# KNOWLEDGE STRUCTURES OVER SIMULATION UNITS

Eduard Kamburjan
Einar Broch Johnsen

SIRIUS Centre, University of Oslo
Gaustadalleen 23B, 0373 OSLO, NORWAY
{eduard,einarj}@ifi.uio.no

## ABSTRACT

Modern cyber-physical applications, such as those adopting the Digital Twin paradigm, typically connect simulators with data-rich components and domain knowledge, both often formalized as *knowledge graphs*. Engineering such applications poses challenges to developers. This paper presents a language-based integration of knowledge graphs and simulators for object-oriented languages. We use *Functional Mock-Up Objects* (FMOs) as a programming layer to encapsulate simulators compliant with the FMI standard into OO structures and integrate FMOs into the class and type systems. We show how FMOs can be integrated into knowledge graphs by means of semantical lifting, and used to ensure structural properties of cyber-physical applications. We provide a prototype implementation of the proposed integration and discuss how it can be realized in other languages. Finally, the use of FMOs in practice is illustrated by two case studies.

**Keywords:** Co-Simulation, Object-Orientation, Software Engineering, Knowledge Graphs.

## 1 INTRODUCTION

In the engineering of modern cyber-physical systems, maintainability requires us to handle modularity, reuse, variability and other structuring principles for highly heterogeneous applications that combine simulators, live data streams, control systems, semantic technologies and (cloud) networking. This is especially critical for Digital Twin applications (Barricelli, Casiraghi, and Fogli 2019), which connect simulators according to domain knowledge and must be able to express their structure in terms of the domain to ensure that the digital twin corresponds to the modeled physical twin.

The structure connecting simulators within the application needs to be *configured* in a domain specific way. However, domain experts are not always available and rarely trained to work in interdisciplinary software engineering teams. One solution is to formalize the required domain knowledge, such that it can be queried algorithmically. *Knowledge graphs* (Hitzler, Krötzsch, and Rudolph 2010) are the established technique to formalize domain knowledge. While their potential to enhance Digital Twins has been recognized (Cameron, Waaler, and Komulainen 2018, Rozanec et al. 2020, Kharlamov et al. 2018), their connection to simulators in general, is hitherto unexplored. Similarly, connecting simulation and knowledge graphs *"has been mostly limited to the creation of knowledge bases in the form of ontologies."* (Listl et al. 2020), i.e., to express knowledge, not *use* it. Indeed, a recent survey by Listl et al. (2020) on this topic concluded that *"previous approaches are often tailored to a specific use case and thus a generic solution that provides the flexibility to implement different applications for several domains is still missing."*

In this paper, we propose such a generic solution. Furthermore, we address how a *co-simulation* framework (Gomes et al. 2018) can be used to connect the simulators meaningfully, i.e., according to formalized

domain knowledge. The Functional Mock-Up Interface (FMI) standard (Blochwitz et al. 2012) provides a uniform interface to simulators by encapsulating them in Functional Mock-Up Units (FMUs). These units are defined broadly enough to encompass most components needed for digital twin applications, for example realizing the connection to live data (Feng et al. 2021, Frasheri et al. 2021). Their generic interface, however, poses a challenge for development of the application itself, as it is not integrated into the structures of the programming language used to develop the application. Indeed, co-simulation with FMUs has so far mostly been considered for static connections or for primitive languages without any reuse mechanism (Thule et al. 2019). Tool support, however, is direly needed to not only provide an efficient and working co-simulation, a brittle task in itself (Hansen et al. 2021, Inci et al. 2021), but also for reconfigurations that change the connections in a meaningful way.

Concretelly, we present an approach that directly connects FMUs and knowledge graphs to drastically simplify the engineering of applications that connect simulators and formalized domain knowledge. We integrate FMUs directly into the object model of an object-oriented language as *Functional Mock-Up Objects* (FMOs). This integration of the FMI is transparent in the sense that programmers can use it as a *foreign language interface*: the model description of the FMU is a first-class concept in the language, completely encapsulated in an FMO. Properties of the loaded FMU are treated as fields and its operations as methods. FMOs enable us to integrate the model information of the FMU directly into the type system without requiring the programmer to learn a new library and its pitfalls.

By connecting FMOs with knowledge graphs we can analyze the structure of a program to detect whether it is a meaningful digital twin, i.e., whether it is indeed mirroring its physical counterpart and, thus, can be used as a meaningful representation of it. We connect FMOs to knowledge graphs by *semantically lifting* the FMOs, i.e., by mapping the FMOs to a part of the knowledge graph. This means that the whole program state is interpreted as a *knowledge graph* and can be investigated using semantic web technologies. Technically, we introduce an extension of the SMOL language (Kamburjan et al. 2021), that introduced semantic lifting of object-oriented runtime structures, and use the FMOs to lift the encapsulated FMUs as well.

We illustrate the general principle of FMOs in terms of a digital shadow that reconfigures its parameters upon detecting a drift between simulated and observed behavior with the house example of the Open Simulation Platform (Smogeli et al. 2020). Using semantical lifting as a structuring constraint, we can formulate correct (w.r.t. the domain) digital twins using knowledge graph shapes. Our approach is implemented and available online,[1] but is general enough to be applied to other object-oriented programming languages.

