Towards an end-to-end metamodeling approach using rust

Léo Olivier, Marcos Didonet del Fabro, Chokri Mraidha, Sebastien Gerard

Towards an End-to-End Metamodeling Approach using Rust

Léo Olivier, Marcos Didonet Del Fabro, Chokri Mraidha, Sébastien Gérard
Université Paris-Saclay, CEA, List
F-91120, Palaiseau, France
firstname.lastname@cea.fr

***Abstract*—Domain-Specific Languages (DSLs) are specialized languages targeted to an application domain. EMF (Eclipse Modeling Framework) is the most popular DSL development framework, with Ecore as its metametamodel and Java as the target language. We can find the translation of Ecore and the implementation of subsets of EMF functionalities targeting different languages. One language that has raised interest is Rust, thanks to the possibility of producing reliable and secure programs that are energy-frugal and executable in web rower via WebAssembly. In this paper, we present our end-to-end approach for metamodeling using Rust. Our solution integrates a code generator from Ecore to Rust and a code interpreter of Ecore metamodels, which enables performing model manipulation operations (CRUD and serialization) within a Rust environment. We provide an implementation using Behavior Trees to validate our approach and discuss the main advantages and difficulties.**

***Index Terms*—Rust, end-to-end metamodeling, Ecore, multi-platform**

I. Introduction

In the face of the climate emergency, the solution is technological and behavioral. On the one hand, it is important to change our habits, because technology alone cannot save us. However, on the other hand, technology remains an important lever to help us combat global warming and achieve carbon neutrality as quickly as possible. Technological challenges are high and rely on multi-disciplinarity teams and the power of collective intelligence. The key to success underlying these two principles is knowledge, specifically the creation and sharing of knowledge, or even better, the federation of knowledge. Model-based engineering aligns with this approach and has already demonstrated its ability to improve knowledge management practices, particularly in a collaborative context, such as online collaborative modeling approaches.

A Domain-Specific Language (DSL) is a specialized language for an application domain. It is built around the technical vocabulary of the domain to increase software development productivity. Domain models produced using a DSL are relevant in the design phase and can constitute actual software artifacts since code can be directly generated from them [1], [2]. DSLs find applications in diverse contexts e.g.; CSS for web styling, SQL for database querying, Coq for formal proof, and Behavior Trees for robotics and computer games.

There exists a wide range of programming languages, frameworks, and language workbenches dedicated to the development of DSLs [3]. Examples of such tools include MetaEdit [4], the GEMOC Studio [5], Rascal [6], and MPS[1], among others. The Eclipse Modeling Framework (EMF) stands out as a popular environment for DSL development. It enables the efficient creation of metamodel-based abstract syntaxes using the Ecore language, which provides a rich ecosystem of model-based software and tools [7]. EMF primarily targets Java as the runtime environment: its code generation tool converts Ecore models into Java source code. The success of EMF encouraged the modeling community to implement Ecore in languages other than Java. For example, there is a C++[2] and Python[3] implementation of Ecore. Recently, a Rust code generation tool laid the groundwork for an implementation of Ecore in this language [8]. Our work provides an extended mapping, and it handles an end-to-end metamodeling chain using Rust.

If our priorities lie in aspects such as performance, energy consumption, bare metal architecture targeting (e. g. embedded systems), and ensuring auditability, safety, security, and certification, then opting for Rust becomes an interesting choice.

Rust is a multi-paradigm and general-purpose programming language which guarantees memory and thread safety without garbage collection through the ownership system. While being as low-level as C or C++, it offers high-level, zero cost abstractions such as traits, pattern matching, algebraic data types, closures, and iterators[4]. It produces high-reliability programs with excellent execution speed. Additionally, Rust is designed for a wide range of execution environments such as embedded systems, network services, command-line tools, and web applications through WebAssembly[5]. However, it should be noted that Rust is not built around a class taxonomy: the language does not feature abstract class or class inheritance.

All these features could be explored in a lightweight modeling environment based on Ecore targeting this low-level and safe language.

In this work, we present an approach to use Rust as a metamodeling engine and as a target language for metamodel-based abstract syntax code generation. This end-to-end metamodeling approach using Rust integrates two components:

[1]http://jetbrains.com/mps
[2]https://github.com/catedrasaes-umu/emf4cpp
[3]https://pyecore.readthedocs.io/en/latest/
[4]https://doc.rust-lang.org/book/
[5]https://www.rust-lang.org/fr/what

RustEcore and Ecore2rs. The first one allow to dynamically manipulate Ecore metamodels and models in Rust, simulating an Ecore infrastructure in memory like EMF does, and the second one generates Rust code from Ecore metamodels. We present an end-to-end implementation, using the Behavior Trees (BT) DSL as an illustrative example to showcase our methodology. Through this evaluation, we analyze the advantages and limitations of applying our approach to the BT scenario.

The rest of this paper is structured as follows. Section 2 introduces terms and concepts used in the paper. Furthermore, we explain how a Behavior Tree (BT) works. Section 3 gives an overview of our contribution and presents the main features of both tools, as well as their integration. Section 4 provides an evaluation of our approach. We discuss the results in Section 5 and the related work in Section 6. The conclusion in Section 7 sums up our project and proposes various improvement axes for our work.

II. Context

In this section, we define what is a metamodel-based DSL, its usage, and we motivate our approach with an illustrative example: Behavior Trees.

A. Metamodel-based DSL

A DSL consists of three main components: abstract syntax, concrete syntax, and semantics [9]. The abstract syntax formally defines the rules to create a syntactically valid model, while the concrete syntax determines how language elements are displayed and edited. Semantics specifies the interpretation of models as representations of or specifications for modeled systems. The DSL's abstract syntax is often defined using a metamodel, that is a set of rules that describes how models should be structured and what components they can contain [9].

Crafting a metamodel can be done with a metamodeling language such as Ecore, which is implemented as part of EMF. It allows the creation of metamodels in the form of a class diagram, with each class representing a concept from the application domain. Classes can have features, operations and references to other classes. They can be organized into packages[6].

