota} \mid v}$
(R-adapt-y)	$\frac{\begin{array}{l} \iota \notin \bar{\iota} \quad \mathbf{D} \neq \mathbf{D}' \quad \mathcal{H}(\iota) = \langle \mathbf{D}, \bar{\mathbf{f}} = \bar{v} \rangle \\ \text{classifies}(\mathbf{D}) = \lambda z.\phi \quad \mathcal{K}^{\text{code}} \models \text{compatible}(\mathbf{D}, \mathbf{D}') \\ \text{classifies}(\mathbf{D}') = \lambda z.\phi' \quad \mathcal{K}^{\text{overall}}(\mathcal{H}, \mathcal{E}) \models \phi[x := \iota] \\ \text{retrieves}(\mathbf{D}') = \lambda z\bar{z}.\psi' \quad \mathcal{K}^{\text{overall}}(\mathcal{H}, \mathcal{E}) \models \psi'[z\bar{z} := \iota\bar{v}'] \end{array}}{\mathcal{H} \mid \bar{\iota} \mid \text{adapt}(\iota) \rightarrow \mathcal{H}[\iota \mapsto \langle \mathbf{D}', \bar{\mathbf{f}}' = \bar{v}' \rangle] \mid \bar{\iota} \mid \iota}$	(R-adapt-n)
		$\frac{\begin{array}{l} \iota \notin \bar{\iota} \\ \mathcal{H}(\iota) = \langle \mathbf{D}, \bar{\mathbf{f}} = \bar{v} \rangle \\ \text{classifies}(\mathbf{D}) = \lambda z.\phi \\ \mathcal{K}^{\text{overall}}(\mathcal{H}, \mathcal{E}) \models \phi[x := \iota] \end{array}}{\mathcal{H} \mid \bar{\iota} \mid \text{adapt}(\iota) \rightarrow \mathcal{H} \mid \bar{\iota} \mid \iota}$
Method body lookup	Evaluation context	Congruence rule
$\frac{\text{aDef}(\mathbf{C})(\mathbf{m}) = \mathbf{T} \ \mathbf{m}(\bar{\mathbf{T}} \ \bar{\mathbf{x}}) \ \{\text{return } \mathbf{e};\}}{\text{mBody}(\mathbf{m}, \mathbf{C}) = (\bar{\mathbf{x}}, \mathbf{e})}$	$E ::= [] \mid E.\mathbf{f} \mid E.\mathbf{m}(\bar{e}) \mid \mathbf{v}.\mathbf{m}(\bar{v}, E.\bar{e}) \mid \text{new } \mathbf{C}(\bar{v}, E, \bar{e}) \mid \mathbf{v}.\mathbf{f} = E \mid \text{ret}(E) \mid \text{adapt}(E)$	$\frac{\mathcal{H} \mid \bar{\iota} \mid e \rightarrow \mathcal{H}' \mid \bar{\iota}' \mid e'}{\mathcal{H} \mid \bar{\iota} \mid E[e] \rightarrow \mathcal{H}' \mid \bar{\iota}' \mid E[e']}$

■ **Figure 8** FSRJ operational semantics: computation rules, method body lookup, evaluation context, and congruence rule.

include stacks to ensure that the operational semantics can model that execution gets stuck when the reclassification of an object on which a method under execution has been invoked is attempted (i.e., that a configuration that is going to perform such a reclassification is a normal form).

The reduction system for FSRJ is given in Figure 8. The reduction rules use the auxiliary function $\text{mBody}(\mathbf{m}, \mathbf{C})$ (for method body lookup) and the *congruence rule* (which uses evaluation contexts) are also given in Figure 8. The rules R-field, R-new and R-assign for field access, object creation and assignment are standard. Rule R-invoke pushes the address of the called object to the stack and a **ret**-expression that stems from method lookup for the called method in the class of the called object, as the new runtime expression. Here, actual parameters and the self-reference are inlined by substitution on the runtime expression. Rule R-ret then pops the object address from the stack, returning the computed value as the new runtime expression.

In FSRJ an object is hot (see Section 2) if and only if its address occurs on the stack. Both rules for dynamic object reclassification, R-adapt-y and R-adapt-n, check whether an object needs to be reclassified and cause the execution to get stuck if an attempt to reclassify a hot object is made. Rule R-adapt-y applies if the object needs to be reclassified (i.e., it cannot be classified as a member of its current class); rule R-adapt-n applies otherwise. We now explain the interaction between execution and the knowledge base in these rules. Both rules check whether the object ι of class \mathbf{D} is cold and, if so, lift the heap \mathcal{H} to construct the knowledge base $\mathcal{K}^{\text{overall}}(\mathcal{H}, \mathcal{E})$. If the classification predicate ϕ of \mathbf{D} holds when instantiated with ι , then rule R-adapt-n is applied and the object is not reclassified. Otherwise rule R-adapt-y is applied. Recall from Axiom A13 of Definition 7 that two classes are compatible if they have the same superclass. Thus, the premises of rule R-adapt-y identify a class \mathbf{D}' that is compatible with \mathbf{D} , for which the classification predicate ϕ' holds when instantiated with ι . The last premise ensures that the **retrieves**-predicate ψ' of \mathbf{D}' holds for values ι, \bar{v}' , and the resulting heap on the right-hand side binds ι to an object of class \mathbf{D}' in which the fields $\bar{\mathbf{f}}$ are bound to \bar{v}' .