**Contribution and Structure.** Our main contribution is a programming language that treats FMUs as a first-order concept and integrates it into its type system and semantical lifting mechanism. We show how this can be used for to develop digital twin structures. We first give preliminaries on the FMI and knowledge graphs in Sec. 2 before we introduce the programming language in Sec. 3. We discuss its semantical lifting in Sec. 4 and its usage in Sec. 5. We discuss the implementation strategy in Sec. 6 and related work in Sec. 7 before we conclude in Sec. 8.

## 2 PRELIMINARIES

There are many definitions of digital twins (Barricelli, Casiraghi, and Fogli 2019). In this work, we understand by a digital twin an application containing simulators, whose connections mirror the structure of some existing system, the physical twin, such that data flows from digital to physical twin and vice versa. A digital shadow has only data flow from the physical twin to the digital twin.

The *functional mock-up interface (FMI)* (Blochwitz et al. 2012) is a standard for the exchange of simulation units, where a simulation units is, following the formalization of Gomes, Lúcio, and Vangheluwe (2019), a

[1] Source code and examples are publically available under https://smolang.org.

tuple $(S, U, Y, \texttt{set}, \texttt{get}, \texttt{doStep})$, where $S$ is the internal state space, $U$ the set of input variables, $Y$ the set of output variables, $\texttt{set} : S \times U \times \mathscr{V} \to S$ the function to set the values of the input variables to some values of domain $\mathscr{V}$, $\texttt{get} : S \times Y \to \mathscr{V}$ the function to get the results and $\texttt{doStep} : \texttt{S} \times \mathbb{R}^+ \to \mathbb{R}$ the function to perform the simulation for a given amount of time. In the context of the FMI, a simulation unit is called a *function mock-up unit* (for co-simulation) (Gomes et al. 2018).

The FMI defines additional structures for simulation units, such as types or parameter variables, which cannot be reset, and additional information on the correct usage, e.g., the order of calls needed to initialize an FMU. Most importantly, it defines the *model description*, an XML formatted description of input and output variables, and further information about the FMU.

For our purposes, a *knowledge graph* (Hitzler et al. 2010) is a formalization of domain knowledge in terms of a *vocabulary* and *axioms* over terms from the vocabulary. The underlying formalisms for knowledge graphs are (mostly) description logics (Baader et al. 2003), which are decidable fragments of first-order logic. Knowledge graphs are usually expressed in the *Web Ontology Language* OWL (W3C, OWL WG 2012). More concretely, the ontology specifies the vocabulary of *classes* and *properties* that can be used by the system model, and a set of *axioms*, i.e., constraints, to which the model must adhere. Ontologies are today used in many different domains, both within organizations, and as parts of large open projects, like SNOMED CT, an open ontology for clinical terms (SNOMED International 2007).

Data can be expressed as axioms over constants in knowledge graphs, and serialized using the graph-based *Resource Description Framework* RDF (W3C, RDF WG 2014). In RDF, each named node of the graph has the form `prefix:name`. RDF represents knowledge graphs as a set of triples of the form *subject predicate object*. For example, to express that some node `pre:node` is a member of the class `pre:special`, one adds the triple `pre:node a pre:Special`, where `a` is short for `rdf:type`, a special predicate for membership. Subjects and objects are summarized as *nodes*.

There is good tool support to check consistency of semantic models, query them, reason over them to infer new facts, or check if concrete facts are implied. In our work, we need (1) first-order reasoning to check whether a knowledge graph is consistent, i.e., that one cannot derive a contradiction from it, (2) data queries, and (3) shape validation. We use the RDF query language *SPARQL* (W3C, SPARQL WG 2013) to retrieve data from a knowledge graph and *Shapes Constraint Language* SHACL (W3C, SHACL WG 2017) to decide whether certain shapes hold in the knowledge graphs. We introduce OWL, RDF, SPARQL and SHACL using examples throughout the paper.

## 3 A FUNCTIONAL MOCK-UP OBJECT LANGUAGE

To engineer cyber-physical systems, we need language and tool support for the used underlying technologies, which for our digital twins are most importantly simulators and knowledge graphs. For this purpose, we define a programming language that includes support for both technologies. The Semantical Micro Object Language (SMOL) (Kamburjan et al. 2021) is a syntactically standard, statically typed object-oriented language and can be seen conceptually as a subset of Java. SMOL goes beyond standard languages by providing special operations for semantical operations, which we detail below. In this paper, we extend SMOL to additionally support a special construct that loads an FMU as a Functional Mock-Up Object (FMO) into its runtime. After creation, the FMO is transparent; i.e., it can be used like a normal object, albeit with a special FMO-type. In this section we introduce the syntax and runtime representation of FMOs by example.

Syntactically, we extend SMOL by (a) a new category of types to describe FMOs, and (b) a new statement that creates them. An FMO is the object-oriented representation of an FMI. Its in- and out-ports become the fields of the FMO and the functions become methods. For example, the `doStep` operations is a method with a floating point parameter. The initialization cycle is done in an (implicit) constructor, i.e., an FMO

is always in the step mode of the FMI co-simulation state machine (Blochwitz et al. 2012), or throws an exception during object creation.