B. Behavior Tree

We use Behavior Trees to illustrate our approach, which are tree structures often used in game development and robotics, to model a task or execution plan [10]. These domains are known to have high requirements in terms of optimization, either because of very resource-intensive operations (i. e. computer games), or because of the low resources of embedded systems. In the case of robotics, systems are often critical and failure is intolerable.

[6]https://www.eclipse.org/modeling/emf/docs/1.x/UG/EMF_v1.0_Users_Guide.html

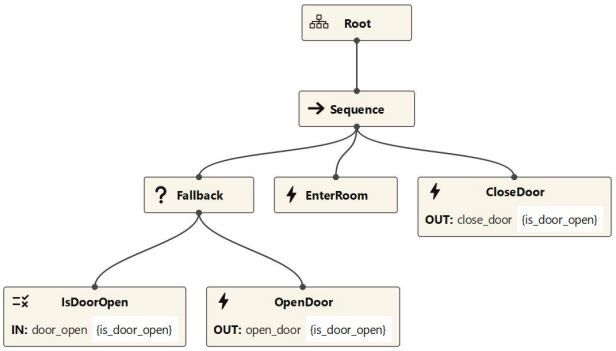

Fig. 1: A Behavior Tree model using Groot IDE [7]

1) Execution Semantics: Behavior Trees are composed of nodes and executed through a series of discrete update steps called *ticks*. Each time a BT is ticked, usually at a specified rate, its children nodes are ticked recursively based on the tree construction (sequential execution). Once a node is ticked, it reports its status to the parent node, which can either be *Success*, *Failure*, or *Running*. The Running status indicates that an asynchronous task is being processed.

2) Nodes: Behavior Tree nodes are of different types[8]. The Root node is the BT entry point: it has one child and no parent. Leaves are execution nodes, i. e. Action (task) or Condition. Condition nodes are usually a simple boolean check (e. g. *is the door open ?*) and cannot return the Running status. Control nodes are internal nodes and determine the traversal of the BT based on the status of their children. They can have any number of children. The main control nodes are Sequence and Fallback (sometimes called Selector). Sequence nodes execute children in order until one child returns Failure or all children return Success[9]. Fallback nodes execute children in order until one of them returns Success or all children return Failure. They are used to design recovery behaviors[10]. Finally, Decorator nodes apply a custom policy to a single child node, changing the status of the decorated node. For example, an Inverter decorator will change Success to Failure, and vice-versa[11].

3) Blackboard: Execution nodes have ports for manipulating a key/value storage called the Blackboard. Input ports can read an entry in the Blackboard, while output ports can write to an entry.

4) Running Example: Figure 1 is a Behavior Tree model which describes the behavior of a robot traversing a room. It has three tasks (OpenDoor, EnterRoom, CloseDoor) and one condition (IsDoorOpen). A Boolean blackboard entry indicates whether the door is open or not. Assuming this Boolean is false, the execution sequence would be: Root, Sequence, Fallback, IsDoorOpen condition returns false, OpenDoor task changes the Boolean to true, EnterRoom task executes, CloseDoor task changes the Boolean to false.

[8]https://www.behaviortree.dev/docs/learn-the-basics/bt_basics/
[9]https://www.behaviortree.dev/docs/nodes-library/SequenceNode
[10]https://www.behaviortree.dev/docs/nodes-library/FallbackNode
[11]https://www.behaviortree.dev/docs/nodes-library/DecoratorNode

Obtaining such a behavior tree using a model-based approach involves designing the metamodel, generating the code, and using it to instantiate models. Our approach aims to fullfill the first two steps, i. e. providing an end-to-end metamodeling environment using Rust.

III. End-to-End Metamodeling in Rust

In this section, we present the design of the approach and its envisioned use by language engineers. We motivate our design choices for both Ecore2rs and RustEcore tools and detail how they fit into the approach we propose.

A. Overview

Figure 2 depicts the presented solution. From left to right, it shows the creation of a DSL by the language engineer and its usage by the domain expert.

First, the language engineer defines the abstract syntax of the DSL in a metamodeling environment integrating the RustEcore framework. Several user interfaces can be used with this Ecore implementation. For example, the abstract syntax could be designed in a Web-based Graphical User Interface (GUI) and then debugged on the fly in a Terminal User Interface (TUI) later in the development cycle. The language engineer can design models and validate them in the environment to test and ensure that the metamodel meets their needs. Then, RustEcore exports the Ecore metamodel to a file in serializable format. The Ecore2rs tool translates it into a Rust-based abstract syntax, ready to use in a Rust program. Finally, the language engineer defines the operational semantics of the DSL in Rust and obtains a pure Rust DSL implementation. Relying on the different compilation toolchains that Rust offers, the DSL runtime can be deployed in several execution environments (e. g. the OS of the host machine, embedded systems, the Web via WebAssembly) to be used by the domain expert.

The following subsections detail the mechanisms of the two central pieces of our approach: the RustEcore and Ecore2rs tools.

B. RustEcore

RustEcore is a modeling framework written in Rust designed for dynamic creation and manipulation of Ecore metamodels and models. It serves as an Ecore metamodeling engine, interpreting models and facilitating seamless integration with diverse user interfaces. It offers the deserialization/serialization of Ecore metamodels in different formats (JSON, XMI, PlantUML) and integrates Ecore2rs as a dependency, allowing to generate the Rust code of a metamodel designed with the framework.

1) Metamodel Creation: RustEcore provides an API for programmatically designing metamodels using Ecore-like elements. Creating a metamodel with it involves defining `EClasses`, adding `EStructuralFeatures`, `EOperations`, and inheritance relationships to them. The `EClasses` are then bundled into an `EPackage`. Figure 3 shows how these elements are organized in the framework

```rust
let mut pkg = EPackage::new("bt");
let mut tree_node = EClass::new_abstract("TreeNode");
tree_node.add_feature(EStructuralFeature::new_unique(
    "ID", Type::EString // name, type
))?;
tree_node.add_feature(EStructuralFeature::new_optional(
    "name", Type::EString
))?;
tree_node.add_operation(EOperation::new(
    "tick", // name
    vec![], // params
    ReturnType::value_unique(Type::EEnum(status_enum_ref)),
))?;
pkg.add_eclass(Rc::new(RefCell::new(tree_node)))?;
```

Listing 1: `TreeNode` Class and Features Declaration Using RustEcore

and Listing 1 illustrates the use of the API to declare a class of the metamodel of Behavior Trees we propose in figure 5.`EClasses` relationships (including inheritance) are reproduced with references between the instances of the Rust structs that represent them. For example, inheriting a class from another is to add a reference to it in its `super_class` field. That is, the framework replicates the semantics of Ecore's object-oriented features that Rust does not support.

