JOT
JOURNAL OF
OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

Weaving executable models and code for software development

Hugo Richard, Eric Cariou, and Jean-Philippe Babau
Université de Brest, Lab-STICC UMR 6285, France

ABSTRACT One of the initial objectives of MDE is to obtain a complete software application from models through code generation. A standard such as fUML (Semantics of a Foundational Subset for Executable UML Models) enables to add abstract code on UML and DSL (Domain-Specific Language) models, allowing models to contain the entire definition of an application, from its architecture to its business algorithms. However, this approach raises the issue of the ability to reuse existing code written in a traditional programming language and of its adoption by software engineers. In this paper, we propose an intermediate vision where an application is composed of an executable model defining its behavior and traditional code defining its business operations. We propose a generic solution for weaving business operation signatures onto elements of an executable DSL (xDSL), allowing these operations to be executed automatically during model interpretation. We apply our solution to the implementation of a drone simulator software: a xDSL is used to define the simulation architecture model and the simulation functions are implemented in Java. We show that the approach facilitates the configuration and the setup of the simulator.

KEYWORDS xDSL, business code, software development, simulation

1. Introduction

Following a code-centric approach, a software application is implemented by writing code in a programming language (e.g., Java, C++). It is typically described beforehand by using modeling languages such as UML (Object Management Group 2017b). The original MDA (Model-Driven Architecture) approach (Object Management Group 2003), which gave rise to Model-Driven Engineering (MDE), aimed to make productive use of models: they were intended to be self-sufficient for specifying completely an application. At the time MDA was proposed, however, the various types of UML diagrams lacked the ability to define algorithmic or computational aspects.

This gap was addressed by the fUML (Semantics of a Foundational Subset for Executable UML Models) standard (Object Management Group 2021), which allows abstract code to be incorporated into UML models, as well as into models written in domain-specific languages (DSLs), notably through its textual syntax ALF (Object Management Group 2017a). This code is considered abstract in the sense that it is not tied to a specific programming language ; nevertheless, fUML provides all the expressive power of a traditional object-oriented programming language. Consequently, using UML and fUML, it is possible to specify an application with sufficient precision to be executed.

While this solution is consistent with the original vision of MDA, questions remain regarding its usability. First, how can legacy source code or libraries be integrated into a fully model-driven development process? Second, is it appropriate to translate highly technical code—such as a REST service call within a Web component implemented in Vue.js—into abstract, model-level code? Finally, software engineers must be encouraged to learn a new language (fUML) and convinced that a fully model-driven approach is more efficient for their development activities, even though many technical platforms already provide substantial support for their preferred programming languages.

To overcome these limitations, we argue that models should

JOT reference format:
Hugo Richard, Eric Cariou, and Jean-Philippe Babau. *Weaving executable models and code for software development.* Journal of Object Technology. Vol. 25, No. 3, 2026.
http://dx.doi.org/10.5381/jot.2026.25.3.a3

be used where they are most relevant, while code written in traditional programming languages is retained where it is most appropriate. For instance, the orchestration of Web service calls can be modeled using a BPMN workflow (Object Management Group 2014). Such executable models are particularly valuable because they make the system's dynamic behavior explicit and facilitate reasoning about its execution. Conversely, the Web services themselves can be implemented in Java using the Spring platform, which is specifically designed for this purpose and greatly facilitates the development of business code.

The main challenge therefore lies in establishing links between existing code and the executable model, and in using these links to enable the execution of the application by coordinating the execution of both the model and the code. Although ad hoc solutions exist for specific types of executable models—for instance, the jBPM platform (De Maio et al. 2014), which executes BPMN workflows invoking Java code—few studies propose a generic solution for integrating calls to operations written in conventional programming languages into the elements of any executable model for any xDSL.

Our previous work Xmodeling Studio (Cariou et al. 2018) provided mechanisms to link conventional code with executable models within the EMF platform. However, this solution mainly focused on basic code invocation and treated models and code as largely independent artifacts. As a result, it does not adequately support realistic applications in which complex interactions between models and code must be handled. In particular, business operations implemented in conventional programming languages should be able to receive model elements as parameters and modify the model state accordingly. Furthermore, exceptions or errors occurring in the code should be propagated to the model level so that their impact on the model execution can be managed through policies defined in the model.

In this paper, we propose a model-based mechanism for managing complex interactions between executable models and conventional code. The mechanism supports business operation calls based on model elements and handles the impact of operation execution at the model level. The approach has been implemented in an enhanced version of Xmodeling Studio.

To illustrate the benefits of this approach, we apply it to the development of component-based drone simulator software. First, we design an xDSL to define the structure of drones and their environment. We then automatically extend this language to support the association of conventional code with its models. Specifically, this code implements, in Java, the simulation logic for each simulation component defined in a model.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 presents a case study involving a drone software simulator. Section 4 details the Xmodeling Studio approach. Section 5 discusses and evaluates the approach. Finally, Section 6 concludes the paper and outlines future research directions.

2. Related works

In this paper, a software is viewed as consisting of a behavior onto which business operations are woven (Cariou et al. 2016). The behavior defines when and how a given business operation is invoked. For example, in a UML state machine modeling a microwave oven, pressing the power button triggers a transition that leads to a state (corresponding to the behavioral part), whose associated operation turns on the light and activates the magnetron (corresponding to business operations).

Following this vision, software behavior is concretely represented as an executable model, such as a UML state machine. UML state machines natively allow operation signatures to be attached to states or transitions; however, the technical challenge lies in defining the concrete implementation of these operations. This issue arises for any kind of executable model, which motivates the need for a generic solution.

There are three main categories of solutions for associating code with an executable model:

- Full-model approaches, in which the code—or an equivalent representation—is entirely defined within models;
- Programmatic approaches, in which the model is directly programmed or transformed into code that is then extended with business logic;
- Intermediate approaches, in which models and code are used in combination.

In full-model approaches, business actions are defined directly at the model level. To this end, fUML (Semantics of a Foundational Subset for Executable UML Models) (Object Management Group 2021) enables abstract, executable code to be incorporated into models, either through a graphical syntax in the form of activity diagrams or through a textual syntax via ALF (Action Language for Foundational UML) (Object Management Group 2017a). Eclipse/EMF-based MDE tools, such as Papyrus[1], integrate support for associating fUML code with UML models or DSLs (Theobald & Tatibouet 2019; Guermazi et al. 2015). This code is abstract in the sense that it is not tied to a specific programming language; as such, it is platform-neutral, but can be transformed into existing languages such as Java or C++ (as demonstrated, for example, in (Lazăr et al. 2010)). The main advantage of this approach is that all concerns are captured at the model level: the behavior is specified in the executable model, while business actions are expressed using fUML code. However, a major drawback is that this approach does not support the reuse of existing code or business and technical libraries, and it raises broader adoption issues for software engineers. Consequently, it requires them to rewrite legacy code and to abandon their usual programming languages and development environments in favor of a fully model-based approach relying on abstract code.

In programmatic approaches, there exists several tools based on state machines. PauWare (Barbier 2016; Barbier & Cariou 2019) is both an API for programming UML state machines directly in Java and an execution engine that implements the UML semantics for their execution. PauWare state machine code can also be generated from UML models defined using most of the existing UML tools (Cariou et al. 2020). Yakindu (Itemis visited July 2020) is another approach based on state machines, relying

[1] https://www.eclipse.org/papyrus

on Harel's original statecharts semantics (Harel 1987). It allows developers to graphically define a state machine, associate business operation signatures with it, and generate executable code for the state machine. PauWare is limited to Java code while Yakindu can generate code in C, C++ or Java. In both approaches, business operations are implemented in the same language as the generated code from the model and they are linked to this code. For business workflow implementations, such as BPMN (Object Management Group 2014), jBPM[2] (De Maio et al. 2014) is a relevant solution. It enables Java code to be embedded into process activities to build executable applications. For more specific xDSL, CARES (Salmon et al. 2024) enables to define models of a component-based simulation and to generate the executable code of the simulation either in Java or C++. For each component, the software engineer must modify manually a method for calling the associated business code.