FMOs are not instantiated from classes but by reading the *Model Description* of a co-simulation FMU. To represent this, we use FMO-types. An *FMO-type* FT is defined by the following grammar, where f ranges over field names and T over non-FMO types (we denote by overbar a comma-separated list of elements):

$$\mathtt{FT} ::= \mathtt{Cont}[\overline{\mathtt{Fld}}] \qquad \mathtt{Fld} ::= (\mathtt{in} \mid \mathtt{out})\ \mathtt{T}\ \mathtt{f}$$

We denote the set of field names with in kind in an FMO-type FT by $\mathtt{iFld}(\mathtt{FT})$, the set of field names with out kind by $\mathtt{oFld}(\mathtt{FT})$, and the type of a field within an FMO-type by $\mathtt{T}(\mathtt{Fld},\mathtt{FT}_1)$.

FMO-types are covariant; i.e., an FMO-type $\mathtt{FT}_1$ is a subtype of another FMO-type $\mathtt{FT}_2$, written $\mathtt{FT}_1 \leq: \mathtt{FT}_2$, if $\mathtt{FT}_1$ contains more fields than $\mathtt{FT}_2$, and the fields that the types share have subtypes in $\mathtt{FT}_1$:

$$\begin{aligned}\mathtt{FT}_1 \leq: \mathtt{FT}_2 \iff & \forall \mathtt{Fld} \in \mathtt{iFld}(\mathtt{FT}_2).\ \mathtt{T}(\mathtt{Fld},\mathtt{FT}_1) \leq: \mathtt{T}(\mathtt{Fld},\mathtt{FT}_2) \\ & \forall \mathtt{Fld} \in \mathtt{oFld}(\mathtt{FT}_2).\ \mathtt{T}(\mathtt{Fld},\mathtt{FT}_1) \leq: \mathtt{T}(\mathtt{Fld},\mathtt{FT}_2)\end{aligned}$$

The FMO-type defined by an FMU is

$$\mathtt{Cont}[\mathtt{in}\ \mathtt{T}_1\ \mathtt{ip}_1,\ldots,\mathtt{in}\ \mathtt{T}_n\ \mathtt{ip}_n,\ldots,\mathtt{out}\ \mathtt{T}_1\ \mathtt{op}_1,\ldots,\mathtt{out}\ \mathtt{T}_m\ \mathtt{op}_m],$$

where Cont (for *cont*inuous behavior) is a type constructor, $\mathtt{ip}_i$ are the input variables of types $\mathtt{T}_i$ and $\mathtt{op}_j$ are the output variables of types $\mathtt{T}_j$. All datatypes of FMI 2.0 are directly representable in SMOL.

To instantiate an FMO, we require an additional statement in SMOL. This statement takes as its parameters the path to a file containing an FMU to read its model description, and a series of assignments that initialize the input variables.

**Definition 1** (FMO Instantiation)**.** *Let* l *range over locations (i.e., object fields and variables),* e *over expressions and* f *over FMO-fields. FMO-instantiation is defined via grammar for the following statement*

$$\mathtt{l}\ =\ \mathbf{simulate}(\mathtt{path},\ \overline{\mathtt{f}\ =\ \mathtt{e}});$$

*Let* $\mathtt{FT}_{\mathtt{path}}$ *be the FMO-type defined by the FMU at* path*,* $\mathtt{T}_\mathtt{l}$ *the type of* l*, and* $\mathtt{T}(\mathtt{e})$ *the type of an expression* e*. An FMO-instantiation is well-typed if* $\mathtt{FT}_{\mathtt{path}} \leq: \mathtt{T}_\mathtt{l}$ *and* $\mathtt{T}(\mathtt{e}) \leq: \mathtt{T}(\mathtt{f},\mathtt{FT}_{\mathtt{path}})$ *for all assignments* f = e*.*

Operations on FMO-typed expressions are field accesses, where in fields can only be written and out fields can only be read. Also, for each supported function there is a corresponding method with the same name.

**Example 1.** *Let us illustrate the use of SMOL to build a self-configuring digital shadow. The shadow consists of one FMU to read the value from a physical system, an FMU of a simulator of this system and a monitor. The simulator has one parameter, which must be adjusted to shadow the system correctly. The monitor compares the current values output by the FMUs to the observations of the physical system and reconfigures itself if the difference is above a certain threshold: A model search is started and the FMU for the simulator is started with possible values for its parameter, until running the simulation in the last time slice matches the system.*[2]

*The following code creates the monitor and the FMOs for the FMUs. The FMUs must both have an output variable* val*, the parameter of the simulator must be an initializable input variable or parameter named* p*. Initially, we set this parameter to* 1.0*.*

[2]For simplicity, we consider the difference between the streams at one point in time and a rather simple model search in this example. Our actual implementation contains less artificial variants of the example.

```
1 Cont[out Double val] sys = simulate("Realsys.fmu");
2 Cont[out Double val] shadow = simulate("Sim.fmu",initVal=sys.val, p=1.0);
3 Monitor monitor = new Monitor(1.0); monitor.run(sys, shadow);
```

*The following code runs the shadow until the threshold is reached. Then, a new shadow based on the last value of the simulator and the last value of the system* prior *to advancing time is searched for. Note that here, the FMUs are transparent and treated as normal object with a* `doStep` *method.*

```
1 class Monitor(Double threshold)
2   Unit run(Cont[out Double val] sys, Cont[out Double val] shadow)
3     while shadow != null do
4       Double last = sys.val;
5       sys.doStep(1.0);  shadow.doStep(1.0);
6       Double d = sys.val - shadow.val;
7       if(d >= threshold) then this.findNewShadow(last, sys.val); end
8     end
9   end
10 end
```