Our implementation relies on the *interior mutability*[12] pattern to handle relations. To avoid memory leaks, only the `EPackage` stores strong references (i. e. `Rc` memory container) to classifiers. `EClasses` among themselves have weak references. This avoids situations where classes with a two-way relationship are not deallocated after being removed from the package.

2) Model Creation and Validation: Model elements are represented by instances of `EObject`. As shown in figure 4, `EObject` has a reference to an `EClass`, a list of attribute values, i. e. a key/value list (the key being the name of the attribute), and a boolean `is_valid` indicating whether the `EObject` conforms to its `EClass` definition. RustEcore prevents the instantiation of an `EObject` that has a reference to an abstract class or interface. Model's classes and objects are both instances of Rust structs and are therefore at the same level of the MOF hierarchy (M0). Thus, the class/instance relationship is emulated and the language's typing cannot enforce the construction of a valid `EObject`. This validation process is handled by RustEcore. On instantiation, an `EObject` is declared invalid. As shown in Listing 2, validation is performed by calling the `validate()` method on the object, ensuring that all attributes (including inherited ones) are set to correct values that respect the constraints defined in the metamodel (concerning cardinalities, type of value, the memory container).

3) Notification system: RustEcore integrates a notification system to listen to changes made to an element. It is designed to adapt to various use cases related to user interfaces. One such application involves tracking all changes made to models, enabling a history feature that captures their evolution over time. Another example would be the case of a distributed