$$\begin{array}{c}
\textbf{Runtime expression typing } \boxed{\Theta \Vdash e : S} \\
\\
\begin{array}{lll}
(\text{RT-addr}) \Theta \Vdash \iota : \Theta(\iota) & (\text{RT-int}) \Theta \Vdash n : \text{Int} & (\text{RT-null}) \Theta \Vdash \text{null} : S \text{ (with } S \in \{\perp\} \cup \text{Cls}\text{)} \\
\\
(\text{RT-field}) \frac{\Theta \Vdash e : C \quad \text{aType}(C)(f) = T}{\Theta \Vdash e.f : T} & & (\text{RT-ret}) \frac{\Theta \Vdash e : T}{\Theta \Vdash \text{ret}(e) : T} \\
\\
(\text{RT-assign}) \frac{\Theta \Vdash e_0.f : T \quad \Theta \Vdash e_1 : S \quad S \leq T}{\Theta \Vdash e_0.f = e_1 : T} & & (\text{RT-invoke}) \frac{\Theta \Vdash e : C \quad \bar{S} \leq \bar{T} \quad \Theta \Vdash \bar{e} : \bar{S} \quad \text{aType}(C)(m) = \bar{T} \rightarrow T}{\Theta \Vdash e.m(\bar{e}) : T} \\
\\
(\text{RT-adapt}) \frac{\Theta \Vdash e : D \quad D \leq C \quad \text{Adp}(C)}{\Theta \Vdash \text{adapt}(e) : C} & & (\text{RT-new}) \frac{\neg \text{Adp}(C) \quad \bar{S} \leq \bar{T} \quad \Theta \Vdash \bar{e} : \bar{S} \quad \bar{T}\bar{f} = \text{fields}(C) \quad \begin{array}{l} D = \begin{cases} \text{Prg}(C)(\text{extends}) & \text{if } \text{Rcl}(C) \\ C & \text{otherwise} \end{cases} \end{array}}{\Theta \Vdash \text{new } C(\bar{e}) : D}
\end{array} \\
\\
\textbf{Well-formed heap } \boxed{\Theta \Vdash \mathcal{H}} \\
\\
\begin{array}{c}
\text{dom}(\Theta) = \text{dom}(\mathcal{H}) = \{\iota \mid \iota \text{ occurs in } \mathcal{H}\} \\
\forall \iota \in \text{dom}(\mathcal{H}). \mathcal{H}(\iota) = \langle C, f_1 = v_1, \dots, f_n = v_n \rangle \quad \text{implies} \\
\left(\begin{array}{l} \Theta(\iota) = C \\ \text{fields}(C) = T_1 f_1, \dots, T_n f_n \\ \forall i \in 1..n, \quad \Theta \Vdash v_i : S_i \quad \text{and} \quad S_i <: T_i \end{array} \right)
\end{array} \\
\\
(\text{WF-heap}) \frac{}{\Theta \Vdash \mathcal{H}} \\
\\
\textbf{Well-formed configuration } \boxed{\Theta \Vdash \mathcal{H} \mid \bar{\iota} \mid e : S} \\
\\
(\text{WF-conf}) \frac{e = E_1[\text{ret}(\dots E_n[\text{ret}(e_0)] \dots)] \text{ for some } E_1, \dots, E_n \text{ and } e_0 \text{ that do not contain } \text{ret-expressions} \quad \Theta \Vdash \mathcal{H} \quad \bar{\iota} = \iota_1, \dots, \iota_n \in \text{dom}(\Theta) \quad n \geq 0}{\Theta \Vdash \mathcal{H} \mid \bar{\iota} \mid e : S}
\end{array}$$

■ **Figure 9** FSRJ run-time typing: runtime expressions, well-formed heaps and configurations.

3.7 Type Soundness for Coherent Programs

In order to prove type soundness for coherent FSRJ programs by a subject reduction theorem and a progress theorem for the small-step operational semantics, we need to formulate a type system for runtime expressions. Expressions containing either a *stupid selection*, i.e., a field selection `null.f` or a method invocation `null.m(...)`, or a *stupid reclassification*, i.e., an object reclassification `adapt(null)`, are not well-typed according to the FSRJ (source level) type system. However, a runtime expression without stupid selections and stupid reclassifications may reduce to a runtime expression containing stupid selections or stupid reclassifications. Therefore, the type system for runtime expressions contains a rule for assigning to the value `null` either the type \perp (like in the source level typing) or any class name.

The type rules for runtime expressions are shown in Figure 9 (left); these rules are of the form $\Theta \Vdash e : S$, where the environment Θ is a finite (possibly empty) mapping from addresses to class names. Figure 9 (bottom) also presents the notions of *well-formed-heaps* and *well-formed configurations*. The notion of well-formed heap ensures that the environment Θ maps all the addresses in the heap into the class of the corresponding object and that, for every object stored in the heap, the fields of the object contain appropriate values. The notion of well-formed configuration ensures that the heap is well-formed, that the addresses in the stack are defined in the heap, that the runtime expression is well-typed and has a structure that is compatible with the stack – this last check is performed by exploiting evaluation contexts for FSRJ runtime expressions (see Figure 8).

Then, type soundness can be proved by using the standard technique of subject reduction and progress theorems [50]; however, we need to account for declarative object reclassification.

► **Lemma 19** (From expression typing to runtime expression typing). *If $\bullet \vdash e : T$, then $\bullet \Vdash e : T$.*

We now consider subject reduction for the execution of semantically reflected programs. Here, program coherence (Section 3.5) and Theorem 18 (which assumes **Coh1** and **Coh2** and is independent from **Coh3–Coh5**) are crucial for the case of R-adapt-y:

- an inconsistent (i.e., not satisfiable) knowledge base, or
- a consistent knowledge base that satisfies **Coh1–Coh4** and does not satisfy **Coh5** because the retrieved values for the fields have a wrong type

would allow us to, e.g., infer erroneous values for **retrieves** λ -predicates (see the premise $\mathcal{K}^{\text{overall}}(\mathcal{H}, \mathcal{E}) \models \psi'[z\bar{z} := \iota\bar{v}']$ of rule R-adapt-y), potentially leading to typing errors for runtime configurations after reclassification. Lifting a heap that is not well-typed, could make the knowledge base inconsistent (see Condition L2 in Definition 14 and Axiom A21 in Figure 7). Thus, well-typed runtime configurations ensure consistent overall knowledge bases, which in turn (together with **Coh5**) ensure that (whenever rule R-adapt-y does not get stuck) dynamic object reclassification produces well-typed runtime configurations.

► **Theorem 20** (Subject reduction). *If $\Theta \Vdash \mathcal{H} \mid \bar{\iota} \mid e : S$ and $\mathcal{H} \mid \bar{\iota} \mid e \rightarrow \mathcal{H}' \mid \bar{\iota}' \mid e'$ then there exists $\Theta' \supseteq \Theta$ such that $\Theta' \Vdash \mathcal{H}' \mid \bar{\iota}' \mid e' : S'$ for some S' such that $S' \leq S$.*

Progress relies on program coherence, but not on consistency of the knowledge base. Concretely, **Coh3–Coh5** guarantee that there exists a class D' such that the following three premises of rule R-adapt-y hold:

$$\mathcal{K}^{\text{code}} \models \text{compatible}(D, D') \quad \mathcal{K}^{\text{overall}}(\mathcal{H}, \mathcal{E}) \models \phi'[x := \iota] \quad \mathcal{K}^{\text{overall}}(\mathcal{H}, \mathcal{E}) \models \psi'[z\bar{z} := \iota\bar{v}']$$

The last case in the statement of the theorem states that the adaptation of hot objects gets stuck (cf. **DC3** in Section 4).