*The model search is given in the method* `findNewShadow`*. We search for a fitting value of the parameter in the range* $[0.5, 1.5]$ *and return the first simulator that stays below the threshold. The method is also part of* `Monitor`.

```
1 Cont[out Double val] findNewShadow(Double last, Double sysVal)
2   Int step = 0;
3   while step <= 10 do
4     this.shadow = simulate("Sim.fmu", initVal=last, p=0.5+step*0.1);
5     this.shadow.doStep(1.0);
6     Double d = sysVal - this.shadow.val;
7     if(d <= threshold) then break; end //New shadow found.
8     step = step + 1;
9   end
10  this.shadow = null; //No new shadow found.
11 end
```

## 4 SEMANTICAL LIFTING OF FMOS

Semantical lifting maps a program state to a knowledge graph, to see the program state through the lense of domain knowledge. In particular, we use semantical lifting to validate the adequacy of the digital twin w.r.t. both the application domain (e.g., to answer questions such as *"is this application connecting the FMUs corresponding to some possible structure of the domain?"*) and digital twin engineering (e.g., to answer questions such as *"are the FMUs connected such that they interact with the physical system in a meaningful way?"*). In this section, we describe the semantical lifting of FMOs. Semantical lifting of program states itself is described in prior work (Kamburjan et al. 2021), and is only described as far as needed.

The knowledge graph we consider consists of the following parts: (i) axioms and data describing general domain knowledge, (ii) axioms describing general knowledge about SMOL states, the *SMOL domain*, and (iii) the data generated by the semantical lifting itself. Part (i) can be some existing ontology, e.g., the IEEE standard ontologies for robotics and automation (IEEE ORA WG 2015, Fiorini et al. 2017). Part (ii), the

SMOL domain, introduces the vocabulary and axioms needed to express knowledge about SMOL program states. We refrain from giving the full definitions here, their RDF formalization is available as part of our open-source implementation. The encoding of FMOs in SMOL is represented as follows.

**OWL Classes.** Extending the basic ontology for object-oriented states (Kamburjan et al. 2021), it states the existence of OO classes, objects, fields, and furthermore the existence of special simulation classes, input ports, output ports. In RDF, the existence of these concepts is expressed by $X$ a `owl:Class.` for

$X \in \{$`smol:OutPort`, `smol:InPort`, `smol:Class`, `smol:Field`, `smol:Simulation`, `smol:Object`$\}$.

Additionally, ports connect an object with some data by $X$ a `owl:DatatypeProperty.` for $X \in \{$`smol:OutPort`, `smol:InPort`$\}$. Each of the names has a prefix (`smol:` or `owl:`) denoting its domain and the identifier itself (e.g., `outPort`).

**OWL Properties.** We furthermore express the domain and range of properties that connect the aforementioned classes as axioms. E.g., `smol:modelName` connects FMOs (modeled as member of `smol:Simulation` with the a String, namely the name of the FMU, taken from its model description. Further properties are `smol:hasPort` connecting FMOs and ports, `smol:hasName` connecting ports with their name, `smol:hasValue` connecting FMOs and their current value.

Next, we explain how the *semantical lifting*, the process of generating a knowledge base from a program state. At runtime, each FMO is represented as a tuple containing (1) an object identifier, (2) the model description of the loaded FMU, (3) a pointer to the FMU itself, and (4) a buffer for all variables to allow reading without invoking the FMU. We do not lift the full model description, which can be easily done using any XML-to-RDF converter, and instead lift a string containing the path to it. Thus, for each FMO the lifting is as follows.

**Definition 2.** *Let* $(X, \texttt{path}, \texttt{fmu}, \texttt{buffer})$ *be the runtime representation of an FMO, where X is an object identifier,* `path` *a string literal containing a file path,* `fmu` *a reference and* `buffer` *a map from port names (as string literals) to pairs* $(\mathsf{kind}, v)$*, where* kind *described the causality of the variable (input, output, local, etc. See (Modelica Association 2021, Sec. 2.2.7)). Let* `name` *be the name as described in the model description in* `path`*. Its lifting is the following set of triples:*

$X$ a `smol:Simulation`      $X$ `smol:hasName` `name`      $X$ `smol:loaded` `path`

*for each variable* `var` $\in \textbf{dom}(\texttt{buffer})$ *there is a node* `run:v` *with:*

$X$ `smol:hasVar` `run:var`    `run:var` `smol:hasKind` $\mathsf{kind}(\texttt{buffer}(var))$    $X$ `run:var` $\mathsf{value}(\texttt{buffer}(var))$

**Example 2.** *Consider the code from Ex. 1 and the state directly before* `run` *is called with an initial value of* $0.0$ *for* `sys.val`*. The lifted knowledge graph is shown in Fig. 1, where none relevant parts of the graph and the* `smol:loaded` *property have been omitted for readability. The core is the lifted* `Monitor` *object: a node* `run:monitor` *that is a member of* `prog : Monitor` *and has properties* `prog : shadow` *and* `prog : sys`*, corresponding to each of its fields. The properties connect to nodes for the FMO, with the lifting as above.*

The knowledge graph can be used to express domain knowledge about the *structure* of a digital twin. For example, we can verify that the monitor indeed has two different FMUs: one for the physical system and one for the simulation. Also we can verify that the monitor stored under `Monitor.shadow` is indeed the shadow (and vice versa). Note that this is *general* knowledge about digital twins. Thus, it can be formulated partially independent of the implementation. Thus, we can achieve a separation of concerns between (1) programming the behavior of a digital twin and (2) ensuring its structural adequacy.