[12] https://doc.rust-lang.org/book/ch15-05-interior-mutability.html

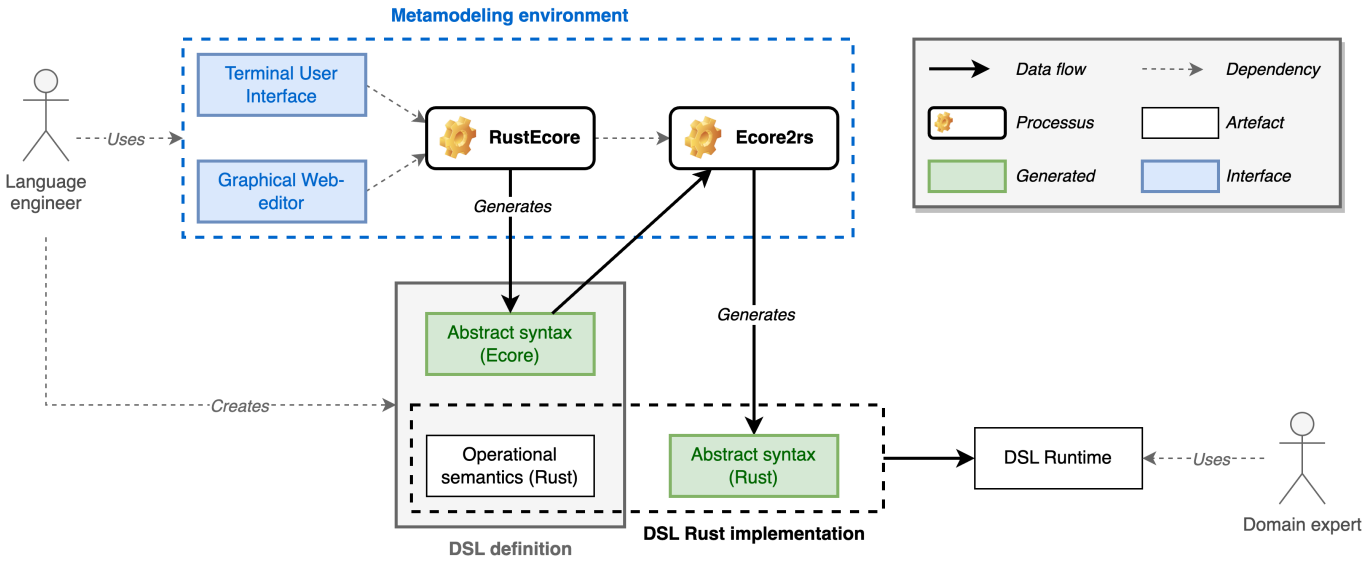

Fig. 2: Approach Overview

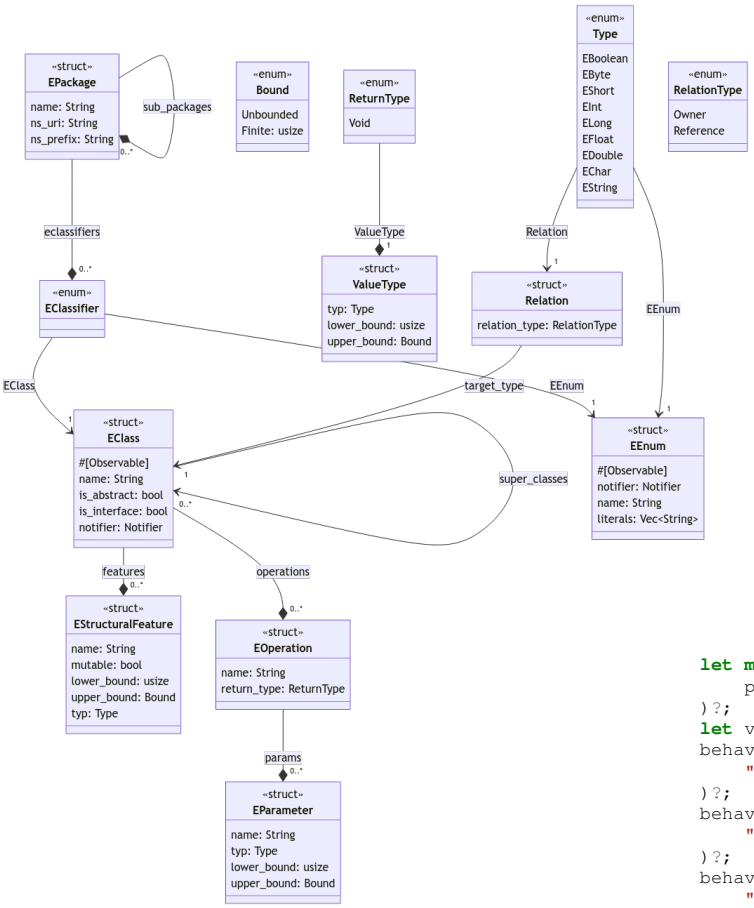

Fig. 3: Diagram of RustEcore for Creating Metamodels

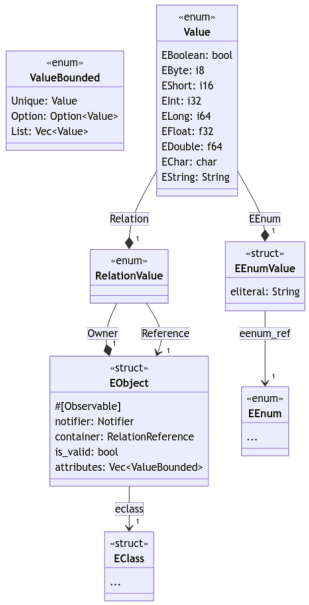

Fig. 4: Diagram of RustEcore for Creating Models

```rust
let mut behavior_tree = EObject::new(
    pkg.eclass_ref("BehaviorTree")? // EClass type
)?;
let validate = true;
behavior_tree.set_unique(
    "ID", Value::from("BT1"), validate
)?;
behavior_tree.set_unique(
    "rootnode", Value::from(sequence), validate
)?;
behavior_tree.set_unique(
    "blackboard", Value::from(blackboard), validate
)?;
behavior_tree.validate()?;
```

Listing 2: Model Building using RustEcore

system where users receive notifications about state changes in a shared collaborative model.

An action can be triggered by a closure depending on the type of event. Table I summarizes the event list. The implementation of the notification system is based on the observer pattern. An observable object implements the `Observable` trait (which is automatically implemented using a macro) and contains a list of adapters. An adapter is a closure comprising the notifying object, optionally the attribute where the state change occurred, and the type of event received. Figures 3 and 4 show that `EClass` and `EObject` implement Observable by default.

Rust enforces several strict rules regarding shared references and the lifetime of values[13]. So instead of having a reference to a Listener, an Observable register closures (adapters) that fire on new notifications. It guarantees at compile time that no operations will be performed on the Listener outside of the Observable. While we prohibit adapters from capturing references, a language engineer will be able to use smart pointers[14] as needed.

Some default listeners are already implemented. For example, if an `EClass` is deleted from the metamodel, the system automatically manages the deletion of all references (including inheritance relationships) to this element, preventing access to broken references.

Event name	Feature type	Change type
Set	Unique	The value has changed
Unset	Optional	The value has been unset
Add	List	A value has been added
AddMany	List	Many values have been added
Remove	List	A value has been removed
RemoveMany	List	Many values have been removed
Move	List	A value has been moved inside a feature
Resolve	Any	A feature has been resolved from a proxy
RemoveAdapter	Any	An adapter has been removed
RemoveRef	Any	The referenced type has been removed

TABLE I: Type of Event That Can Be Notified Depending on the Feature Type

C. Ecore2rs

Ecore2rs is a code generator written in Rust that translates Ecore metamodels to Rust. The tool can be used in two ways: as a command-line tool, where it takes an Ecore metamodel as input and generates a Rust file, or as a Rust procedural macro. In the latter case, it takes the path to a metamodel as input and directly produces the resulting code as token trees within the Rust compiler [11].

1) Translation Into Rust: Table II presents the mapping between Ecore and Rust types in `EStructuralFeatures` [8]. `EClass` translation uses the *interior mutability* pattern[15] and a *trait object*[16].

[13]https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html
[14]https://doc.rust-lang.org/book/ch15-00-smart-pointers.html
[15]https://doc.rust-lang.org/book/ch15-05-interior-mutability.html
[16]https://doc.rust-lang.org/book/ch17-02-trait-objects.html
[17]Interfaces in Ecore are represented by abstract `EClasses` and marked as interfaces.

Ecore	Rust
`EByte`	`i8`
`EShort`	`i16`
`EInt`	`i32`
`ELong`	`i64`
`EFloat`	`f32`
`EDouble`	`f64`
`EBoolean`	`bool`
`EChar`	`char`
`EString`	`String`
`EEnum T`	`T`
`EClass T`	`Rc<RefCell<dyn TLike>`

TABLE II: `EStructuralFeature` Type Conversion Table

Ecore feature bounds	Rust type
1	`T`
0 or 1	`Option<T>`
More than 1	`Vec<T>`

TABLE III: `EStructuralFeature` Bounds Conversion Table

Table III shows which data structures are used to convert features based on their arity. Table IV summarizes the encoding of Ecore elements in Rust.

Ecore2rs provides a systematic encoding of the object-oriented aspects of Ecore metamodels, specifically addressing the inheritance relation between classes, including multiple inheritance.

The Ecore inheritance system has three aspects which need to be addressed: feature inheritance, operation inheritance, and subtyping [12]. Feature inheritance means that a subclass contains all of its superclasses' data fields (i. e. attributes and references) recursively in addition to its own. On the other hand, operation inheritance is about methods: a subclass contains all of its superclasses' methods recursively in addition to its own. The implementation of a given method is that of the closest parent (including itself) providing an implementation for it. Finally, subtyping is the ability to substitute the actual type of a value with the type of one of its superclasses (i. e. polymorphism).