All these programmatic tools allow applications to be implemented based on executable models, onto which business code written in Java (or another language) is woven. Their main limitation is that each tool is dedicated to a single type of executable model, that is, to a specific xDSL. Moreover, most of them requires a manual intervention of the software engineer to concretely link the business code with the executable code generated from the models. In contrast, Xmodeling Studio proposes a generic solution applicable to any xDSL and requires almost no manual intervention to obtain a complete application.

Within the category of intermediate approaches, Umple[3] is a model-oriented programming approach (Badreddin & Lethbridge 2013; Lethbridge et al. 2021). It provides an object-oriented programming language extended with UML modeling constructs, such as associations between classes and states and transitions of state machines. From Umple code, concrete code can be generated for a wide range of target programming languages. The generated code is directly executable and complete, without requiring any manual modifications. However, a limitation of Umple is that developers must adopt its specific abstract programming language and that the set of integrated modeling constructs is fixed. Pamela[4] (Guérin et al. 2021) allows models to be dynamically defined using annotations directly within Java code, and enables behavioral code to be associated with these annotations. In this approach, the model is not strictly defined by an explicit metamodel; nevertheless, it allows a model to be woven over the Java code.

These intermediate approaches differ from ours in their objectives. They aim to enrich code with modeling constructs, whereas in Xmodeling Studio, the xDSL and its models are defined first, and links to conventional code are subsequently added to the models.

3. The SPICE case study

To demonstrate the relevance of the proposed approach, we apply it to a concrete case study involving the model-driven development of component-based drone simulation softwares. We then introduce an xDSL, named SPICE, implemented within the EMF platform, to define both the simulation components and their configurations.

3.1. Principles and challenges of the simulator software

To define SPICE, we reuse the CARES concepts and rely on a time-driven simulation paradigm, similar to that adopted by the FMI standard (Modelica Association 2023). A drone simulator typically consists of independent, interacting entities, referred to as components. These components model both the drones and their environment (e.g., wind). They exchange information by sharing state attribute values such as a drone's position or wind speed. In addition to components, the simulator provides views for visualizing the evolution of the simulation.

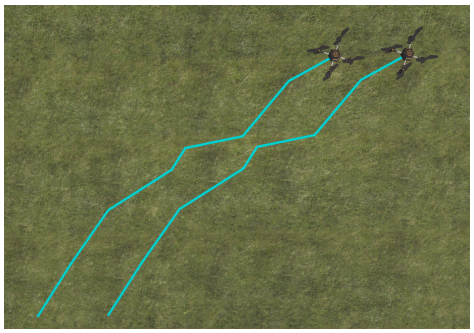

Figure 1 3D view of a simulation with SPICE

Figure 1 illustrates the outcome of a simulation involving two drones. The simulation consists of computing at each time step a new position for each drone. Once the simulation is completed, the resulting simulation trace is visualized in a 3D environment implemented using the Unity engine (Figure 1). The blue lines in the figure represent the simulated trajectories of the drones.

This case study raises several challenges for the software engineer when defining a simulation:

- **Execution characteristics of components**: configuring the execution semantics of components to ensure a realistic simulation (e.g., whether wind evolution should be computed before drone movement, or vice versa).
- **Simulation precision**: determining the appropriate time-step duration required to achieve sufficient realism.
- **Scenario exploration**: enabling the easy evaluation of different wind conditions or drone behaviors, both for configuring the simulation and for exploring multiple scenarios.

These challenges can be summarized as follows: how to provide sufficient agility when defining the model in order to cover a wide range of behaviors.

To address this case study, as usual in MDE, we distinguish in our approach two roles:

- **The language engineer**: responsible for designing the structure (metaclasses) and the execution semantics of the xDSL. In our case, it implements the SPICE xDSL and its execution engine.

[2] https://www.jbpm.org/
[3] https://cruise.umple.org/umple/
[4] https://openflexo.org/pamela/

– **The software engineer**: responsible for using the xDSL to implement a software by combining, in our case, an executable model of this xDSL with concrete Java methods.

3.2. The SPICE language

The creation of the SPICE language is performed by the language engineer. As with any xDSL, the implementation of SPICE is composed of two main elements (Breton & Bézivin 2001; Combemale et al. 2009; Clarke et al. 2013): (i) a metamodel defining the concepts of the language, and (ii) an execution engine responsible for interpreting (i.e., executing) models conforming to this metamodel.

3.2.1. SPICE metamodel

Figure 2 presents the Ecore metamodel of SPICE. Metaclasses shown in blue are defined by the language engineer and correspond to xDSL-specific concepts. Metaclasses shown in green are used to attach operation signatures to xDSL elements. These metaclasses are described later.

To support time-driven simulations, the *Simulation* metaclass defines a start time (*startTime*), an end time (*endTime*), and a simulation step duration (*durationStep*), all expressed in milliseconds. It provides two operations: *init()*, which initializes the simulation, and *run()*, which executes it. The simulation progresses in discrete time steps, during which each *Component* may invoke its *doStep()* operation. This method encapsulates the component's behavior, while *update()* commits attribute changes after each step.

The *Component* metaclass defines a priority level and a period, which determine the components' activation frequency and execution order. The boolean attribute *isDataFlow* specifies the adopted data-flow semantics: when set to *false*, attributes are updated immediately after each *doStep()*, whereas updates are deferred to the next simulation step otherwise. Each component also defines a set of attributes (metaclass *Attribute*) characterized by a multiplicity, a type name, and their observability by a view. The *BasicAttribute* and *ArrayAttribute* specializations store the attribute value (*value*) as an object or a list of objects, respectively. The *newValue* attribute temporarily holds the computed value before it is assigned to *value* according to the data-flow semantics.

3.2.2. SPICE execution semantics

The execution engine implements the execution semantics of the xDSL. Concretely, implementing the execution engine consists in providing the implementations of the Java methods generated by EMF from the Ecore metamodel: *init()* and *run()* in the *Simulation* metaclass, *doStep()* and *update()* in the *Component* metaclass, and *start()* and *update()* in the *View* metaclass.

In SPICE, at each simulation step the engine executes the operations associated with each component according to their period, while respecting component priorities and the selected data-flow semantics. Listing 1 presents an excerpt of the *run()* method of the *Simulation* metaclass. The method first sorts components by priority and initializes the views defined in the model (omitted in the listing and represented by "*(...)*"). The loop starting at line 4 iterates over active components and invokes their *doStep()* method (line 7), which implements the component behavior. Data-flow semantics is enforced as follows: attribute updates occur either immediately after *doStep()* (line 13) or after all components have executed *doStep()* (loop at line 16). This defines the semantics of state attribute sharing, i.e., when modifications become visible to other components. The interaction with Java code execution (lines 8–11) and its impact on the execution engine (lines 20–21) are described later.

Finally, to support the editing of SPICE models, the language engineer defines a concrete syntax, here a textual syntax using Xtext.

3.3. A SPICE-based simulator

Once the metamodel is defined, the software engineer builds a software system (in our example, a drone simulator) by creating an executable model using the concrete syntax.

Listing 3 presents a SPICE model written using the Xtext editor. It defines a simulation involving two drones drifting under wind influence in a terrestrial environment. On line 1, the three numbers following the name *sim* correspond to the start time, the end time, and the simulation step duration, respectively; together, they define the temporal configuration of the simulation.