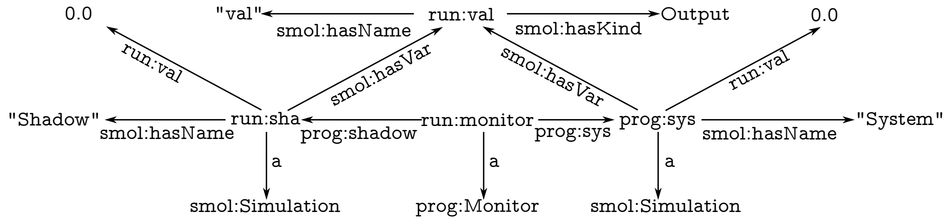


Figure 1: Knowledge graph for the digital shadow example.

## 5 EXPLOITING THE SEMANTICAL LIFTING

Lifted program states can be used to validate the current structure of the program, including its simulators, against domain constraints. This can be either domain constraints from the domain of digital twin engineering, or from the application domain. We first introduce the semantic technologies by example using the domain of digital twin engineering and the digital shadow example from Ex. 1 and Ex. 2: Consistency checking, shape validationg and query answering.

First, we discuss consistency. A simple validation is to check that the lifted state is indeed consistent with the domain knowledge. This task is standard for all description logic reasoners. For example, consider the domain knowledge that expresses that (1) an FMO loading an FMU with name "`Shadow`" is a `ShadowFMO` and (2) that the property `prog:shadow` points to a ShadowFMO. This can be expressed, in the Manchester syntax for OWL (Horridge et al. 2006), as follows:

$$:\texttt{ShadowFMO}\ \textbf{subClassOf}\ \texttt{smol:hasName}\ \textbf{value}\ \text{“}\texttt{Shadow}\text{”} \tag{1}$$

$$\texttt{prog:shadow}\ \textbf{Range}\ :\texttt{ShadowFMO} \tag{2}$$

The first axiom expresses that every object that has the value "`Shadow`" in its `smol:hasName` property is a member of the OWL class `ShadowFMO`. The second axiom expresses that the range of the `prog:shadow` are only members of `ShadowFMO`. If this set of axioms is consistent with the semantically lifted state, then the field `Monitor.shadow` indeed only points to FMUs that are shadowing the real system.

Consistency relies on adding axioms and performing reasoning on the knowledge graph. In contrast, the second semantic technology, SHACL, uses graph shape constraints on the subgraphs that can occur in the knowledge graph. Checking these graphs does not require reasoning and it is thus, potentially, very efficient. For example, the property stating that every `Monitor` instance loads a `ShadowFMO` in its `Monitor.shadow` field can be expressed with the following SHACL shape:

```
ex:ShadowShape a sh:NodeShape; sh:targetClass prog:Monitor ;
  sh:property [
    sh:path ( prog:shadow smol:hasName );
    sh:hasValue "Shadow" ; ].
```

This shape expresses that every node of class `prog:Monitor` has a path through the properties `prog:shadow` and `smol:hasName` which ends in the value `"Shadow"`.

Lastly, while SHACL defines shapes that must be adhered to, it is not capable of expressing negative properties. To this end, we can predefine SPARQL queries that access the knowledge base and should return empty sets. Continuing our example, the following SPARQL query returns `Monitor` instances that have loaded their simulator FMU as the FMU connecting it to the physical system.

```
SELECT ?m WHERE { ?m prog:system ?fmu. ?fmu a :ShadowFMO }
```

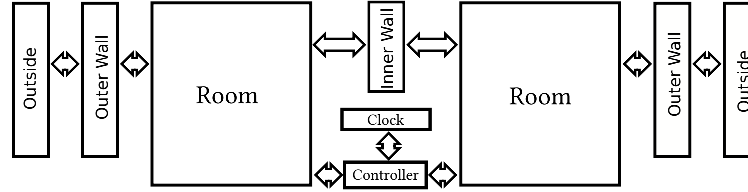


Figure 2: Co-Simulation structure of the Open Simulation Platform house case study.

```
1 abstract class Dyn()
2   abstract Int propagate()
3   abstract Int advance(Double db) end
4 abstract class Wall extends Dyn() end
5 class Room extends Dyn(Cont[in Double h_InnerWall, in Double h_OuterWall,
6                            in Double h_powerHeater, out Double T_room] f,
7                        Wall inner, Wall outer, Controller ctrl,
8                        Boolean left, Int id) end
9 class Controller extends Dyn(Cont[in Double T_room1, in Double T_room2,
10                                  in Double T_clock, out Double h_room1,
11                                  out Double h_room2] dynamics,
12                             Cont[out Double Clock] clock,
13                             Room r1, Room r2, Int id) end
14 class InnerWall extends Wall(Cont[in Double T_room1, in Double T_room2,
15                                  out Double h_wall] dynamics,
16                             Room left, Room right) end
```

Listing 1: SMOL class definitions for the digital twin of the house.

At its core, the query defines also a shape, but it may also use reasoning to derive new knowledge. The above example uses the notion of a `ShadowFMO`, which must be derived first.