► **Theorem 21** (Progress). *Let $\mathcal{H} \mid \bar{\iota} \mid e$ be a well-typed normal form. Then*

1. *either e is a value and $\bar{\iota} = \bullet$; or*
2. *for some evaluation context E we can express e as*
 - a. *either $E[\text{null}.f]$ for some f , or*
 - b. *$E[\text{null}.m(\bar{v})]$ for some m and \bar{v} , or*
 - c. *$E[\text{null}.f = v]$ for some f and v , or*
 - d. *$E[\text{adapt}(\text{null})]$, or*
 - e. *$E[\text{adapt}(\iota)]$ for some $\iota \in \bar{\iota}$.*

Type soundness follows from Lemma 19 and Theorems 20 and 21. After Theorem 20, this theorem is a second place where consistency (needed only for Theorem 20) and program coherence (needed for Theorem 21) interact: We need the overall knowledge base $\mathcal{K}^{\text{overall}}(\mathcal{H}, \mathcal{E})$ to be consistent so that **Coh3–Coh5** indeed select the correct class D' to reclassify.

► **Theorem 22** (Type soundness). *Let $\text{Prg} = \mathcal{K} \overline{\text{CD}} e_0$ be well-typed and coherent FSRJ program such that $\bullet \vdash e_0 : T_0$, and $\emptyset \mid \bullet \mid e_0 \rightarrow^* \mathcal{H} \mid \bar{\iota} \mid e$ with $\mathcal{H} \mid \bar{\iota} \mid e$ being a normal form. Then e is*

1. *either an integer n and T_0 is **int** and $\bar{\iota} = \bullet$; or*
2. ***null** and T_0 is either \perp or a class name and $\bar{\iota} = \bullet$; or*
3. *an address ι such that $\mathcal{H}(\iota) = \langle C, \dots \rangle$ with $C \leq T_0$ and $\bar{\iota} = \bullet$; or*
4. *an expression containing either **null.f** or **null.m**(\bar{v}) or **null.f** = v or **adapt**(**null**) or **adapt**(ι) for some f, m, \bar{v}, v and ι such that $\iota \in \bar{\iota}$.*

4 Discussion: Design Choices, Implementation and Performance

This section discusses design choices for declarative dynamic object reclassification. We first consider overall design choices for the declarative reclassification mechanism (as reflected in FSRJ), then design choices related to the implementation of declarative dynamic object reclassification (as reflected in our prototype implementation), and finally the implementation itself.

4.1 Overall Design Choices

Our main goal with declarative dynamic object reclassification is to realise a separation of concerns and decoupling between the *application logic*, which describes program behaviour within a given context, and the *adaptation logic*, which describes how the behaviour of the program changes according to context changes. To focus on the basic interaction between adaptation and application logic, we aimed for a simple programming construct for dynamic object reclassification and considered the following 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.

Concerning **DC1**, to focus on the general mechanisms of semantic reflection and declarative dynamic object reclassification, FSRJ formalises the knowledge base as a set of FOL formulas. Observe that FSRJ does not impose any restrictions on queries to the knowledge base. For example, `classifies`-predicates may depend on the class of other objects; thus, the reclassification of one object may lead to another object becoming inconsistent. We believe that this flexibility is convenient in a digital twin setting; e.g., a change of requirement monitor for a plant may trigger a change in the controller for the water pump. An alternative design choice would be to restrict queries to, e.g., only depend on the external context (i.e., the domain and synchronisation knowledge). In this case, we avoid the reclassification of one object triggering the reclassification of another, but we could potentially reclassify one object too early, violating the program logic (e.g., changing the watering policy before changing the humidity monitor in the greenhouse). Furthermore, self-adaptation in FSRJ is not deterministic: an object may be reclassifiable to more than one class. If determinism is desired, one can add the additional constraint that the classification queries of its reclassifiable subclasses of all adaptable classes are pairwise disjoint.

Concerning **DC2**, it can be worth observing that (1) the `links`, `classifies` and `retrieves` clauses could be expressed as, e.g., program annotations or even in a separate file; (2) typing just enforces a programming pattern that treats an adaptable class as an abstract class and its reclassifiable subclasses as final classes that cannot be used as types; and (3) the `adapt` expression could be implemented as a library function.

Concerning **DC3**, there are several ways to relax this restriction. First, the reclassification of a hot object could be delayed until the object becomes cold (this raises the issue of whether the ordering between delayed adaptations of different objects should be preserved). Second, the `adapt` expression could be replaced or complemented by a class-level adaptation expression to trigger the adaptation of all objects of a class. Third, the `adapt` expression could be externalised; e.g., replaced or complemented by an adaptation triggered by the runtime system, which performs adaptation on all objects of adaptable classes, and delays adaptation for hot objects until they become cold. Relaxing **DC3** suggests several interesting directions for further work, including

- a type and effect system [3, 57] to ensure that hot object adaptation cannot occur;
- coordinated adaptation of object aggregates (i.e., groups of interconnected objects);
- reclassifiable classes extended by other reclassifiable classes (cf. [15, 20]); and
- adaption of hot objects, e.g., the expression `adapt(this)` (cf. [15, 20]) or a method defined in a class `C` with an expression `adapt(e)` where `e` has type `C` (cf. [16]).

For the last point, the adaptation logic could depend on the values of fields in the target object; when the object is cold, reclassification might rely on these values satisfying some class invariant (following a program verification methodology, e.g., [1, 48]). So, hot object adaptation should require to devise a suitable invariant that the object must satisfy when, e.g., a specific occurrence of the expression `adapt(this)` is executed. For this reason, we decided that even rule R-adapt-n gets stuck on hot objects.

4.2 Implementation and Performance

Implementation. To interact with context in an implementation, we need to be able to check entailment and satisfiability of first-order formulas, and to retrieve witnesses when evaluating the satisfiability of λ -predicates (for `retrieves`-queries). Whereas such functionality is available in, e.g., satisfiability modulo theory solvers, the querying mechanism for declarative dynamic object reclassification would additionally benefit from decidability and availability of formalised domain knowledge. Hence, we consider two possibilities: ontology-based knowledge graphs and logic programming for *DC2*.

Knowledge Graphs are widely used triple-based data formalisms with numerous variants, of which the RDF/OWL stack is well-suited for reclassification. First, these knowledge graphs are used in numerous industrial ontologies and is an established, core technology for autonomous systems, e.g., in digital twins or robotics [7, 42]. Thus, they enable reuse and provide general knowledge across industries, especially in engineering. Second, these knowledge graphs have a foundation in Description Logics [5], which are decidable fragments of FOL with efficient reasoners and advanced tool-supported pragmatics for non-experts [28].