The simulation is composed of three components: *drone1* (line 2), *drone2* (line 24), and *wind* (line 29). A view is defined on line 38. The drones define a position attribute (lines 3 and 25), while the wind defines a direction attribute (line 30); all the position state attributes can be observed by the view (line 40).

For each component (*drone1*, *drone2* and *wind*), the pair of numbers following its name corresponds to the component's priority and its execution period. A period of 1 indicates execution at every simulation step, whereas a period of 2 indicates execution once every two steps. At the end of the lines 2 and 24, the *DF* keyword indicated that *drone1* and *drone2* components use a data-flow semantics.

At this point, the simulation configuration and the component architecture have been defined. However, a fundamental aspect is still missing: the implementation of each component's behavior (i.e., drones and wind), namely the logic executed by the *doStep()* method.

3.4. Definition of the simulation business logic

The simplest way to implement the computation functions is to rely on conventional code, in this case Java, in order to remain consistent with the language used for xDSL development in EMF. This results in the methods of the *DroneSimulationServices* class, shown in Listing 4. These are standard Java methods relying solely on basic Java types.

Notably, the *checkForCollision* method may raise two exceptions that are specifically defined for the drone simulation domain, indicating either a risk of collision or the occurrence of an actual collision.

The execution of these business functions must be driven by the content of the model: the parameters passed to the Java methods correspond to values of model components (e.g., drone position). This execution may also modify the model itself. For instance, when computing a drone's new position, the corresponding position attribute must be updated.

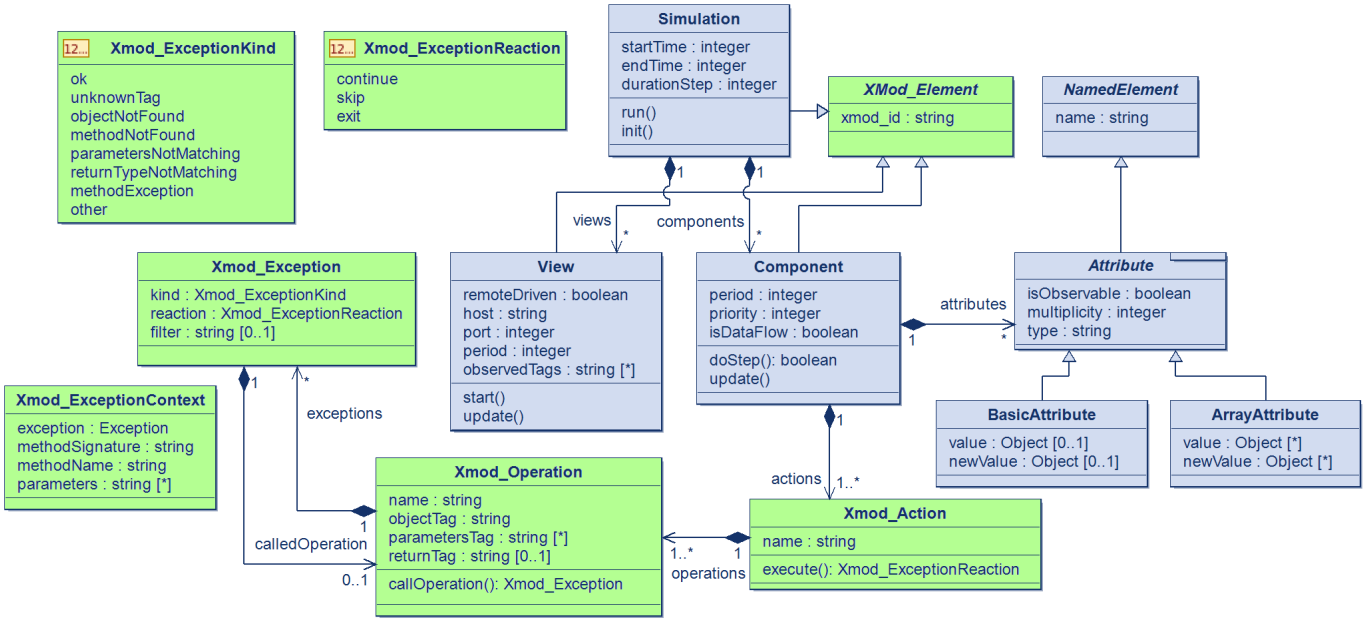

Figure 2 The SPICE Ecore metamodel with SPICE's metaclasses in blue and Xmodeling Studio's in green.

```
public void run() {
    (...)
    // main loop
    for (long i = startTime; i < endTime; i++) {
        for (Component c : components) {
            if (i % c.getPeriod() == 0) {
                Xmod_ExceptionReaction res = c.doStep();
                if (res != null) {
                    switch (res) {
                    case LOCALSTOP: stopped.add(c);
                    }
                }
                if (!c.isIsDataFlow()) c.update();
            }
        }
        for (Component c : components) {
            if (c.isIsDataFlow()) c.update();
        }
        fi.step();
        for (Component c : stopped) {
            components.remove(c);
        }
    }
    (...)
}
```

Listing 1 Excerpt of code in the class *Simulation*

```
public Xmod_ExceptionReaction doStep() {
    for (Xmod_Action action : actions)
        if (action.getName().equals("doStep"))
            return action.execute();
    return null;
}
```

Listing 2 Excerpt of code in the class *Component*

Moreover, it may affect the overall behavior of the simulation. When a drone is removed after a collision, the configuration of the simulation is altered accordingly. Finally, it must be possible to rely on objects external to the model during execution, such as an instance of the class implementing the simulation functions.

For this case study, a model conforming to the xDSL has been defined, and the corresponding business functions have been implemented in Java. What remains is to define and provide appropriate concepts and tooling to facilitate the interaction between the model and the business functions described in Java. This requires working simultaneously at both the language level and the model level. This is what the Xmodeling Studio approach proposes.

4. The Xmodeling Studio approach

4.1. Overview of the Xmodeling Studio approach

Xmodeling Studio is fully generic and can be applied to any xDSL. A detailed description of its concepts and implementation is provided in the next section. Here, we focus on the key aspects as applied to the case study.

First, the metamodel of the xDSL is automatically extended to integrate metaclasses that allow the definition of links to business operations directly within the model. These correspond to the metaclasses shown in green in Figure 2. At this level, we notably introduce the concept of action (metaclass *Xmod_Action*) which is composed of operation calls (metaclass *Xmod_Operation*). An operation call specifies the invocation of a Java method.

Next, metaclasses of the xDSL that requires to define operations whose behavior is implemented in Java are linked to the *Xmod_Action* metaclass. This association enables components to explicitly reference the business logic to be executed during the simulation. In the case study, the *Component* metaclass, due

```
Simulation xmod_id:sim (0,10,1000) {
    Component xmod_id:drone1 (1,1) DF {
        observable Double[3] position = (0.0, 3.0, 0.0);
        Action doStep {
            call calcNextPos(
                    "model:drone1.attributes[position].value",
                    "model:wind.attributes[direction].value",
                    "model:sim.durationStep")
                on "ext:calcs"
                returns "model:drone1.attributes[position].newValue"
                onError methodException("NoChange") then skip
                onError other call handleException("model:xmod_context.exception")
                    on "ext:calcs" then exit;

            call checkForCollision(
                    "model:drone1.attributes[position].value",
                    "model:drone2.attributes[position].value")
                on "ext:calcs"
                onError methodException("OtherDroneIsClose") then continue
                onError methodException("DronesCollided") then localstop;
        }
    }

    Component xmod_id:drone2 (1,1) DF {
        observable Double[3] position = (3.0, 3.0, 0.0);
        (...)
    }

    Component xmod_id:wind (0,3) {
        observable Double[3] direction = (0.0, 0.0, 0.0);

        Action doStep {
            call updateWind("model:this.attributes[direction].value")
            on "ext:calcs" returns "model:this.attributes[direction].newValue";
        }
    }

    View xmod_id:unity {
        port 2223
        observe "drone1.position", "drone2.position";
    }
}
```