The example above shows that we can use knowledge graphs to express properties about digital twins. Next, we show that they can also be used to show that a digital twin is correctly mirroring its physical counter part, i.e., to check its configuration. To this end, we first introduce a co-simulation program based on a house model from the Open Simulation Platform (Smogeli et al. 2020). It models the temperature changes in a house with two rooms and a controller that controls the heaters in the rooms.

**Example 3.** *The structure of the FMUs for a house with two rooms and two heaters is shown in Fig. 2. There is one FMU per room, which is connected to the FMU of the controller of the house and two FMUs for its walls. The outer walls are connected to an outside FMU, while the inner wall is connected to the rooms. The heaters are also connected to the controller FMU itself, which has an external clock FMU.*

A digital twin must be *adequate*: it must mirror the physical system to give meaningful results. Additionally, hypothetical scenarios (e.g., possible changes to the structure of the house) must adhere to basic domain knowledge to be useful. We now give the central parts of the SMOL code and give examples for how we can check both the adequacy of the twin itself and the adequacy of hypothetical scenarios.

**Example 4.** *The classes used to model Ex. 3 are defined in Lst. 1. We omit all methods, as the propagation of value and time advance are not important for adequacy, but only the structure of connections. The* `Dyn` *class realizes a common superclass for co-simulation. Each FMU is encapsulated in one object, which has explicit connections to its neighbors (except the controller and clock FMUs, which share one object).*

**Domain adequacy.** We formalize knowledge about (this kind of) houses in terms of axioms and check whether a certain setup adheres to it. For example, consider the knowledge that a controller must control two different rooms, which is expressed by requiring the following query to always return an empty result:

```
SELECT ?room WHERE { ?ctrl a prog:Controller.
                     ?ctrl prog:r1 ?room. ?ctrl prog:r2 ?room }
```

Observe that this is not knowledge about how to configure a twin, as we considered for the digital shadow. Instead, it is knowledge about how the *physical* controller must be used and, thus, how the *digital twin* must be structured *internally*.

**Structural adequacy.** Let us assume that the digital twin is twinning a concrete building. The structure of the building is captured by the following (partial) knowledge graph:

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

There are two heaters (`asset:heater1` and `asset:heater2`) in two different rooms `asset:room1` and `asset:room2`. For the asset (of which we only show the identifier), we have the following spatial information: `asset:room1` is left of `asset:room2`. We express that the structure of the digital twin corresponds to the physical system as follows. We first introduce knowledge about which heater must be left of the other via a `modelsLeft` property. Note that the heater and room are one object in the digital twin, but they are two nodes in the physical twin.

```
:modelsLeft subPropertyOf asset:in o asset:leftOf o inverse(asset:in)
```

The following query returns all correctly configured digital twins: a SMOL heater and a physical heater are considered the same if they have the same identifier. If the heater `h1` is modeled as left of the heater `h2`, then the SMOL room for `o1` must be the leftmost and `o2` must be not.

```
SELECT * WHERE { ?o1 prog:id ?id1. ?h1 asset:id ?id1.
                 ?o2 prog:id ?id2. ?h2 asset:id ?id2.
                 ?h1 :modelsLeft ?h2.
                 ?o1 prog:left True. ?o2 prog:left False. }
```

## 6 IMPLEMENTATION

The implementation, including the above case studies and further examples, is available as part of the SMOL interpreter (see online auxiliary material). The source code also contains a formal definition of SMOL syntax as an xText (Eysholdt and Behrens 2010) grammar. The SMOL runtime is implemented as an interpreter that takes a set of FMU files and SMOL files. FMOs are handled separately from normal objects, but we do allow `null` to be assigned to an FMO-type location. When loading FMUs, the runtime performs some basic additional checks; e.g., it will only load FMUs for co-simulation and not for model exchange.

The SMOL interpreter implements interactive execution using a Read-Evaluate-Print-Loop (REPL) for debugging, and in particular allows to buffer the output variables of an FMO. Using a special command `draw`, one can print the graph of the buffered FMO.

Our approach in this paper has been based on SMOL in order to reuse the semantical lifting mechanism, which so far has only been implemented for SMOL. The FMO concept itself, however can easily be adopted in any object-oriented language implementing the FMI. The implementation strategy is as follows:

1. Create a wrapper class around the FMI calls to a single loaded FMU, based on the model information of the FMU. This can be done statically or dynamically, if the FMU is not known at compile time.
2. Create a factory that takes an FMU file and returns an instance of the corresponding FMO class.
3. Export a knowledge graph from an object structure, e.g., by implementing a special serializer.

The runtime semantics of SMOL can be used as a guide to design the wrapper classes. The implementation in the interpreter does not explicitly require such a class and uses a more generic Kotlin implementation. The recovery of FMI concepts must then be implemented on top of an abstraction of the program that concentrates on the FMO — a suitable abstraction could be SMOL itself.

As for possible extensions, our implementation so far only supports FMI 2.0 and its data types. One can easily add the vector types of FMI 3.0 and units on top of the data types using standard type system approaches (Bennich-Björkman and McKeever 2018). FMI 3.0 adds event handling to FMUs, which is naturally handled in an object-oriented setting using observer or publish-subscribe mechanisms. Remark that our implementation is under active development and does not yet support all properties of FMI 2.0.