Logic Programming is widely used and perhaps more rooted in programming and artificial intelligence than knowledge graphs, and also used for autonomous systems [7, 32]; numerous implementations are available for knowledge representation and reasoning, e.g., Golog [45]. Logic programming has several variants, such as Prolog-style languages or Answer Set Programming [10, 46], which easily produce the witnesses we need for our queries. Logic programming is based on rules that encode domain knowledge, and powerful reasoning engines to achieve high efficiency. However, they are less prevalent for ontologies (i.e., general knowledge with few or no individuals), and seem better suited for concrete planning tasks.

We aim to leverage domain knowledge for self-adaptation in autonomous systems, e.g., digital twins. Observe that FSRJ's knowledge base and queries fit into Description Logic (specifically, the logic *SRQIQ* underlying OWL 2 [30]). In particular, program coherence (Section 3.5) here maps to *decidable reasoning operations*: *Coh1* expresses satisfiability of a *SRQIQ* knowledge base, *Coh2* the entailment of two sets of *SRQIQ* axioms, and *Coh3–Coh5* the entailment of a subconcept relation from a set of axioms. In Description Logic syntax, we have $\exists \text{in}.\{C\} \sqsubseteq \text{cl}_C$ (*Coh3*) and $\exists \text{in}.\{C\} \sqcap \text{cl}_C \sqsubseteq \bigsqcup_b \text{cl}_b$ (*Coh4*), where cl_C is the classification concept of class `C`. *Coh5* is a special case of type checking knowledge graph access under semantic reflection [39]. Hence, our implementation uses knowledge graphs that connect to existing ontologies.

Further, an implementation of declarative dynamic object reclassification needs to realise semantic lifting to integrate program and knowledge base, and dynamic reclassification via `adapt`-expressions. For these reasons, our implementation is based on SMOL [38], an interpreted research language used to explore connections between programming languages and knowledge graphs, that natively supports heap lifting into a knowledge base. The language is restricted to Description Logics and OWL-based knowledge graphs. The backend uses HermiT [24] for reasoning and Apache Jena (<https://jena.apache.org/>) for graph data management. SMOL was designed to support semantic reflection for the implementation of digital twins [40], but has not yet offered direct programmatic support for reclassification as proposed in this paper.

Our implementation, extending SMOL, goes slightly beyond the design decisions of FSRJ concerning minimality. The main differences are: (1) an explicit constructor that is called after reclassification to establish the classification predicate of the new class; (2) `links`, `classifies` and `retrieves` clauses are inherited; and (3) a more elaborate lifting mechanism, which can be encoded in the one of FSRJ. The constructor ensures a form of reclassification stability, while inheritance of classification predicates ensures a form of behavioural subtyping corresponding to *Coh3* (see Section 3.5). Our implementation supports both the `adapt`-expression of FSRJ and an option for reclassification to be triggered periodically by the runtime system. For static program analysis, we exploit the decidability of description logics underlying knowledge graphs to ensure program coherence. Figure 10 gives the SMOL version of our running example, as well as the knowledge graph used as the knowledge base.

Performance. The most expensive operations of FSRJ are on the lifted heap. Experimental results [35] that compare querying a description logic knowledge base with lifted heaps to querying a representation of the same knowledge expressed in program code, show that reasoning in the description logic knowledge base scales significantly better than reasoning with the knowledge represented in the program, both for more complex knowledge bases (i.e., more possible classification targets) and for more elements (i.e., more constants or individuals in the knowledge graph). The direct representation in the program only performs better for simple (<10 classification targets) and small (<2000 individuals) knowledge graphs. Further, remark that heap lifting in SMOL does not generate a full knowledge base every time the lifted heap is accessed. Instead it uses *virtualisation* [59]: the implementation determines which parts of the knowledge base are relevant for a query, and only generates this part.

5 Evaluation of the Prototype Implementation

To evaluate the prototype implementation of declarative dynamic object reclassification, we aim to demonstrate that the reclassification works in practice as expected from the calculus. Concretely, we consider the following research questions:

- RQ1:** Does derivation in the knowledge base identify the correct class for reclassification?
- RQ2:** Does derivation in the knowledge base instantiate state correctly during reclassification?

These RQs address essential aspects of the declarative reclassification process, namely that the `classifies`-and `retrieves`-predicates are resolved correctly in the knowledge base at runtime.

Experimental Design and Setup. This evaluation is based on GreenhouseDT [41], an open-source digital twin exemplar of a mini greenhouse developed in SMOL as a sensor-driven system that controls the temperature and humidity on different shelves of a greenhouse, as

OWL

```

ctx:NormalOps EquivalentTo: ctx:Pump and (ctx:temp some xsd:double[<= 50.0])
ctx:Overheating EquivalentTo: ctx:Pump and (ctx:temp some xsd:double[> 50.0])
FunctionalDataProperty(ctx:id)

```

(a) General knowledge as OWL ontology.

SMOL

```

1 abstract class Pump (domain String id, Int GpioPin, Plant plant)
2   links "a ctx:Pump";
3   Unit pump() /* ... */ end end
4   class Normal extends Pump () classifies "<ctx:NormalOps>";
5   /* methods */ end
6   class Overheating extends Pump (Int maximal) classifies "<ctx:Overheating>";
7   retrieves "?this ctx:hasMaintenanceProfile [ctx:hasMaximalPower ?maximal]";
8   /* methods */ end

```

(b) SMOL code.

RDF

```

1 ctx:pump1 a ctx:Pump; ctx:id "2"; ctx:temp 52.5. run:ob1 a prog:Seedling.
2 run:ob2 a prog:NormalPump; prog:id "2"; prog:GpioPin "7", prog:plant run:ob1.
3 run:ob2 smol:links [a ctx:Pump; ctx:id "1"].

```

(c) Lifting in RDF.

■ **Figure 10** Instantiation of the pump in the Greenhouse digital twin in the SMOL implementation.

well as the soil moisture level of the plants on these shelves. The digital twin uses these sensor readings to determine the control policy of pumps (the actuators of the digital twin) for watering purposes. In our experiments, we focus on the behaviour of the actuators, i.e., the pumps used to water the plants, and use declarative object reclassification to dynamically determine which controller implementation to use for the pumps and to adopt the controller objects accordingly.

We extend *GreenhouseDT* to capture the following *condition-based maintenance scenario*: to ensure that pumps work properly:

1. the pumps need to operate within a given temperature range, and
2. the pumps should not exceed their expected life time in hours of operation.