Listing 3 A drone simulation model

```java
public class DroneSimulationServices {

    /**
     * Checks the position of two drones for possible
     * collision.
     *
     * @param p1    the position of the first drone
     * @param p2    the position of the second drone
     * @throws OtherDroneIsClose
     *      if the distance between them is <= 2.0
     * @throws DronesCollided
     *      if the distance between them is < 1.0
     */
    public void checkForCollision(Double[] p1, Double[] p2)
            throws OtherDroneIsClose, DronesCollided {
        if (p1.length != p2.length) return;
        double sum = 0.0;
        for (int i = 0; i < p1.length; i++) {
            double d = p1[i] - p2[i];
            sum += d * d;
        }
        double distance = Math.sqrt(sum);
        if (distance < 1.0) throw new DronesCollided();
        if (distance <= 2.0) throw new OtherDroneIsClose();
    }

    /**
     * Calculate the next position of a drone based on its
     * current position and the wind.
     *
     * @param currentPos the current position of the drone
     * @param wind the direction and the force of the wind
     * @param nbSteps the number of simulation steps to use
     * @throws NoChange if the drone has not moved
     * @return the new position of the drone
     */
    public Double[] calcNextPos(Double[] currentPos,
            Double[] wind, int nbSteps) throws NoChange {
        (...)
    }
    (...)
}
```

Listing 4 Simulation functions in plain Java

to its *doStep()* operation, is therefore associated with actions.

Once domain-specific metaclasses are associated with the *Xmod_Action* metaclass, the language engineer has to perform few adaptations to the execution engine code. First, the *doStep()* method of the *Component* class (in the Java code generated by EMF from the metamodel) must be implemented in a generic manner. This implementation is shown in Listing 2: it simply retrieves the action named *doStep* and invokes the *execute()* method on it. This method is provided by Xmodeling Studio and automatically executes all the operations defined by the action, that is, concretely, it invokes the Java methods as specified in the model.

By instantiating this link, it becomes possible to associate operation calls with components (i.e., drones or wind) directly in the model (Listing 3). For the *drone1* component, an action named *doStep* is defined on line 4. This action consists of a sequence of two operation calls: first, a call to the *calcNextPos* method (line 5), which computes the drone's next position, and then a call to the *checkForCollision* method (line 15), which checks whether the drone is at risk of colliding with the other drone.

Both methods are executed on "ext:calcs" (lines 9 and 18), which references an external Java object, named *calcs*, that lies outside the model. This object implements the computation functions used in the simulation and, in practice, corresponds to an instance of the *DroneSimulationServices* class. The first method takes three parameters that are defined in the model: the current position of the *drone1* component (line 6), the wind value provided by the *wind* component (line 7), and the simulation step duration (line 8). The result of this method is stored in the *position* new value attribute of *drone1* (line 10), thereby updating the drone's state.

The *onError* statements (lines 11 and 12) specify how exceptions raised during the execution of the *callNextPos* method are handled. If an unexpected exception occurs, the simulation is terminated using the *exit* statement (line 13), after invoking the *handleException* method (line 12) to log the error. The second method, *checkForCollision*, detects collision situations or potential collision risks by throwing dedicated exceptions. Two distinct behaviors are defined when such an exception is raised: if the drones are close but not colliding, execution continues normally (line 19, *continue*); if an actual collision occurs, the *drone1* component is stopped (line 20) with the *localStop* statement.

The integration of these specific exceptions requires to adapt the main execution loop (Listing 1). On line 10, the concerned component is added to the list of stopped components and is subsequently removed from the simulation in the loop at line 20.

Finally, Listing 5 presents the few lines of code required by the software engineer to launch a simulation; the code is shown in its entirety. First, the Java class implementing the computation functions is instantiated (line 6) and associated with the execution engine using the tag *calcs* (line 7). This tag is used to identify the external object on which methods are invoked (lines 9, 13, 18, and 33 in the model shown in Listing 3). Next, the model is loaded (line 9), which returns an instance of the *Simulation* class, corresponding to the root element of the Ecore metamodel. Finally, execution is started by simply invoking the *init()* and *run()* methods (lines 11 and 12) on the simulation object loaded from the model. The *SpiceUtils* class used to register external objects and load the model is a utility class that is fully and automatically generated by our tool.

The simulator model is thus ready to be executed through interpretation, seamlessly integrating business logic. We now describe the underlying concepts and their implementation that enable the automation of this process.

4.2. The Xmodeling Studio concepts and implementation

Xmodeling Studio is an EMF plugin[5] whose aim is to add operations on elements of any xDSL. Concretely, these operations are Java methods developed in any class. It offers the following features through what is called the Xmodeling tranformation:

- A metamodel transformation for adding the metaclasses associating operation calls with the xDSL;
- The corresponding Java code for their automatic invocation during the execution;

[5] The plugin is available here: https://github.com/h-richard/Xmodeling_Studio
The SPICE case study using Xmodeling Studio is available here: https://github.com/h-richard/Spice

```java
public class Controller {

    static final String MODEL = (...) + "/sim.spice";

    public static void main(String[] args) {
        DroneSimulationServices calcs = new DroneSimulationServices();
        SpiceUtils.put("calcs", calcs);

        Simulation sim = SpiceUtils.loadSimulationFromSpiceModel(MODEL);

        sim.init();
        sim.run();
    }
}
```

Listing 5 Launching the execution engine with the model and the external Java objects

– A fragment of Xtext grammar for editing the operation calls and a validator for verifying their correctness;
– The generation of an utility Java class for loading and connecting the model to external Java objects used by the operation calls.

4.2.1. Requirements for model-code integration

The SPICE case study illustrates how the integration between models and code is achieved through the definition of operation calls associated with the model, involving both external Java objects and model elements used during operation calls.

More generally, we identify the following requirements for supporting the integration between models and code:

RQ1 The execution engine must be able to automatically invoke the Java methods associated with model elements.

RQ2 Any external Java object may be used as the object on which the method is called (it will be called the *target object* in the following), as well as a parameter or a return value of the method call.

RQ3 Any model element may be used as the target object, as well as a parameter or a return value of the method call.

RQ4 If a method cannot be invoked for any reason (e.g., a missing object or an invalid method name), the issue must be handled by the execution engine according to the strategy specified by the software engineer in the model.

RQ5 If a method call raises an exception, it must be handled by the execution engine according to the strategy specified by the software engineer in the model.

RQ6 During model editing, the software engineer must be informed whenever an operation call definition is invalid (e.g., due to an incorrect navigation in the model leading to an invalid object reference).

The first version of Xmodeling Studio (Cariou et al. 2018) only fulfilled RQ1 and RQ2. It allowed operations to be performed solely on external Java objects, without access to model elements, and did not provide any support for handling errors raised during method calls. As a result, it was limited to a proof of concept and did not enable the development of realistic applications.

In this article, we present a major enhancement of Xmodeling Studio that fulfills all the identified requirements (albeit only partially for RQ6), thereby offering a more seamless integration between models and Java code. This enhanced version significantly broadens the applicability of Xmodeling Studio beyond a proof-of-concept setting.

4.2.2. Xmodeling transformation

As said above, the Xmodeling transformation generates four modeling or code elements that enable an xDSL to define business operations and to execute them while interpreting an executable model.