## 7 RELATED WORK

The FMI standard has a number of libraries for bindings into (object-oriented) languages, which are listed under https://fmi-standard.org/tools/. For example, at least three bindings exist for Java alone, of which JavaFMI (Kremers et al. 2022) is used as a library in our implementation. None of these bindings considers the connection to knowledge graphs or semantic web technologies, so we refrain from discussing them in detail. Our approach can easily be converted to applications using any of these libraries following the implementation strategy outlined in Sec. 6. More in line with our work, Thule et al. (2019) give a more formal and language-based approach by introducing the Master Algorithm specification language. Although it does not connect to knowledge graphs, it aims to give a semantic foundation to interconnected simulators.

The idea of using knowledge graphs for co-simulation has been explored by Lu, Wang, and Törngren (2020) to define high-level scenarios. Silver et al. use similar ideas to include a domain ontology to express constraints for simulation in a discrete event simulation framework (Silver, Hassan, and Miller 2007, Han, Miller, and Silver 2011, Silver et al. 2011). In contrast to our paper, their work is mostly concerned with the modeling of the simulation domain and not the surrounding structure. Similarly, Turnitsa, Padilla, and Tolk (2010) give an knowledge graph that encompasses both modeling and simulation. An overview over the combination of knowledge graphs and simulation in more specific fields can be found in the recent survey of Listl et al. (2020).

Contrary to the approaches described above, our aim is not to show that knowledge graphs can be used to express knowledge *about* simulations, simulation scenarios and simulating structures, but to show (a) how to use this knowledge *practically within* a programming language framework, which makes our approach more general and applicable, and (b) how knowledge graphs and simulation interact specifically for digital twins. We believe that this is an underexplored research direction despite the recognized potential of knowledge graphs in this area (Cameron, Waaler, and Komulainen 2018, Rozanec et al. 2020, Kharlamov et al. 2018).

## 8 CONCLUSION

We have presented a language-based approach to develop applications that combine simulators and formalized domain knowledge, as well as its implementation in SMOL. The presented language extension uses Functional Mock-Up Objects as a transparent programming layer that encapsulates the FMI in a standard OO structure and tightly integrates it into the type system. Furthermore, we demonstrate how semantical lifting enables the use of knowledge graphs to ensure structural properties, especially for digital twins.

We show that semantical lifting can be used to express and check crucial properties of both digital twin engineering and domain constraints. For future work, beyond extending our implementation to cover the upcoming FMI 3.0 standard and include the aforementioned unit type system, we plan to investigate the use of semantical lifting at runtime for *dynamic reconfigurations* that reflect changes in the physical twin, as well as inheritance for FMUs in context of semantic lifting (Kamburjan, Klungre, and Giese 2022).

## REFERENCES

Baader, F., D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider. (Eds.) 2003. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press.

Barricelli, B. R., E. Casiraghi, and D. Fogli. 2019. “A Survey on Digital Twin: Definitions, Characteristics, Applications, and Design Implications”. *IEEE Access* vol. 7, pp. 167653–167671.

Bennich-Björkman, O., and S. McKeever. 2018. “The next 700 unit of measurement checkers”. In *SLE*, pp. 121–132, ACM.

Blochwitz, T., M. Otter, J. Åkesson, M. Arnold, C. Clauss, H. Elmqvist, M. Friedrich, A. Junghanns, J. Mauss, D. Neumerkel, H. Olsson, and A. Viel. 2012. “Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models”. In *Modelica Conference*, pp. 173–184, The Modelica Association.

Cameron, D., A. Waaler, and T. M. Komulainen. 2018. “Oil and Gas digital twins after twenty years. How can they be made sustainable, maintainable and useful?”. In *SIMS 59*, pp. 9–16.

Eysholdt, M., and H. Behrens. 2010. “Xtext: implement your language faster than the quick and dirty way”. In *SPLASH/OOPSLA Companion*, pp. 307–309, ACM.

Feng, H., C. Gomes, C. Thule, K. Lausdahl, M. Sandberg, and P. G. Larsen. 2021. “The Incubator Case Study for Digital Twin Engineering”. *CoRR* vol. abs/2102.10390.

Fiorini, S. R., J. Bermejo-Alonso, P. J. S. Gonçalves, E. P. de Freitas, A. O. Alarcos, J. I. Olszewska, E. Prestes, C. Schlenoff, S. V. Ragavan, S. A. Redfield, B. Spencer, and H. Li. 2017. “A Suite of Ontologies for Robotics and Automation [Industrial Activities]”. *IEEE Robotics Autom. Mag.* vol. 24.

Frasheri, M., H. Ejersbo, C. Thule, and L. Esterle. 2021. “RMQFMU: Bridging the Real World with Co-simulation Technical Report”. *CoRR* vol. abs/2107.01010.

Gomes, C., L. Lúcio, and H. Vangheluwe. 2019. “Semantics of Co-simulation Algorithms with Simulator Contracts”. In *MoDELS (Companion)*, pp. 784–789, IEEE.

Gomes, C., C. Thule, D. Broman, P. G. Larsen, and H. Vangheluwe. 2018. “Co-Simulation: A Survey”. *ACM Comput. Surv.* vol. 51 (3), pp. 49:1–49:33.

Han, J., J. A. Miller, and G. A. Silver. 2011. “Sopt: ontology for simulation optimization for scientific experiments”. In *WSC*, pp. 2914–2925, IEEE.