If the temperature in an old pump exceeds the temperature thresholds, the pumps could get damaged. For that reason, we implement an extension of the motivating example discussed in Section 2, in which pumps can have one of the following distinct modes of behaviour: normal *operating* mode for pumps that work normally, *overheating* mode for pumps whose temperature is too high, *underheating* mode for pumps whose temperature is too low, and *maintenance* mode for pumps whose temperature is outside the operating range and that have exceeded the expected life span (and therefore need to be stopped). We model the different behavioural modes as reclassifiable subclasses of an adaptable class *Pump*. The exact values for the temperature and age thresholds depend on the type of pump considered; thus, the exact reclassification parameters differ, depending on the specifications of the different pumps. These extensions to the example are realistic as different actuators typically come with different specifications. The extensions also add complexity to the adaptation logic; it is easy to see that as this complexity increases, then a clear separation of concerns between the adaptation logic and the program logic also becomes increasingly attractive, as promoted in the approach of this paper.

■ **Table 1** Pump states based on input temperature and reclassification cycles.

Pump ID	Model	Time (H)	Temperature	Expected Class	Actual Class
<i>After Instantiation</i>					
Pump 1	R385	0	5	Operational	Operational
Pump 2	WPS27	30	80	Overheating	Overheating
Pump 3	R365	3000	94	Maintenance	Maintenance
<i>After First Reclassification</i>					
Pump 1	R385	0	−5	Underheating	Underheating
Pump 2	WPS27	30	41	Overheating	Overheating
Pump 3	R365	2000	50	Operational	Operational
<i>After Second Reclassification</i>					
Pump 1	R385	2500	−5	Maintenance	Maintenance
Pump 2	WPS27	30000	41	Maintenance	Maintenance
Pump 3	R365	300	1	Underheating	Underheating

Our evaluation is performed in terms of two sets of experiments: in the first set, we provide different temperature measurements to evaluate how the system adapts pump controller objects in response to changes in the temperature. We considered scenarios with three pumps with different specifications, and three temperature measurements for each pump: one below the specified temperature range, one above the range and one within the range. In the second set, we additionally let the age of the pumps change to capture a more complex reclassification scenario. In this set, we expanded the previous scenarios with additional age update events that enable the reclassification. In the experiments, we opted to keep the knowledge base invariant; i.e., the actual components that constitute the greenhouse remain the same over time in the domain knowledge base, and only the behaviour of the pump controllers in the virtual layer of the digital twin will change and will require dynamic object reclassification in the digital twin. For **RQ1**, we then inspected the class targeted by the reclassification in each experiment and, for **RQ2**, the resulting object state.

Results. Table 1 illustrates the results from the experiments, comparing the expected class of each pump with the actual class after running `adapt`-expressions on objects with different pump specifications in the domain knowledge base in different scenarios. The actual class of each pump in the experiments is retrieved via the API, whereas the expected class of the pumps in the system is determined from the specifications of the different pumps, combined from constraints for the adaptation logic as expressed in the domain knowledge base, and the actual state of the objects in the system after the reclassification process.

To answer **RQ1**, our experiments used different inputs to adapt the `Pump` objects in the model. When calling `adapt` on an object, the system changes the class of the object depending on the input event, the object state and the domain knowledge base. We checked that the reclassification process worked as formalised in FSRJ by comparing the returned class to the expected ones after running the `adapt` on the `Pump` objects. We ran the two sets of experiments described above on our prototype implementation, in all cases the dynamic object reclassification determined by the `classifies`-predicate selected the correct reclassifiable subclass of the `Pump` class.

To answer **RQ2**, we ran the same sets of experiments as for **RQ1**, and inspected the state of the `Pump` objects after the reclassification process, comparing the values of the fields to the expected ones. In all the conducted experiments, the actual state of the `Pump` objects was

consistent with the expected state, demonstrating that the system was able to derive correct values for the `retrieves`-predicates from the domain knowledge base at runtime. The second set of experiments showed that when considering more complex reclassification constraints, our prototype implementation was able to find a correct state.

Threats to Validity. The evaluation of the prototype implementation of declarative dynamic object reclassification has been done on a specific system, `GreenhouseDT`. Although the adaptation logic involves multiple classes in the knowledge base, the solutions to knowledge base queries have not been very complex. In particular, queries have deterministic solutions in our evaluation setup. We do not foresee particular challenges with non-deterministic queries, as any solution would then be correct (although not necessarily optimal). Furthermore, although the queries in our evaluation involve multiple constraints, they do not involve complex constraint solving (i.e., constraints with many variables). Although well-suited for knowledge representation, knowledge graphs are not particularly strong at complex constraint solving. As our prototype implementation relies completely on the decidable fragments of FOL available in knowledge graphs to answer the queries, the prototype implementation could potentially fail to address *RQ2* in cases where solutions might exist. Remark that we do not see this limitation of the prototype as a limitation of the proposed approach, as constraint solvers such as SMT could be integrated in a more refined query-answering mechanism.

6 Related Work

We focus this discussion of related research on two complementary aspects of our work: dynamic object reclassification techniques and programming with external domain knowledge. To the best of our knowledge, the combination of these aspects is novel to our paper.

There is a body of work on dynamic reclassification and related programming mechanisms (e.g., [2, 8, 13–16, 20, 26]), that addresses “self-extension” [13] and extensions of Java and use of the JVM [8, 14, 16, 26], to the design of novel programming languages [2, 20]. This line of work shares the following limitation: the requirements for the reclassification process are handled internally, as part of the program. In contrast, our paper proposes a novel approach to dynamic reclassification, in which we shift the reclassification process from ensuring internal consistency, with an operational description, to external consistency with a declarative description. This way, our work enables a separation of concerns between the adaptation logic and the application logic of the program, that has not been considered in previous work. In contrast to the works mentioned above, we have considered a very basic programming construct for dynamic object reclassification that does not require an advanced type (and effect) system.

Dynamic Software Updating (DSU) [27, 49, 51, 55] addresses runtime changes to the program code, such as software patching and task updates in real-time. DSU leverages techniques such as aspect-oriented programming [52], or changes Java program behaviour [49, 51] at the JVM or Bytecode level by modifying the code base or the instantiated objects, without halting execution. As code elements change, ensuring consistency becomes a major concern: new objects must allow the system to remain consistent with the specification. This can be solved by modifying old objects or by enforcing type renaming [27]. For DSU on asynchronous systems, statically collected type constraints can be enforced at runtime, delaying upgrades until they are type-safe [33]. Compared to reclassification mechanisms, including our work, DSU addresses externally triggered runtime changes to the program

code, while dynamic reclassification programmatically changes the class of objects based on changes in the heap or context, but does not modify the class table. Although software updates are triggered externally, DSU does not consider external consistency as done in our work.