The first step of the transformation consists of a metamodel transformation that extends the metamodel with the required metaclasses to enable the association of operation calls with selected xDSL meta-elements. The language engineer specifies the connection points for the newly introduced metaclasses (shown in green in the figure) by annotating the domain metaclasses (shown in blue in Figure 2) with the following Ecore annotations:

– *Xmod_main*: identifies the root element of the metamodel (here, *Simulation*);
– *Xmod_elem*: identifies elements that can be referenced in operation calls defined in the model (here, instances of *Simulation*, *Component*, and *View*);
– *Xmod_exec*: identifies executable elements to which operation calls must be associated (here, *Component*).

The metamodel transformation simply consists of integrating the Xmod metaclasses (shown in green) into the metamodel, and linking the metaclasses annotated with *Xmod_elem* and *Xmod_exec* to *Xmod_Element* through inheritance and to *Xmod_Action* through composition, respectively, based on the annotations added into the metamodel.

Second, the transformation generates the corresponding EMF Java code of these new metaclasses. The code of the added metaclasses contains also a generic Java code for automatically and dynamically execute the operation calls defined in the model. To implement this behavior, the Java language was chosen for its compatibility with Eclipse's EMF framework, but also for its reflection capabilities. The method to be executed is retrieved through introspection on the Java object on which the method has to be executed, based on the types of the object parameters defined in the operation call. The language engineer needs to

link this generic code in the execution engine simply by calling the *execute()* method of an instance of *Xmod_action* as done line 4 in listing 2. This generic code for executing any Java method associated with the model fulfills RQ1.

Third, a fragment of the Xtext grammar along with its associated validator is also automatically generated. This grammar defines the textual syntax for defining operation calls as used in the model of listing 3 (see 4.2.6).

Fourth, the transformation generates the utility class for loading a model based on the root metaclass of the metamodel (here, *Simulation* that has been annotated with *Xmod_main*) and for setting external objects before launching the execution of the model. This class, named *SpiceUtils* in our example, is used lines 7 and 9 of listing 5.

This Xmodeling transformation step provides the structural foundations required to support model-driven invocation of business operations. The rest of this section details the way the operation calls are defined and how the problems that can occur during the execution of the corresponding Java methods are managed.

4.2.3. Java objects referencing

When defining operation calls, it is necessary to reference Java objects that may serve as the target object of a method invocation, as parameters of the method, or as locations in which the result of a method execution is stored. In our approach, objects are referenced using a string that follows the pattern (that will be called a *reference* in the following): "nameSpace:objectTag.navigationPath". The *nameSpace* and *objectTag* parts are mandatory, whereas the *navigationPath* part is optional.

The namespace specifies whether the referenced Java object is external to the model (value *ext*) or corresponds to a model element (value *model*). The object tag denotes the name assigned to the object. For instance, "ext:calcs" references an external object named *calcs*. This reference is used in the model on line 12 (Listing 3) and is concretely associated with the corresponding Java object after its instantiation in the simulation launch code on line 7 (Listing 5). In this way, a link is established between the object reference used in the model and the concrete Java object available at runtime.

For references in the *model* namespace, the referenced object corresponds to a model element, that is, an instance of a metaclass. Such object must be an instance of a metaclass specializing *Xmod_Element*; the tag corresponds here to the value of its *xmod_id* attribute. This attribute enables to uniquely identify the elements of the model. For example, "model:drone1" references the model element whose identifier is *drone1*, namely the component defined on line 2 of Listing 3.

Within the model, the *navigationPath* enables navigation starting from a given object by following its attributes or associations, according to the structure of the metamodel. As commonly done for model navigation, we use dot notation. A reference of the form "model:a.b.c" denotes access to attribute (or association) *c* of attribute (or association) *b* of object *a*. For instance, "model:drone1.attributes" retrieves the list of attributes of the *drone1* component. This is a collection, since the *attributes* association in the *Component* metaclass has a cardinality of 0..*.

For collections, we provide a basic filtering mechanism to select a specific element. The filter selects an element based on its name, which is either the value of its *xmod_id* attribute or the value of a string meta-attribute named *name* defined in the corresponding metaclass. For example, the reference "model:drone1.attributes[position]" retrieves the attribute of *drone1* whose *xmod_id* value is *position*. Once this element has been selected, its *value* attribute—defined in one of the specialization of the *Attribute* metaclass—can be accessed using the reference "model:drone1.attributes[position].value".

Finally, we introduce two special references. The reference "model:this" denotes the current model element in which the reference string is defined, enabling navigation without explicitly naming the element (it is used line 33 and 34 in the model for the *wind* component). The reference "model:xmod_context" denotes an instance of the *Xmod_ExceptionContext* metaclass, which is updated during model execution and is primarily used to retrieve the last exception raised during the execution of an operation call (it used line 12 in the model).

At runtime, a string-based object reference is evaluated to retrieve the corresponding concrete Java object by dynamically invoking a sequence of getter methods that mirrors the navigation chain expressed in the reference. This sequence is built dynamically for each use of the reference. When a reference is used as a return value, the final step of the sequence consists in invoking a setter method to update the required object.

This mechanism for getting a Java object from a reference is fully generic (i.e., it applies to any metamodel) and dynamic. It can also be applied to external objects. In this case, navigation does not rely on attributes or associations defined in the metamodel, but instead invokes the corresponding getter or setter methods defined in the Java classes of the external objects.

4.2.4. Operation calls mechanism

In this subsection, we explain how an operation call defined in the model results, at runtime, in the invocation of a concrete Java method.

After the metamodel transformation, an *Xmod_Action* is associated with xDSL metaclasses (here, *Component*). Each action defines a sequence of operation calls (*Xmod_Operation* metaclass). An action is identified by a *name*, which is instantiated at the model level. Several actions can be defined for a model element. This is useful when different operations must be executed at different moment of the execution for the same element. For example, the UML state machine semantics enable to define three actions for a state: as entry (when the state is activated), as exit (when a state is deactivated) and as what to do when the state is active.

In the model (Listing 3), an action named *doStep* is defined for the drone component on line 4 and contains two operation calls (lines 5 and 15). In the execution engine code, the *doStep()* method of the metaclass *Component* (Listing 2) selects this action using its name (line 3) and executes it (line 4).

An operation call (metaclass *Xmod_Operation*) contains the name of the Java method to call (*methodName*), the reference of the Java object on which the method has to be called (*objectTag*), the references of the Java objects passed as parameters

(*parametersTag*) and the reference on the Java object that is set by the returned value of the method (*returnTag*). At line 5 on the model, there is the call of the *calNextPosition* Java method on the Java object referenced by "ext:calcs" with 3 parameters (in the parentheses) and that is modifying the *newValue* attribute of *drone1*.

As the references involved in an operation call may either be model elements or external Java objects, this fulfills requirements RQ2 and RQ3.

4.2.5. Exceptions mechanism

At runtime, an operation call definition results in the invocation of a Java method. Two main types of issues may arise during this process: either the method cannot be resolved because the information defining the operation call is invalid, or the method execution raises an exception instead of returning normally.

The *Xmod_ExceptionLiteral* enumeration defines the different types of issues that may occur during the execution of an operation call:

- *unknownNamespace*: the specified *nameSpace* is invalid.
- *objectNotFound*: the Java object cannot be retrieved from the reference.
- *methodNotFound*: no Java method with the specified name can be found for the target object.
- *parametersNotMatching*: no Java method with the specified name matches the given list of parameter types for the target object.
- *returnTypeNotMatching*: the return type of the Java method is not assignable to the type of the object to be updated.
- *methodException*: the Java method has been invoked but has thrown an exception (hereafter referred to as a *business exception*).
- *other*: represents any exception that is not explicitly handled (used only during model editing and not during execution).
- *ok*: no issue occurred during execution.