Hansen, S. T., C. Gomes, M. Palmieri, C. Thule, J. van de Pol, and J. Woodcock. 2021. “Verification of Co-simulation Algorithms Subject to Algebraic Loops and Adaptive Steps”. In *FMICS*, Volume 12863 of *LNCS*, pp. 3–20, Springer.

Hitzler, P., M. Krötzsch, and S. Rudolph. 2010. *Foundations of Semantic Web Technologies*. Chapman and Hall/CRC Press.

Horridge, M., N. Drummond, J. Goodwin, A. L. Rector, R. Stevens, and H. Wang. 2006. “The Manchester OWL Syntax”. In *OWLED*, Volume 216 of *CEUR Workshop Proceedings*, CEUR-WS.org.

IEEE ORA WG 2015. “IEEE Standard Ontologies for Robotics and Automation”. *IEEE Std 1872-2015*, pp. 1–60.

Inci, E. O., J. Croes, W. Desmet, C. Gomes, C. Thule, K. Lausdahl, and P. G. Larsen. 2021. "The Effect and Selection of Solution Sequence in Co-Simulation". In *ANNSIM*, pp. 1–12, IEEE.

Kamburjan, E., V. N. Klungre, and M. Giese. 2022. "Never Mind the Semantic Gap: Modular, Lazy and Safe Loading of RDF Data". In *ESWC*, Volume 13261 of *LNCS*, Springer.

Kamburjan, E., V. N. Klungre, R. Schlatte, E. B. Johnsen, and M. Giese. 2021. "Programming and Debugging with Semantically Lifted States". In *ESWC*, Volume 12731 of *LNCS*, pp. 126–142, Springer.

Kharlamov, E., F. Martín-Recuerda, B. Perry, D. Cameron, R. Fjellheim, and A. Waaler. 2018. "Towards Semantically Enhanced Digital Twins". In *BigData*, pp. 4189–4193, IEEE.

Kremers, E., S. Gasnier, V. Galtier, J.-P. Tavella, and M. Caujolle. 2022. "JavaFMI". https://bitbucket.org/siani/javafmi/wiki/Home.

Listl, F. G., J. Fischer, D. Beyer, and M. Weyrich. 2020. "Knowledge Representation in Modeling and Simulation: A survey for the production and logistic domain". In *ETFA*, pp. 1051–1056, IEEE.

Lu, J., G. Wang, and M. Törngren. 2020. "Design Ontology in a Case Study for Cosimulation in a Model-Based Systems Engineering Tool-Chain". *IEEE Syst. J.* vol. 14 (1), pp. 1297–1308.

Modelica Association 2021. "FMI Standard 2.0.3". https://github.com/modelica/fmi-standard/releases/download/v2.0.3/FMI-Specification-2.0.3.pdf, access: 01.07.22.

Rozanec, J. M., L. Jinzhi, A. Kosmerlj, K. Kenda, K. Dimitris, V. Jovanoski, J. Rupnik, M. Karlovcec, and B. Fortuna. 2020. "Towards Actionable Cognitive Digital Twins for Manufacturing". In *SeDiT@ESWC*, Volume 2615 of *CEUR Workshop Proceedings*, pp. 1–12, CEUR-WS.org.

Silver, G. A., O. A. Hassan, and J. A. Miller. 2007. "From domain ontologies to modeling ontologies to executable simulation models". In *WSC*, pp. 1108–1117, IEEE.

Silver, G. A., J. A. Miller, M. Hybinette, G. T. Baramidze, and W. S. York. 2011. "DeMO: An Ontology for Discrete-event Modeling and Simulation". *Simul.* vol. 87 (9), pp. 747–773.

Smogeli, Ø. R., K. B. Ludvigsen, L. Jamt, B. Vik, H. Nordahl, L. T. Kyllingstad, K. K. Yum, and H. Zhang. 2020. "Open Simulation Platform – An Open-Source Project for Maritime System Co-Simulation". In *COMPIT*, Technische Universität Hamburg-Harburg.

SNOMED International 2007. "SNOMED CT". https://www.snomed.org/, access: 01.07.22.

Thule, C., M. Palmieri, C. Gomes, K. Lausdahl, H. D. Macedo, N. Battle, and P. G. Larsen. 2019. "Towards Reuse of Synchronization Algorithms in Co-simulation Frameworks". In *SEFM Workshops*, Volume 12226 of *LNCS*, pp. 50–66, Springer.

Turnitsa, C. D., J. J. Padilla, and A. Tolk. 2010. "Ontology for Modeling and Simulation". In *WSC*, pp. 643–651, IEEE.

W3C, OWL WG 2012. "Web Ontology Language". https://www.w3.org/OWL, access: 01.07.22.

W3C, RDF WG 2014. "Resource Description Framework". https://www.w3.org/RDF, access: 01.07.22.

W3C, SHACL WG 2017. "Shapes Constraint Language". https://www.w3.org/TR/shacl/, access: 01.07.22.

W3C, SPARQL WG 2013. "SPARQL". https://www.w3.org/TR/sparql11-query/, access: 01.07.22.

## AUTHOR BIOGRAPHIES

**EDUARD KAMBURJAN** is a Postdoctoral Fellow at the University of Oslo. His research interests include distributed hybrid systems and engineering of semantic web applications. Email: eduard@ifi.uio.no

**EINAR BROCH JOHNSEN** is a Professor at the University of Oslo. His research centers on formal methods for distributed systems and digital twins. Email: einarj@ifi.uio.no