Feature inheritance is achieved with a Rust `struct`, i. e. a data structure containing data fields. This means that a subclass has, for each of its superclasses, a field to store an instance of it. Therefore, it is possible to access, by induction, the inherited fields from all of its superclasses, recursively. Listing 3 shows the code resulting from the conversion of two classes of the metamodel of Behavior Trees we propose in figure 5. Properties inherited from `TreeNode` by `Decorator` are stored in the `inst_tree_node` field.

Operation inheritance consists in abstracting a class by its functionalities, which includes operations defined for this class

Ecore	Rust
`EEnum T`	`Enum T`
`EClass T` (including abstract)	`struct T` + `trait TLike`
`EInterface T` [17]	`trait TLike`

TABLE IV: Data Structure Conversion Table

```rust
pub struct TreeNode {
    pub id: String,
    pub name: Option<String>,
}

pub struct Decorator {
    pub inst_tree_node: TreeNode,
    pub treenode: Rc<RefCell<dyn TreeNodeLike>>,
}
```

Listing 3: Feature Inheritance Example Featuring the Abstract Class `Decorator` Inheriting the Abstract Class `TreeNode`

and getters/setters to access its properties. It is achieved by using two traits: `TLike`, which is generated for each class `T` in the metamodel, and `AsInstanceOf<T>`, which use a generic parameter to convert one type to another or to itself, getting a reference to that target type. Traits define a set of methods that can be implemented by types, allowing code reuse and polymorphism, similar to interfaces in object-oriented programming languages. Listing 4 shows the trait `DecoratorLike` generated for `Decorator`. To implement `TLike`, `T` must implements `AsInstanceOf` for all its superclasses and itself. This is automatically generated and allows providing a default implementation for getters/setters in `TLike` by accessing the properties of the `T` struct, including inherited properties.

To achieve inheritance of regular method implementation, each subclass `T` must retrieve the implementation of the method in its superclass `SupT` within its `TLike` trait. As a result, `T` implements all its parent traits `SupTLike`, which is allowed because `T` implements `AsInstanceOf<SupT>` for all its superclasses.

However, in the top parent class of the inheritance hierarchy, the method implementation does not exist because Ecore metamodels only specify the method signature and not whether a method is abstract or not. Ultimately, methods must be implemented manually by the developer, who is the only one who knows where the method is concrete in the inheritance hierarchy. Therefore, the implementation of the traits `TLike` for the structs `T` cannot be generated and the resulting code does not compile directly.

The developer has to implement the top parent class methods when implementing `SupClassLike` for `SupClass`[18]. Subclasses inherit these implementations. They can be overridden by the developer by supplying a new one when implementing `SubClassLike` for `SubClass`.

Finally, subtyping consists in abstracting a value of type `T` to a `dyn SupTLike`, that is any type implementing the `SupTLike` trait. Since `T` implements all `SupTLike` traits of its superclasses, calling a method on `dyn SupTLike` will resolve by the runtime to the actual type of the value (dynamic dispatch) and use the definition from the `TLike` implementation for `T` (which can be inherited or overridden).

2) Customizing Code Generation: Rust is a low-level system language with a strong type system, which requires explicit specification of mutability at the type level. To offer Ecore designers greater control over code generation, Ecore2rs incorporates an annotation framework. It is based on Ecore annotations (i. e. `EAnnotation`), which allows attaching key/value pairs to packages, classes, features, references, and operations. With this framework, designers can customize code generation aspects such as parameter mutability and choice of memory containers for storing values.

[18]Abstract methods must have an implementation but can be expressed using `panic!`, which terminates the program when called.

```rust
pub trait DecoratorLike
where
    Self: for<'a> AsInstanceOf<'a, Decorator>,
{
    fn treenode(&self) -> &Rc<RefCell<dyn TreeNodeLike>> {
        &self.as_inst_ref().treenode
    }
    fn id(&self) -> &String {
        TreeNodeLike::id(&self.as_inst_ref().inst_tree_node)
    }
    fn name(&self) -> &Option<String> {
        TreeNodeLike::name(&self.as_inst_ref().inst_tree_node)
    }
    fn tick(&self) -> Status {
        TreeNodeLike::tick(&self.as_inst_ref().inst_tree_node)
    }
    // For clarity, setters have been omitted.
}
```

Listing 4: Operation Inheritance Example Featuring the Generated Trait `DecoratorLike`, Exposing the `Decorator` Functionalities

IV. Implementation and Experiments

In this section, we present a first evaluation of our approach using a DSL of Behavior Tree. We consider the following research questions:

- *RQ 1*: Does the approach allow end-to-end design of DSL in Rust?
- *RQ 2*: Does RustEcore fit into various user interfaces and address interoperability challenges with other standards and modeling formats?
- *RQ 3*: Does the code generated by Ecore2rs conform to the semantics of the Ecore metamodel and how much manual implementation effort does it avoid?

A. Experiment

Our evaluation protocol follows the steps for creating a DSL, from designing the abstract syntax to implementing a model. First, we create an Ecore metamodel of Behavior Tree programmatically with RustEcore. We use two user interfaces, namely a TUI and a Web-based GUI, both integrating RustEcore, to modify and visualize the metamodel. Then, we generate the Rust abstract syntax from the metamodel. We implement the operational semantics in Rust, thus obtaining a Rust implementation of the DSL. Finally, we recreate the model shown in figure 1 to demonstrate that our implementation works correctly.