Programming with external domain knowledge is common for self-adaptive systems [11, 44, 58] in, e.g., robotics [7, 9] and digital twins [17], but these systemic approaches generally lack the guarantees that can be provided by language abstractions. Semantic reflection, originally introduced in SMOL [38], has been used for many purposes in digital twins [40], including to repair external consistency by means of self-adaptive techniques [35–37, 40, 41], but so far the adaptation logic had to be programmed by hand. Semantic reflection has also been applied in digital twin architectures [23] to detect internal inconsistencies that trigger reconfigurations, and to drive domain-aware simulations [53]. Whereas this line of work exploits the lifting of runtime states, the self-adaptation has been concerned with the composition of components and not with changing their internal behaviour. In all cases, the adaptation logic was implemented manually without language or type support. Our work overcomes this limitation by means of linguistic support for declarative reclassification.

The adaptation and retrieval queries of our paper can be seen as a form of LINQ [47], used for reflection with knowledge bases for the underlying serialisation. We are not aware of previous applications of such techniques for reclassification. Golog [45] uses FOL to examine and pick elements from its own state and allows decidable model checking when restricted to description logics [6], but has no type or class system. The systems of Fagin et al. [18] and Calvanese et al. [12] include knowledge bases that can be manipulated and updated by the program through epistemic operators, but have no type or class system either. Here, restrictions to description logics gives tractable knowledge base revision [21, 22, 43]. We are not aware of other work on core calculi for semantically reflected programs, as studied in our paper.

7 Conclusion and Future Work

This paper presents declarative dynamic object reclassification and its formalisation in Featherweight Semantically Reflected Java, a minimal core calculus that formalises declarative reclassification in terms of interaction between a program’s runtime configurations, domain knowledge and environment. We further show how to implement declarative dynamic object reclassification, and how to use these features in practice for adaptable digital twins. A key advantage of this approach is that developers can specify the reclassification process in a declarative way, disentangled from the business code, using queries to determine when and how an object should be reclassified. This makes it easier to reason about the reclassification process, and to ensure that the system remains consistent with an external context. Unlike previous approaches, declarative reclassification is based on external consistency; i.e., reclassification is driven by external domain knowledge, rather than by the internal state of the system. This makes our approach better suited for applications that must adapt to changes in the environment; it further provides a separation of concerns between the program’s business code and its adaptation logic.

Some limitations and possible lines for future work are mentioned in Section 4. This paper does not address the issue of ensuring that the reclassification process is triggered at the right time; instead, reclassification is explicitly invoked. An implicit trigger might extend the range of applications for declarative reclassification; e.g., reclassification could be automatically triggered when certain conditions are met, at fixed time intervals, or during the lifting process.

Such systems would be more dynamic and responsive to changes in the environment. When doing so, it would be crucial to ensure that the system is “free” to act; e.g., the target objects are cold or wait until reclassification is safe. Such constraints point towards DSU and asynchronous update mechanisms; in fact, integrating such mechanisms with a declarative adaptation logic would be an interesting extension of our work. Another interesting line of work is to consider more flexible class hierarchies, e.g., by allowing adaptable subclasses or reclassifiable subclasses of reclassifiable classes, as well as static restrictions to ensure progress, such as type and effect systems to eliminate reclassification attempts on hot objects. Finally, this paper focused on adaptation of cold objects in isolation. In future work we would like to generalise declarative reclassification to reclassification of hot objects and to synchronised reclassification of multiple objects (e.g., all instances of a class or object aggregates) tagged for reclassification, while maintaining the separation of concerns and type soundness of the language.

References

- 1 Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer, 2016. doi:10.1007/978-3-319-49812-6.
- 2 Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. Typestate-oriented programming. In *Proc. 24th Conference Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA 2009)*, pages 1015–1022. ACM, 2009. doi:10.1145/1639950.1640073.
- 3 Torben Amtoft, Hanne Riis Nielson, and Flemming Nielson. *Type and effect systems - behaviours for concurrency*. Imperial College Press, 1999.
- 4 Peter B. Andrews. *An introduction to mathematical logic and type theory: to truth through proof*, volume 27 of *Applied Logic Series*. Kluwer Academic Publishers, 2 edition, 2002. doi:10.1007/978-94-015-9934-4.
- 5 Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003. doi:10.1017/CB09780511711787.
- 6 Franz Baader and Benjamin Zarriß. Verification of Golog programs over description logic actions. In Pascal Fontaine, Christophe Ringeissen, and Renate A. Schmidt, editors, *Proc. 9th International Symposium on Frontiers of Combining Systems (FroCoS 2013)*, volume 8152 of *Lecture Notes in Computer Science*, pages 181–196. Springer, 2013. doi:10.1007/978-3-642-40885-4_12.
- 7 Michael Beetz, Daniel Beßler, Andrei Haidu, Mihai Pomarlan, Asil Kaan Bozcuoglu, and Georg Bartels. Know Rob 2.0 - A 2nd generation knowledge processing framework for cognition-enabled robotic agents. In *Proc. International Conference on Robotics and Automation (ICRA 2018)*, pages 512–519. IEEE, 2018. doi:10.1109/ICRA.2018.8460964.
- 8 Lorenzo Bettini, Sara Capecchi, and Betti Venneri. Extending Java to dynamic object behaviors. In Viviana Bono and Michele Bugliesi, editors, *Proc. Workshop on Object Oriented Developments (WOOD 2003)*, volume 82 (8) of *Electronic Notes in Theoretical Computer Science*, pages 33–52. Elsevier, 2003. doi:10.1016/S1571-0661(04)80801-6.
- 9 Darko Bozhinoski, Mario Garzon Oviedo, Nadia Hammoudeh Garcia, Harshavardhan Deshpande, Gijs van der Hoorn, Jon Tjerngren, Andrzej Wąsowski, and Carlos Hernández Corbato. MROS: runtime adaptation for robot control architectures. *Advanced Robotics*, 36(11):502–518, 2022. doi:10.1080/01691864.2022.2039761.
- 10 Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, 2011. doi:10.1145/2043174.2043195.