By default, the behavior in case of an issue is to ignore it. This behavior can be customized at the model level. The software engineer may define a specific reaction for each type of issue or business exception. Since an action is defined as a sequence of operation calls, it is also possible to specify how the execution of the remaining operations in the sequence should proceed when an error occurs.

The *Xmod_ExceptionReaction* enumeration defines three generic reactions that can be applied when an error occurs during an operation call:

- *continue*: execution of the current sequence of *Xmod_Operation* elements continues.
- *skip*: execution of the current sequence of operations is aborted.
- *exit*: execution of the entire program is terminated.

We also provide language engineers with the capability to define xDSL-specific reactions. A specific reaction stops the execution of the operation sequence (it can be seen as a specialized *skip*). In the case study, this is illustrated by the introduction of the *localStop* literal in the enumeration. This reaction leads to stop the execution of a component when an issue occurs. The simulation then continues its execution without this component.

An operation call defines error management with instances of *Xmod_Exception* metaclass that associates a specific kind of error with a corresponding reaction. When dealing with a business exception, the *filter* attribute can be used to specify the class name of the exception, thereby enabling the definition of a reaction for a particular business exception. Optionally, an operation may be executed to provide more advanced exception handling. This is achieved through the *calledOperation* composition, which allows an operation call to be defined in the same way as regular operation calls.

This operation may take as parameter the reference "model:xmod_context" (instance of the *Xmod_ExceptionContext* metaclass) that encapsulates the exception context. This context contains the Java exception that was raised, the name of the invoked method, the list of parameter types used in the call, and a method signature combining the latter two.

Now, the exception reaction is described at operation level. To make a link with the execution engine, the reaction is propagated to *execute()* method of the metaclass *Xmod_Action*. It is the reaction that is returned by the last error management executed for an operation call. In our example, this mechanism is used to retrieve the *localStop* reaction in the code of the execution engine (lines 7 and 10 of listing 1). This reaction is sent by the exception management defined in the model (line 20 of listing 3).

To illustrate this management of error, in the use case model (Listing 3), exception reactions correspond to all lines starting with the *onError* keyword. For instance:

- **Line 12 (position computation)**: when computing the next position of a drone, any exception that is not explicitly handled triggers the invocation of the *handleException* method on the "ext:calcs" object, with "model:xmod_context.exception" passed as a parameter, which contains the exception that was raised. The execution then terminates using the *exit* reaction. As a result, any unexpected exception stops the entire simulation after logging the error.
- **Line 20 (collision checking)**: when checking whether two drones are close, if a business exception of type *DroneCollided* is raised, a *localStop* reaction is selected. This reaction is propagated as part of the result of the enclosing action execution and is handled by the execution engine (line 10 of Listing 1), which removes the current component from the simulation (lines 10 and 20). The simulation then continues without the destroyed component.

This fine-grained error management at the model level fulfills requirements RQ4 and RQ5.

4.2.6. Editing facilities

A textual concrete syntax based on Xtext is provided and has been illustrated through the case study model. In addition to the Xtext grammar fragment, a dedicated validator is also proposed. During model editing, the validator

analyzes the references whenever the *nameSpace* is *model*. It ensures that these references refer either to an existing model element or to an element that may be instantiated later, provided that the metamodel allows it.

If the last element of a reference corresponds to a field (i.e., neither a collection element nor a primitive value) and this field is not explicitly defined, it may refer to a getter method that does not have a corresponding field (see Section 4.2.7 for an illustrative example). In such cases, the validator reports a warning rather than an error.

However, the validator cannot check references for external objects (*nameSpace* is *ext*), as these objects are not present in the model and are instantiated only when the simulation is launched. Likewise, it cannot verify the existence of a method with a given name and parameter types on the target object. For these reasons, requirement RQ6—concerning the validation of operation calls during model editing—is only partially fulfilled.

4.2.7. Type alignment Beyond the verification of references and method names, object type checking would also be required. However, this aspect is currently outside the scope of this paper. At present, only explicit type casts are supported to handle type alignment issues between the model and Java types. Java simulation methods (e.g., Listing 4) use basic standard Java types—for instance, a 3D position is represented as a *Double[]* array—whereas EMF relies on its own type hierarchy. Although primitive types are generally compatible (e.g., EMF's *EDouble* and Java's *Double*), this compatibility does not extend to all types, particularly collections.

For instance, in the model (line 3), the position attribute of component *drone1* contains three double values. Because the model is edited with Xtext on top of EMF, this collection is represented as an *EList<Object>* (EMF's list implementation). However, Java business methods typically expect an array of doubles (e.g., *Double[]*) or similar types such as *List<Double>*. Consequently, invoking a method that expects a *Double[]* with the reference "drone1.attributes[position].value" fails.

To overcome this issue, we propose to implement two methods in the metaclass *ArrayAttribute*:

- *public void setDoubleArrayValue(Double[] value)*
- *public Double[] getDoubleArrayValue()*

These methods perform the conversion between an *EList<Object>* (the concrete type of the attribute value in EMF) and a *Double[]* (the type expected by Java methods).

Consequently, the model must use the reference "drone1.attributes[position].doubleArrayValue" for the drone position[6]. Since navigation relies on the dynamic invocation of getters and setters, these methods are transparently called, even though the *doubleArrayValue* attribute does not exist in the metamodel.

4.3. Discussion on usability

The goal of our approach is to achieve the smoothest possible integration between models and code, while at the same time remaining generic for any xDSL. We discuss here these aspects by considering model editing and verification activities from the perspective of software engineers.

Navigation within model content is based on the structure of the metamodel. For example, retrieving the position value of a drone corresponds to the reference "drone.attributes[position].value". This reference indicates that the set of attributes (the composition called *attributes* from the metaclass *Component* to the metaclass *Attribute*) for the instance of component named *drone*, is filtered to obtain the element named *position*, from which the *value* attribute is then accessed. An expression such as "drone.position.value" appears more readable, as it manipulates only business-level model concepts. However, navigation based on the metamodel structure is required because the code that translates such references into a sequence of Java object calls is generic: this single piece of code must operate on any model conforming to any metamodel, without requiring any modification. As a result, this approach requires software engineers to be familiar not only with the syntax used to edit the model, but also with the structure of its underlying metamodel.

We provide a checker integrated with the Xtext grammar to verify that references to model elements are syntactically correct. However, this mechanism cannot checked the references to external Java objects as the structure of their classes is unknown at model-editing time. Similarly, it is not possible to determine whether a method with a given name and parameter types actually exists, because the class of the object on which the method will be invoked is not known during model editing—unless the object is part of the model itself. As a consequence, such errors are detected when the built software is executed, at which point an exception is thrown to indicate that the method was not found.

In a Xmod action, sequences of operations are the only control structures provided. Although conditional constructs such as if–then–else statements or loops could have been introduced, this design choice was made to avoid reintroducing the complexity of a general-purpose programming language into the modeling layer. When such control structures are required, they can be implemented within Java methods using standard language constructs. These higher-level Java methods can then be associated with model elements and invoked as single operations within operation sequences.

5. Evaluation

In this section, we evaluate the capabilities and advantages of the XModeling Studio approach for software implementation based on executable models.

5.1. Improvement of Xmodeling Studio

Compared to the first version of Xmodeling Studio (Cariou et al. 2018), the new version fulfills all the requirements listed in Section 4.2.1, albeit only partially for RQ6. The initial version of Xmodeling Studio satisfied only RQ1 and RQ2. The main improvement lies in the bidirectional communication between models and code: Java methods can now access model content

[6] For readability, Listing 3 uses the intuitive notation "drone1.attributes[position].value". In the SPICE GitHub implementation, the adapted reference is used.