1) Metamodel: Figure 5 shows the metamodel of Behavior Trees we propose, based on the retro-engineering works of *Ghzouli et al.* [10] and the implementation of the BehaviorTree.CPP library[19]. This library is used in the

[19]https://www.behaviortree.dev/

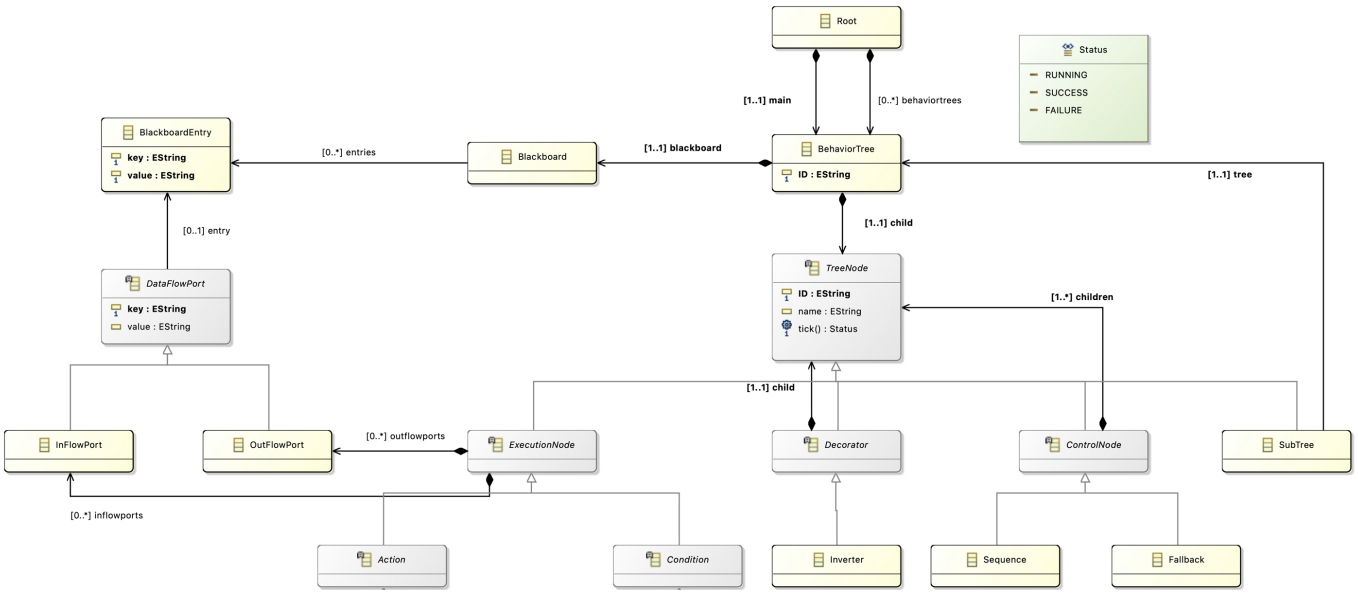

Fig. 5: A Behavior Trees Metamodel Using the EMF Graphical Editor

Papyrus4Robotics modeling development environment[20] [13], [14].

The metamodel exposes only the main subset of nodes needed to run a viable Behavior Tree. Control nodes such as Parallel or decorator nodes such as RepeatNode have been excluded for clarity of the model.

BT's `tick()` function is asynchronous, but Ecore gives no way to express it. This problem could be circumvented by using `EAnnotations` to mark the methods that need to be async during generation. However, such an `EAnnotation` is currently not implemented in Ecore2rs because async functions in traits are not yet stabilized in Rust. As of the time of writing, this feature is expected to arrive by the end of 2023[21]. We ended up to manually modifying the generated code with the help of the widely-used `async_trait` crate[22].

2) Abstract syntax definition: We programmatically define the Ecore metamodel with RustEcore using the API exposed by the framework. We evaluate its accuracy in a TUI integrating RustEcore, which offers options to modify the metamodel and visualize it in PlantUML format. We also load the serialized metamodel into a Web-based GUI called RustEcoreWeb, allowing the metamodel to be viewed as a class diagram. RustEcoreWeb uses Rust's WebAssembly compilation target to run on the Web. These interfaces are illustrated in figures 6 and 7 respectively.

3) Operational Semantics Implementation: All the interfaces presented previously have an option to generate the abstract syntax from the metamodel with Ecore2rs. The resulting code does not compile directly. We have to implement `TLike` traits for each `T` struct. Listing 5 shows the `tick()` method implementation for the Sequence node.

[20]https://www.eclipse.org/papyrus/components/robotics/

[21]https://blog.rust-lang.org/inside-rust/2023/05/03/stabilizing-async-fn-in-trait.html

[22]https://crates.io/crates/async-trait

```
Ecore input file provided. Deserializing EPackage...
Ecore EPackage deserialized in: 363.208µs
? What do you want to do?
> Display EPackage
  Modify EPackage
  Add new EClass
  Add new EEnum
  Save as XMI
  Generate code
v Display an EClassifier
[↑↓ to move, enter to select, type to filter]
```

Fig. 6: A Terminal User Interface integrating RustEcore

```rust
#[async_trait(?Send)]
impl SequenceLike for Sequence {
    async fn tick(&self) -> Status {
        for treenode in self.treenodes() {
            let status = treenode.tick().await;
            if status != Status::Success {
                return status;
            }
        }
        Status::Success
    }
}
```

Listing 5: Operational Semantics Implementation for the Sequence Node of Behavior Trees

4) Model Creation and Execution: We create the BT model depicted in figure 1 in Rust using the generated abstract syntax and implemented operational semantics. Listing 6 shows some elements of the model. The OpenDoor action has the "`open_door`" ID, no name and input ports, and an ouput port named "`opening_door`", allowing to modify the value of the Blackboard entry specifying if the door is closed or open. The Root node contains the Sequence node as the first execution node, a Blackboard, and no subtree.