- 11 Yuriy Brun, Giovanna Di Marzo Serugendo, Cristina Gacek, Holger Giese, Holger M. Kienle, Marin Litoiu, Hausi A. Müller, Mauro Pezzè, and Mary Shaw. Engineering self-adaptive systems through feedback loops. In Betty H. C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 48–70. Springer, 2009. doi:10.1007/978-3-642-02161-9_3.
- 12 Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Actions and programs over description logic knowledge bases: A functional approach. In Gerhard Lakemeyer and Sheila A. McIlraith, editors, *Knowing, Reasoning, and Acting: Essays in Honour of Hector J. Levesque*. College Press, 2011.
- 13 Alberto Ciaffaglione, Pietro Di Gianantonio, Furio Honsell, and Luigi Liquori. A prototype-based approach to object evolution. *J. Object Technol.*, 20(2):4:1–24, 2021. doi:10.5381/JOT.2021.20.2.A4.
- 14 Tal Cohen and Joseph Gil. Three approaches to object evolution. In Ben Stephenson and Christian W. Probst, editors, *Proc. 7th International Conference on Principles and Practice of Programming in Java (PPPJ 2009)*, pages 57–66. ACM, 2009. doi:10.1145/1596655.1596665.
- 15 Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. Fickle : Dynamic object re-classification. In *Proc. 15th European Conference, on Object-Oriented Programming (ECOOP 2001)*, volume 2072 of *Lecture Notes in Computer Science*, pages 130–149. Springer, 2001. doi:10.1007/3-540-45337-7_8.
- 16 Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. More dynamic object reclassification: Fickle_{||}. *ACM Trans. Program. Lang. Syst.*, 24(2):153–191, 2002. doi:10.1145/514952.514955.
- 17 Romina Eramo, Francis Bordeleau, Benoit Combemale, Mark van Den Brand, Manuel Wimmer, and Andreas Wortmann. Conceptualizing digital twins. *IEEE Software*, 39(2):39–46, 2021. doi:10.1109/MS.2021.3130755.
- 18 Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. Knowledge-based programs. *Distributed Comput.*, 10(4):199–225, 1997. doi:10.1007/S004460050038.
- 19 Kevin Feichtinger, Karl Kegel, Romain Pascual, Uwe Aßmann, Bernhard Beckert, and Ralf H. Reussner. Towards formalizing and relating different notions of consistency in cyber-physical systems engineering. In *Proc. 27th International Conference on Model Driven Engineering Languages and Systems, MODELS Companion 2024*. ACM, 2024. doi:10.1145/3652620.3688565.
- 20 Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. Foundations of typestate-oriented programming. *ACM Trans. Program. Lang. Syst.*, 36(4):12:1–12:44, 2014. doi:10.1145/2629609.
- 21 Giuseppe De Giacomo, Maurizio Lenzerini, Antonella Poggi, and Riccardo Rosati. On the update of description logic ontologies at the instance level. In *Proc. 21st Conference on Artificial Intelligence (AAAI 2006)*, pages 1271–1276. AAAI Press, 2006. URL: <http://www.aaai.org/Library/AAAI/2006/aaai06-199.php>.
- 22 Giuseppe De Giacomo, Maurizio Lenzerini, Antonella Poggi, and Riccardo Rosati. On the approximation of instance level update and erasure in description logics. In *Proc. 22nd Conference on Artificial Intelligence (AAAI 2007)*, pages 403–408. AAAI Press, 2007. URL: <http://www.aaai.org/Library/AAAI/2007/aaai07-063.php>.
- 23 Santiago Gil, Eduard Kamburjan, Prasad Talasila, and Peter Gorm Larsen. An architecture for coupled digital twins with semantic lifting. *Software and Systems Modeling*, November 2024. doi:10.1007/s10270-024-01221-w.
- 24 Birte Glimm, Ian Horrocks, Boris Motik, Giorgos Stoilos, and Zhe Wang. Hermit: An OWL 2 reasoner. *J. Autom. Reason.*, 53(3):245–269, 2014. doi:10.1007/S10817-014-9305-1.
- 25 Barbara Hayes-Roth. A blackboard architecture for control. *Artificial intelligence*, 26(3):251–321, 1985. doi:10.1016/0004-3702(85)90063-3.
- 26 Chengwan He, Zhijie Nie, Bifeng Li, Lianlian Cao, and Keqing He. Rava: Designing a Java extension with dynamic object roles. In *Proc. International Symposium and Workshop on Engineering of Computer Based Systems (ECBS 2006)*, pages 453–459. IEEE Computer Society, 2006. doi:10.1109/ECBS.2006.57.

- 27 Michael W. Hicks, Jonathan T. Moore, and Scott Nettles. Dynamic software updating. In Michael Burke and Mary Lou Soffa, editors, *Proc. Conference on Programming Language Design and Implementation (PLDI 2001)*, pages 13–23. ACM, 2001. doi:10.1145/378795.378798.
- 28 Pascal Hitzler. A review of the semantic web field. *Commun. ACM*, 64(2):76–83, 2021. doi:10.1145/3397512.
- 29 Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia d’Amato, Gerard de Melo, Claudio Gutierrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, Axel-Cyrille Ngonga Ngomo, Axel Polleres, Sabbir M. Rashid, Anisa Rula, Lukas Schmelzeisen, Juan F. Sequeda, Steffen Staab, and Antoine Zimmermann. Knowledge graphs. *ACM Comput. Surv.*, 54(4):71:1–71:37, 2022. doi:10.1145/3447772.
- 30 Ian Horrocks, Oliver Kutz, and Ulrike Sattler. The even more irresistible SROIQ. In Patrick Doherty, John Mylopoulos, and Christopher A. Welty, editors, *Proc. 10th International Conference on Principles of Knowledge Representation and Reasoning (KR 2006)*. AAAI Press, 2006. URL: <http://www.aaai.org/Library/KR/2006/kr06-009.php>.
- 31 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001. doi:10.1145/503502.503505.
- 32 Félix Ingrand and Malik Ghallab. Deliberation for autonomous robots: A survey. *Artif. Intell.*, 247:10–44, 2017. doi:10.1016/j.artint.2014.11.003.
- 33 Einar Broch Johnsen, Marcel Kyas, and Ingrid Chieh Yu. Dynamic classes: Modular asynchronous evolution of distributed concurrent objects. In Ana Cavalcanti and Dennis Dams, editors, *Proc. Second World Congress on Formal Methods (FM 2009)*, volume 5850 of *Lecture Notes in Computer Science*, pages 596–611. Springer, 2009. doi:10.1007/978-3-642-05089-3_38.
- 34 Einar Broch Johnsen and Ingrid Chieh Yu. Dynamic software updates and context adaptation for distributed active objects. In *Principled Software Development*, pages 147–164. Springer, 2018. doi:10.1007/978-3-319-98047-8_10.
- 35 Eduard Kamburjan, Nelly Bencomo, Silvia Lizeth Tapia Tarifa, and Einar Broch Johnsen. Declarative lifecycle management in digital twins. In *Proc. 1st International Conference on Engineering Digital Twins (EDTconf 2024)*, MODELS Companion’24, pages 353–363. ACM, 2024. doi:10.1145/3652620.3688248.
- 36 Eduard Kamburjan and Einar Broch Johnsen. Knowledge structures over simulation units. In *Annual Modeling and Simulation Conference (ANNSIM 2022)*, pages 78–89. IEEE, 2022. doi:10.23919/ANNSIM55834.2022.9859490.
- 37 Eduard Kamburjan, Vidar Norstein Klungre, Rudolf Schlatte, David Cameron, S. Lizeth Tapia Tarifa, and Einar Broch Johnsen. Digital twin reconfiguration using asset models. In *Proc. 11th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2022)*, volume 13704 of *Lecture Notes in Computer Science*, pages 71–88. Springer, 2022. doi:10.1007/978-3-031-19762-8_6.
- 38 Eduard Kamburjan, Vidar Norstein Klungre, Rudolf Schlatte, Einar Broch Johnsen, and Martin Giese. Programming and debugging with semantically lifted states. In *Proc. Extended Semantic Web Conference (ESWC 2021)*, volume 12731 of *Lecture Notes in Computer Science*, pages 126–142. Springer, 2021. doi:10.1007/978-3-030-77385-4_8.
- 39 Eduard Kamburjan and Egor V. Kostylev. Type checking semantically lifted programs via query containment under entailment regimes. In *Description Logics*, volume 2954 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2021. URL: <https://ceur-ws.org/Vol-2954/paper-19.pdf>.
- 40 Eduard Kamburjan, Andrea Pferscher, Rudolf Schlatte, Riccardo Sieve, Silvia Lizeth Tapia Tarifa, and Einar Broch Johnsen. Semantic reflection and digital twins: A comprehensive overview. In Mike Hinchey and Bernhard Steffen, editors, *The Combined Power of Research, Education, and Dissemination: Essays Dedicated to Tiziana Margaria on the Occasion of Her 60th Birthday*, volume 15240 of *Lecture Notes in Computer Science*, pages 129–145. Springer, 2025. doi:10.1007/978-3-031-73887-6_11.