(in reading and modification), and exceptions raised during execution can be captured and handled through policies defined at the model level.

The SPICE xDSL and its implementation of a drone simulator demonstrate that the first version of Xmodeling Studio was not sufficient to implement simulation software representative, in our view, of systems combining model-based behavior and business logic. In contrast, the version presented in this paper is capable of supporting such systems. The new version of Xmodeling Studio therefore provides significant enhancements to the tool's capabilities.

5.2. Applicability to other xDSL

To illustrate the applicability of Xmodeling Studio to any type of xDSL with varying execution semantics, we apply it to a UML-like hierarchical state machine xDSL featuring event-based semantics. According to the UML specification (Object Management Group 2017b), states may define entry, exit, and do actions, and transitions may be annotated with actions and guards. In Xmodeling Studio, such actions and guards are implemented in Java and woven into the model. As a case study, we model the classical microwave oven application by combining a state machine model with microwave control logic. Due to space constraints, the details are not presented here. For more details see [7].

5.3. Dependency on the EMF environment

In this section, we show how to implement a software based on an executable model outside of the Eclipse/EMF environment in order to broaden software development capability.

Xmodeling Studio is integrated into the Eclipse ecosystem and relies on EMF. The xDSL metamodel is defined in Ecore using Eclipse/EMF tools, the execution engine is built on the Java/EMF code generated from this metamodel, and models are edited in Eclipse using the Xtext editor. Nevertheless, once the execution engine has been implemented, it can be packaged as a JAR file, together with the required EMF and Xtext dependencies[8]. These artifacts can then be reused in a standard Java project developed with other IDEs, such as IntelliJ IDEA, Visual Studio Code, or NetBeans, where the Java methods associated with the model are implemented. As a result, it is possible to build and run a Java application that executes models outside of the Eclipse environment.

As an example, the case study of this paper has been implemented in the following way:

- Once the SPICE xDSL defined, the model of the drone simulation has been edited under Eclipse/EMF (listing 3).
- An IntelliJ Java project has been created for implementing the simulation operations associated with the model (listing 4).
- The Java program implemented with IntelliJ launches the simulation, i.e, the SPICE engine with the model as parameter (listing 5). The model expresses with a view that after each simulation step, the position of the two drones is sent, under a JSON format, to the TCP port number 2223 (lines 38-40 of listing 3).
- A C# program[9], also implemented with IntelliJ, is listening at this port and is drawing the 3D environment in which the drones are moving, by using the Unity framework (a screenshot of this program is on figure 1).

5.4. Generated code versus executable models

Our approach is based on co-execution of models and code. This is an alternative to code generation that is used by a lot of approaches for executing models as explained in the related work section. In this section, we compare the capabilities and ease to use of each technique as we have already developed a similar simulation platform based on code generation.

CARES (presented in (Salmon et al. 2024)) and SPICE (presented in this paper) share the same objective: the development of drone simulators. Both approaches rely on similar concepts and adopt time-based execution semantics inspired by FMI. They are both implemented as a model-based EMF environment. The main difference lies in their execution strategy. SPICE is based on model interpretation (i.e., direct execution) and is fully dynamic, whereas CARES relies on code generation.

In CARES, simulations models are defined in a similar way as with SPICE. There is no significant difference in terms of complexity when defining the components of a simulation (drone, environment, etc.). Once the models defined, CARES generates the executable code of the simulation (in Java or C++). This code contains two parts: the generic code executing the simulation loop and the specific code of the components defined in the model. Each model's component is translated into a class containing the component's attributes and an empty *void doStep()* method. Simulation developers must then manually implement this *doStep()* method by adding the code required to invoke the corresponding simulation functions. In addition, the effects of these simulation functions on model attributes are handled directly at the code level through explicit attribute updates.

Listing 6 gives the implementation of the *doStep()* method for the *Drone* Java class generated by CARES. The developer must implement by hand the call of the two operations and the management of their exceptions. This code implements the same behavior for a drone as in the SPICE model of listing 3.

Overall, writing this code is not much more complex than defining these calls directly in the model with SPICE. However, there are two main problems. The first is that it requires finding the *Drone* class in a hierarchy of 7 directories and a total of 21 generated files. This *Drone* class already contains 105 lines of generated code which must not be changed. This makes adding the simulation part of a drone more tedious to do.

The second problem, which is much more significant, is that it is not possible to handle exceptions that may have a specific

[7] https://github.com/h-richard/StateMachine.

[8] There are 9 EMF and related JAR files to add to a project. We provide the part of the pom.xml file for integrating these dependencies into a Java Maven project: https://github.com/h-richard/Xmodeling_Studio/blob/main/pom.xml

[9] https://github.com/h-richard/Spice_Unity_View

```
1  public void doStep() {
2      try {
3          this.position =
4              calcs.calculateNextPos(this.position,
5                                     this.windDirection,
6                                Clock.getStepTime * 1000);
7      }
8      catch (NoChange ex1) {
9          return; // skip
10     }
11     catch (Exception ex2) {
12         ext.handleException(ex2);
13         System.exit(); // exit
14     }
15     try {
16         calcs.checkForCollision(this.position,
17                                 this.otherDronePosition);
18     }
19     catch (OtherDroneIsClose eX3) { } //  continue
20     catch (DronesCollided eX4) {
21         // do not know how to make a local stop ...
22     }
23 }
```

Listing 6 excerpt of the *doStep()* method in the *Drone* class within CARES

impact on the simulation semantics. If the skip, exit or continue reactions can be carried out (respectively lines 8, 12 and 18 in listing 6), the specific local stop reaction cannot be managed for removing a component from the simulation.

The reason is that the CARES *doStep()* generic method is not designed to return anything, and the simulation loop generated is not designed in consequence to handle such a return. With SPICE, adding support for this local stop is very simple: the language engineer needs to add the value in an enumeration and simply adds the lines 10, 20 and 21 of the listing 1 in the code of the execution engine (it requires also to declare the *stopped* variable). There are only 5 lines of code to add. With CARES, the engineer has to manually edit the signature of the *doStep()* method to throw an exception and to add, in the generated code of the simulation loop, the management of this exception. This adds up to 16 lines to modify in 4 Java classes that already contains a total of 375 generated lines of code. Then, once this code has been validated, the engineer has to modify CARES' Acceleo code generator (3 files for a total of 341 lines of MTL code) so that this code will now be generated from a model. This therefore requires significant and complex modifications to integrate this specific behavior.

5.5. Agility in software configuration

We illustrate here how our approach bring agility in software development, here simulator.

SPICE has been used to develop several successive versions of the drone simulator presented in the paper. In an initial version, a simple wind behavior was considered, with constant direction and speed. This version was used to identify the priority levels and the required dataflow semantics of each component. This was achieved by running 8 possible configurations obtained by different combinations of priorities and compliance with a dataflow semantics. In particular, we showed that the wind component must have a higher priority than the drone components, and that drones components respectively comply with a dataflow execution semantics while the wind component does not.

Subsequent three versions were then developed to determine the execution frequency required to achieve a realistic simulation (here, 500 ms). Once these simulation characteristics were established, a more realistic version incorporating variable wind behavior was implemented.

All these variants were obtained by modifying the models only—namely simulation and component properties[10] and the Java method calls associated with different wind behaviors[11]—thereby demonstrating the relevance of the proposed approach for rapidly configuring and evolving simulation software.

6. Conclusion

In this paper, we propose an approach for developing software by combining an executable model with standard code. To do this, we provide an EMF plugin, Xmodeling Studio, which allows a set of metaclasses to be added to the metamodel of an xDSL, enabling operation calls to be added to the elements of its models. We also provide the EMF-code of these metaclasses in order to automatically execute within the execution engine of the xDSL the Java methods corresponding to the operation calls defined in the model.