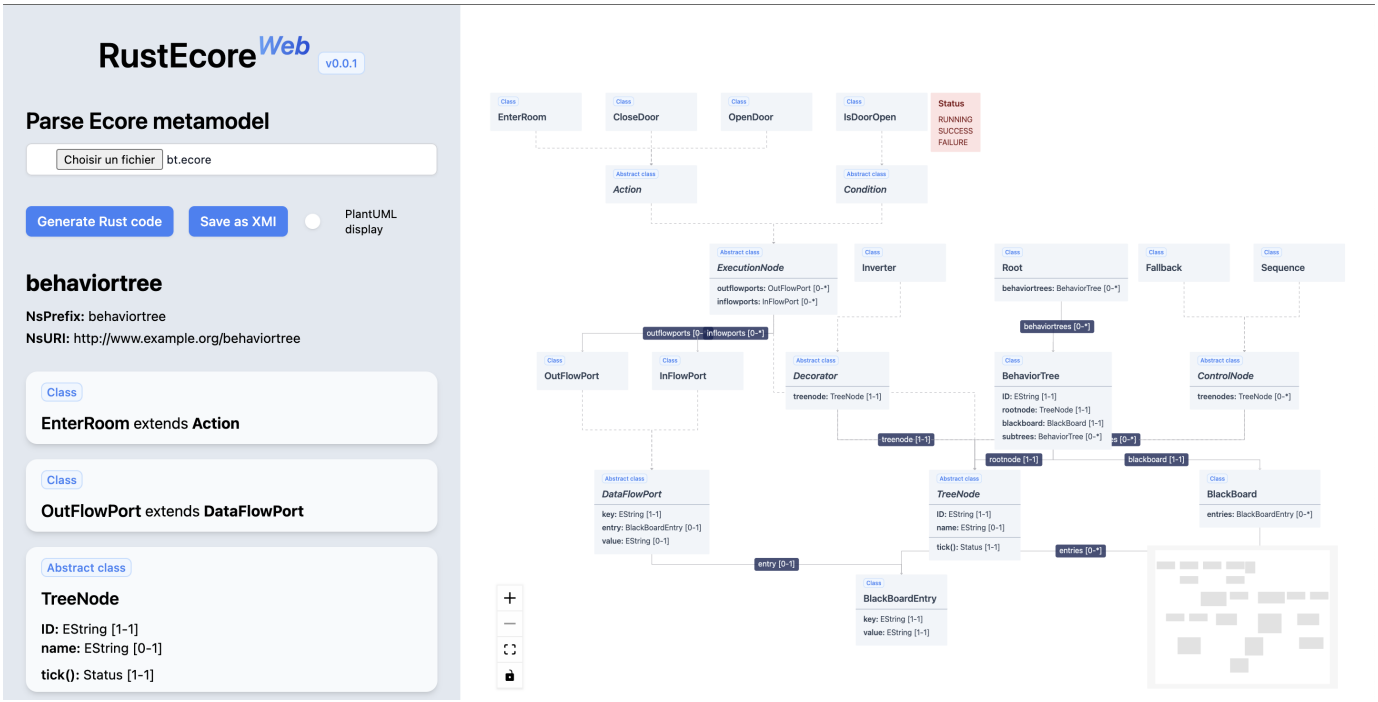

Fig. 7: A Web-Based Graphical User Interface Integrating RustEcore

```
...
let open_door = OpenDoor {
    inst_action: Action::new("open_door".to_string(), None,
        vec![], vec![
        OutFlowPort {
            inst_data_flow_port: DataFlowPort {
                name: "opening_door".to_string(),
                entry: Some(Rc::clone(&bb_entry_ref)),
                value: None,
            },
        }
    ]),
};
...
let seq = Sequence {
    inst_control_node: ControlNode {
        inst_tree_node: TreeNode {
            id: "seq".to_string(),
            name: None,
        },
        treenodes: vec![
            Box::new(fallback),
            Box::new(enter_room),
            Box::new(close_door),
        ],
    },
};

Root {
    behaviortrees: vec![BehaviorTree {
        id: "bt".to_string(),
        rootnode: Box::new(seq),
        blackboard,
        subtrees: vec![],
    }],
}
```

Listing 6: Elements of Implementing a Behavior Tree Model Using the Generated Abstract Syntax

```
Executing behavior tree...
Tick!
Entering in a sequence node
Entering in a fallback node
  Door is closed
Executing `OpenDoor` action
  Door has been opened
Executing `EnterRoom` action
  Going inside...
Executing `CloseDoor` action
  Door has been closed
Behavior tree execution returned status: Success
```

Fig. 8: Execution Log of the Behavior Tree Model Shown in Figure 1

The execution logs for this model are shown in figure 8 and match the execution description made in Section 2.2.4, thus validating our implementation.

B. Discussions

In this subsection, we discuss the results and outline potential challenges and encourage further exploration and refinement of the tools in future research endeavors.

1) RQ 1: The approach[23] was applied successfully for end-to-end metamodeling: we designed the abstract syntax, then obtained the corresponding Rust code and finally produced a Rust DSL of the BTs. The DSL is a tree-structure, but it uses a large part of metamodeling concepts. To avoid implementing all elements from scratch, we integrated a translator from Rust to Ecore with our Rust Ecore editor. The utilisation of the Ecore metamodel as a basis enabled interoperability with EMF. However, the translation from an object oriented modeling

[23]The resulting implementation will be published on an open source project.

framework to a functional one is not straightforward, due to conceptual mismatch.

2) RQ 2: RustEcore has been successfully integrated in two user interfaces. Interoperability is achieved through a XMI (de)serializer, allowing to load Ecore metamodels designed in other modeling environments. RustEcore partially reimplements Ecore: it lacks some features of the metamodeling language, such as generic type parameters, subpackages, custom `EDataTypes`, and `EAnnotations`. Since Rust has no reflection, the framework cannot provide the same API for manipulating model data reflexively as EMF.

3) RQ 3: The code generated by Ecore2rs respects the semantics of Ecore, but some limitations should be noted. First, Ecore2rs cannot resolves packages other than the current one, including nested packages. Second, abstract classes in the metamodel are treated as concrete classes, allowing the instantiation of structs that represent abstract classes. Another minor limitation is that Ecore2rs does not support type parameters, including bounded polymorphism. Finally, it has restrictions on down/upcasting capabilities, i. e. type conversions between related classes in an inheritance hierarchy. These restrictions are related to the Rust compiler and prevent casting a `dyn ClassLike` to a `dyn SupClassLike`, even if the former inherits from the latter. Indeed, they are fundamentally of different types and do not share the same virtual table. Potential solutions are being evaluated: trait upcasting coercion is experimented in the Rust compiler[24] and the `downcast-rs`[25] crate allows downcasting, although some edge cases are not supported.

In terms of avoided manual implementation effort, the resulting DSL behavior tree consists of 3 362 lines of code, of which 97.5% is generated code.[26] Indeed, reproducing Ecore's inheritance semantics requires a significant amount of generated code.