- 41 Eduard Kamburjan, Riccardo Sieve, Chinmayi Prabhu Baramashetru, Marco Amato, Gianluca Barmina, Eduard Occhipinti, and Einar Broch Johnsen. GreenhouseDT: An exemplar for digital twins. In *Proc. 19th Intl. Symp. on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'24)*, pages 175–181. ACM, 2024. doi:10.1145/3643915.3644108.
- 42 Erkan Karabulut, Salvatore F. Pileggi, Paul Groth, and Victoria Degeler. Ontologies in digital twins: A systematic literature review. *Future Gener. Comput. Syst.*, 153:442–456, 2024. doi:10.1016/j.future.2023.12.013.
- 43 Hirofumi Katsuno and Alberto O. Mendelzon. On the difference between updating a knowledge base and revising it. In James F. Allen, Richard Fikes, and Erik Sandewall, editors, *Proc. 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR 1991)*, pages 387–394. Morgan Kaufmann, 1991.
- 44 Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003. doi:10.1109/MC.2003.1160055.
- 45 Hector J. Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard B. Scherl. GOLOG: A logic programming language for dynamic domains. *J. Log. Program.*, 31(1-3):59–83, 1997. doi:10.1016/S0743-1066(96)00121-5.
- 46 Vladimir Lifschitz. What is answer set programming? In Dieter Fox and Carla P. Gomes, editors, *Proc. 23rd Conference on Artificial Intelligence (AAAI 2008)*, pages 1594–1597. AAAI Press, 2008. URL: <http://www.aaai.org/Library/AAAI/2008/aaai08-270.php>.
- 47 Erik Meijer, Brian Beckman, and Gavin M. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis, editors, *Proc. International Conference on Management of Data*, page 706. ACM, 2006. doi:10.1145/1142473.1142552.
- 48 Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 2 edition, 1997. URL: <http://www.eiffel.com/doc/oosc/page.html>.
- 49 Alessandro Orso, Anup Rao, and Mary Jean Harrold. A technique for dynamic updating of Java software. In *Proc. 18th International Conference on Software Maintenance (ICSM 2002)*, pages 649–658. IEEE Computer Society, 2002. doi:10.1109/ICSM.2002.1167829.
- 50 Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.
- 51 Luís Pina, Luís Veiga, and Michael W. Hicks. Rubah: DSU for Java on a stock JVM. In Andrew P. Black and Todd D. Millstein, editors, *Proc. International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA 2014)*, pages 103–119. ACM, 2014. doi:10.1145/2660193.2660220.
- 52 Susanne Cech Previtali and Thomas R. Gross. Dynamic updating of software systems based on aspects. In *Proc. 22nd International Conference on Software Maintenance (ICSM 2006)*, pages 83–92. IEEE Computer Society, 2006. doi:10.1109/ICSM.2006.23.
- 53 Yuanwei Qu, Eduard Kamburjan, Anita Torabi, and Martin Giese. Semantically triggered qualitative simulation of a geological process. *Applied Computing and Geosciences*, 21:100152, 2024. doi:10.1016/j.acags.2023.100152.
- 54 Tim Rentsch. Object oriented programming. *ACM SIGPLAN Notices*, 17(9):51–57, 1982. doi:10.1145/947955.947961.
- 55 Habib Seifzadeh, Hassan Abolhassani, and Mohsen Sadighi Moshkenani. A survey of dynamic software updating. *J. Softw. Evol. Process.*, 25(5):535–568, 2013. doi:10.1002/smr.1556.
- 56 Bjarne Stroustrup. What is object-oriented programming? *IEEE Softw.*, 5(3):10–20, 1988. doi:10.1109/52.2020.
- 57 Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. In *Proc. 7th Symposium on Logic in Computer Science (LICS '92)*, pages 162–173. IEEE Computer Society, 1992. doi:10.1109/LICS.1992.185530.
- 58 Danny Weyns. *An Introduction to Self-Adaptive Systems: A Contemporary Software Engineering Perspective*. John Wiley & Sons, UK, 2020. doi:10.1002/9781119574910.

- 59 Guohui Xiao, Diego Calvanese, Roman Kontchakov, Domenico Lembo, Antonella Poggi, Riccardo Rosati, and Michael Zakharyashev. Ontology-based data access: A survey. In *Proc. Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI 2018)*, pages 5511–5519. ijcai.org, 2018. doi:10.24963/IJCAI.2018/777.
- 60 Jinzhi Lu Xiaochen Zheng and Dimitris Kiritsis. The emergence of cognitive digital twin: vision, challenges and opportunities. *International Journal of Production Research*, 60(24):7610–7632, 2022. doi:10.1080/00207543.2021.2014591.