The approach enables rich model–code interactions: elements of the model can be passed as parameters to Java methods and an element of the model can be modified with the result of a Java method. It also provides a fine-grained exception management whose policies are defined in the model.

The approach has been applied to the development of a drone simulator, demonstrating its usefulness in facilitating software configuration, particularly for experimenting with different behaviors corresponding to the integration of various existing Java code bases.

One of the main forthcoming extensions of Xmodeling Studio is to enable explicit typing of the objects used in the definition of operation calls within models, so that type checking can be fully performed at model-editing time. Currently, it is not possible to entirely ensure during the model edition that an operation call definition is valid.

In the current version of the tool, the proposed solution for type alignment may require tedious manual adaptations. We are therefore working on a more generic solution in which the code generated by the plugin provides a set of conversion methods for defining generic type adapters.

We will incorporate the use of Xmodeling Studio into our MDE courses at the university of Brest. This will allow us to gather feedback on this model/code development paradigm for software developers. It will help us in improving the approach and the tool.

[10] The timing configuration of the simulation is changed on line 1 of the model (listing 3). The priority and the data-flow semantics of a component is changed at the first line of its definition (lines 2, 24 and 29).

[11] Changing the Java method calculating the wind behavior simply consists in modifying the call statement on line 33 of the model (listing 3).

About the authors

Hugo Richard is PhD student in software engineering at the university of Brest / Lab-STICC, France. You can contact the author at hugo.richard@univ-brest.fr.

Eric Cariou is assistant professor in software engineering at the university of Brest / Lab-STICC, France. You can contact the author at eric.cariou@univ-brest.fr or visit https://lab-sticc.univ-brest.fr/~ecariou/.

Jean-Philippe Babau is full professor in software engineering at the university of Brest / Lab-STICC, France. You can contact the author at jean-philippe.babau@univ-brest.fr or visit https://lab-sticc.univ-brest.fr/~babau/.

References

Badreddin, O., & Lethbridge, T. C. (2013). Model oriented programming: Bridging the code-model divide. In *5th International Workshop on Modeling in Software Engineering (MiSE)*. doi: 10.1109/MiSE.2013.6595299

Barbier, F. (2016). *Reactive Internet Programming – State Chart XML in Action*. the Association for Computing Machinery and Morgan & Claypool. doi: 10.1145/2872585

Barbier, F., & Cariou, E. (2019). Executable Modeling for Reactive Programming. In *Model-Driven Engineering and Software Development (MODELSWARD 2018)* (Vol. 991, pp. 1–8). Springer. doi: 10.1007/978-3-030-11030-7_1

Breton, E., & Bézivin, J. (2001). Towards an understanding of model executability. In *Proceedings of the international conference on Formal Ontology in Information Systems (FOIS '01)* (pp. 70–80). ACM. doi: 10.1145/505168.505176

Cariou, E., Brunschwig, L., Le Goaer, O., & Barbier, F. (2020). A software development process based on UML state machines. In *The 4th Edition of the International Conference on Advanced Aspects of Software Engineering (ICAASE'20)* (pp. 1–8). IEEE. doi: 10.1109/ICAASE51408.2020.9380117

Cariou, E., Le Goaer, O., & Barbier, F. (2016). On the Executable Nature of Models. In *2nd International Workshop on Executable Modeling at MoDELS (EXE 2016)* (Vol. 1760). CEUR Workshop Proceedings. Retrieved from https://ceur-ws.org/Vol-1760/paper8.pdf

Cariou, E., Le Goaer, O., Brunschwig, L., & Barbier, F. (2018). A generic solution for weaving business code into executable models. In *The 4th International Workshop on Executable Modeling at MODELS (EXE 2018)* (Vol. 2245). CEUR Workshop Proceedings. Retrieved from https://ceur-ws.org/Vol-2245/exe_paper_2.pdf

Clarke, P. J., Wu, Y., Allen, A. A., Hernandez, F., Allison, M., & France, R. (2013). Formal and Practical Aspects of Domain-Specific Languages: Recent Developments. In M. Mernik (Ed.), (chap. 9: Towards Dynamic Semantics for Synthesizing Interpreted DSMLs). IGI Global. doi: 10.4018/978-1-4666-2092-6

Combemale, B., Crégut, X., Garoche, P.-L., & Xavier, T. (2009). Essay on Semantics Definition in MDE – An Instrumented Approach for Model Verification. *Journal of Software*, *4*(9). Retrieved from https://hal.science/hal-00371553

De Maio, M. N., Salatino, M., & Aliverti, E. (2014). *jBPM6 Developer Guide*. Packt Publishing Ltd. (isbn: 9781783286621)

Guérin, S., Polet, G., Silva, C., Champeau, J., Bach, J.-C., Martinez, S., … Beugnard, A. (2021). PAMELA: an annotation-based Java Modeling Framework. *Science of Computer Programming*, *210*. doi: 10.1016/j.scico.2021.102668

Guermazi, S., Tatibouet, J., Cuccuru, A., Seidewitz, E., Dhouib, S., & Gérard, S. (2015). Executable Modeling with fUML and Alf in Papyrus: Tooling and Experiments. In *1st International Workshop on Executable Modeling at MoDELS (EXE 2015)* (Vol. 1560, pp. 3–8). Retrieved from https://ceur-ws.org/Vol-1560/paper1.pdf

Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, *8*(3), 231-274.

Itemis. (visited July 2020). *YAKINDU Statechart Tools Web site*. Retrieved from https://www.itemis.com/en/yakindu/state-machine/

Lazăr, C.-L., Lazăr, I., Pârv, B., Motogna, S., & Czibula, I.-G. (2010). Tool Support for fUML Models. *International Journal of Computers Communications & Control*, *5*(5). doi: 10.15837/ijccc.2010.5.2237

Lethbridge, T. C., Forward, A., Badreddin, O., Brestovansky, D., Garzon, M., Aljamaan, H., … Zakariapour, A. (2021). Umple: Model-driven development for open source and education. *Science of Computer Programming*, *208*, 102665. doi: 10.1016/j.scico.2021.102665

Modelica Association. (2023). *Functional mock-up interface specification, version 3.0.1*. Retrieved from https://fmi-standard.org/docs/3.0.1/

Object Management Group. (2003). *MDA Guide Version 1.0.1*. Retrieved from http://www.omg.org/cgi-bin/doc?omg/03-06-01

Object Management Group. (2014). *Business Process Model and Notation (BPMN) Version 2.0.2*. Retrieved from https://www.omg.org/spec/BPMN/2.0.2/

Object Management Group. (2017a). *Action Language for Foundational UML (ALF) Specification, version 1.1*. Retrieved from https://www.omg.org/spec/ALF/1.1/

Object Management Group. (2017b). *Unified Modeling Language (UML) Specification, version 2.5.1*. Retrieved from https://www.omg.org/spec/UML/2.5.1/

Object Management Group. (2021). *Semantics of a Foundational Subset for Executable UML Models (fUML) Specification, version 1.5*. Retrieved from https://www.omg.org/spec/FUML/1.5

Salmon, L., Pillain, P.-Y., Guillou, G., & Babau, J.-P. (2024). NAVIDRO, a CARES architectural style for configuring drone co-simulation. *ACM Trans. Embed. Comput. Syst.*, *23*(3). doi: 10.1145/3651889

Theobald, M., & Tatibouet, J. (2019). Using fUML Combined with a DSML: An Implementation using Papyrus UML/SysML Modeler. In *7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2019)* (pp. 248–255). SciTePress. doi: 10.5220/0007310702500257