V. Related Work

The PyEcore framework provides a Python implementation of EMF, allowing to manipulate metamodels and Ecore models with this language.[27]

Ecore2rs builds on preliminary work of *Oliver et al.*'s, which focused primarily on the structural encoding of Ecore to Rust [8] (i. e. mapping Ecore primitive types, reference, class, and enum to Rust). It extends this work significantly by proposing a systematic encoding of the object-oriented aspects of Ecore meta-models, including inheritance.

The EMF framework provides three main building blocks: core EMF, EMF.edit and EMF.Codegen (Models, Adapters and Editors). Our work enables model manipulation operations with a notification system, and code generation for model and metamodel manipulation operations. Extensions for generating editors and adapters could be part of future work. *Gonzalez-Perez et al.* have worked on a formal definition of Behavior Trees through a metamodel [15]. Our metamodel is similar, which has been used as the primary use case.

[24]https://doc.rust-lang.org/beta/unstable-book/language-features/trait-upcasting.html

[25]https://crates.io/crates/downcast-rs

[26]This metric was obtained with the CLOC tool: https://github.com/AlDanial/cloc

[27]https://pyecore.readthedocs.io/en/latest/

VI. Conclusion

We presented an end-to-end metamodeling approach to create DSLs using Rust. The approach integrates two tools: RustEcore, an ongoing implementation of the Ecore metamodeling language in Rust allowing to dynamically define metamodels and models, and Ecore2rs, a code generator written in Rust that translates Ecore metamodels to Rust. They make it possible to design and then automatically generate the DSL abstract syntax. To evaluate our solution, we applied it to the DSL of Behavior Trees, which follows a tree-structure, but it uses a large part of metamodeling concepts.

We consider different directions of research to continue our work. Both of our tools need to be extended to support all of Ecore's features. RustEcore could support model (de)serialization and Ecore2rs the generation of their code. Additionally, it would be interesting to generate part of the operational semantics with Ecore2rs.

References

[1] M. Fowler, *Domain-specific languages*. Pearson Education, 2010.

[2] M. Brambilla, J. Cabot, and M. Wimmer, "Model-driven software engineering in practice," *Synthesis lectures on software engineering*, vol. 3, no. 1, pp. 1–207, 2017.

[3] S. Erdweg, T. van der Storm, M. Völter, L. Tratt, R. Bosman, W. R. Cook, A. Gerritsen, A. Hulshout, S. Kelly, A. Loh, G. Konat, P. J. Molina, M. Palatnik, R. Pohjonen, E. Schindler, K. Schindler, R. Solmi, V. Vergu, E. Visser, K. van der Vlist, G. Wachsmuth, and J. van der Woning, "Evaluating and comparing language workbenches: Existing results and benchmarks for the future," *Computer Languages, Systems Structures*, vol. 44, pp. 24–47, 2015, special issue on the 6th and 7th International Conference on Software Language Engineering (SLE 2013 and SLE 2014). [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1477842415000573

[4] S. Kelly, K. Lyytinen, and M. Rossi, "Metaedit+ a fully configurable multi-user and multi-tool case and came environment," in *Advanced Information Systems Engineering*, P. Constantopoulos, J. Mylopoulos, and Y. Vassiliou, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 1–21.

[5] E. Bousse, T. Degueule, D. Vojtisek, T. Mayerhofer, J. Deantoni, and B. Combemale, "Execution framework of the gemoc studio (tool demo)," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 84–89. [Online]. Available: https://doi.org/10.1145/2997364.2997384

[6] P. Klint, T. van der Storm, and J. Vinju, "Rascal: A domain specific language for source code analysis and manipulation," in *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, Sep. 2009, pp. 168–177.

[7] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*. Pearson Education, 2008.

[8] L. Olivier, L.-A. Sauvětre, E. Bousse, and G. Sunyé, "A tool-assisted approach to engineer domain-specific languages (dsls) using rust," in *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, ser. MODELS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 712–721. [Online]. Available: https://doi.org/10.1145/3550356.3563133

[9] A. Kleppe, *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*, 1st ed. Addison-Wesley Professional, 2008.

[10] R. Ghzouli, T. Berger, E. B. Johnsen, S. Dragule, and A. Wasowski, "Behavior trees in action: A study of robotics applications," in *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 196–209. [Online]. Available: https://doi.org/10.1145/3426425.3426942

[11] J. Blond, A. Carcano, and A. Champion, "Compiling ecore meta-models to rust," OcamlPro, Tech. Rep., 2023.

[12] W. R. Cook, W. Hill, and P. S. Canning, "Inheritance is not subtyping," in *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '90. New York, NY, USA: Association for Computing Machinery, 1989, p. 125–135. [Online]. Available: https://doi.org/10.1145/96709.96721

[13] A. Radermacher, M. Morelli, M. Hussein, and R. Nouacer, "Designing drone systems with papyrus for robotics," in *Proceedings of the 2021 Drone Systems Engineering and Rapid Simulation and Performance Evaluation: Methods and Tools Proceedings*, ser. DroneSE and RAPIDO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 29–35. [Online]. Available: https://doi.org/10.1145/3444950.3444956

[14] S. Gérard, C. Dumoulin, P. Tessier, and B. Selic, "Papyrus: A UML2 Tool for Domain-Specific Language Modeling Model-Based Engineering of Embedded Real-Time Systems," in *MBEERTS 2007 - International Dagstuhl Workshop on Model-Based Engineering of Embedded Real-Time Systems*, ser. Model-Based Engineering of Embedded Real-Time Systems, H. Giese, G. Karsai, E. Lee, B. Rumpe, and B. Schätz, Eds., vol. LNCS-6100. Dagstuhl Castle, Germany: Springer Berlin Heidelberg, Nov. 2007, pp. 361–368. [Online]. Available: https://inria.hal.science/hal-00943540

[15] C. Gonzalez-Perez, B. Henderson-Sellers, and G. Dromey, "A meta-model for the behavior trees modelling technique," in *Third International Conference on Information Technology and Applications (ICITA'05)*, vol. 1, July 2005, pp. 35–39 vol.1.